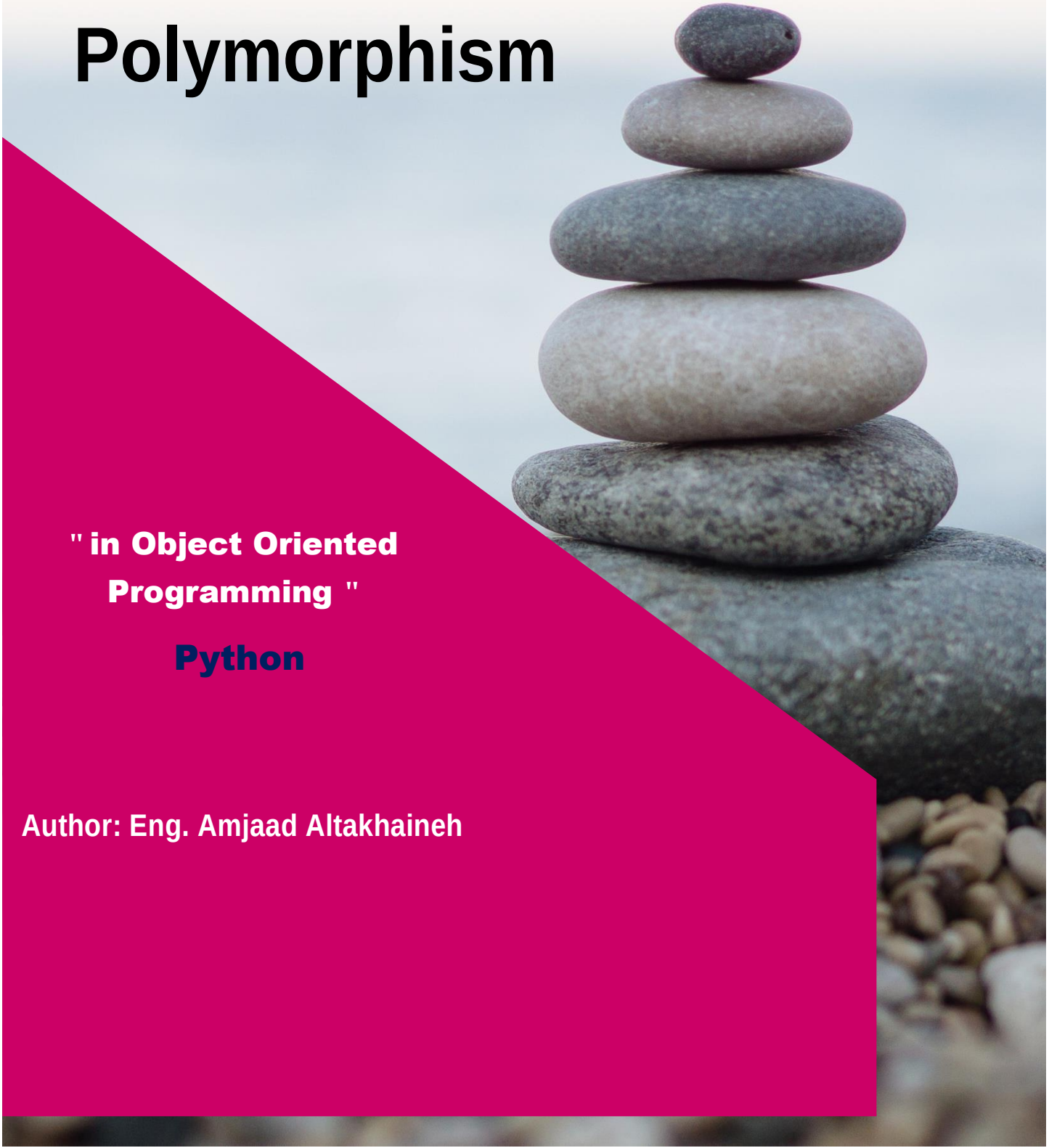
Inheritance and Polymorphism



" in Object Oriented Programming "

Python

Author: Eng. Amjaad Altakhaineh

keywords: inheritance, single inheritance , multiple inheritance, multiple inheritance , polymorphism,

1- WHAT IS THE PYTHON LANGUAGE?

Python is an extremely efficient language: your applications will do more in fewer lines of code than many other languages. Python syntax will also assist you in writing "clean" code. In comparison to other languages, your code will be easier to comprehend, debug, extend, and build upon. Python is used for a variety of purposes, including game development, online application development, business issue solving, and artificial intelligence/machine learning . Python is also widely utilized in scientific disciplines for academic study and practical practice [1].

One of the most essential reasons we continue to use Python is because of the Python community, which includes an incredibly diverse and welcoming group of people. Community is essential to programmers because programming isn't a solitary pursuit. Most of us, even the most experienced programmers, need to ask advice from others who have already solved similar problems. A well-connected and supportive community is critical in helping you solve problems, and the Python community fully supports people like you who are learning Python as your first programming language. One of the exciting features of python programming tools is folder management, this brilliant feature is used to achieve flexibility in using folders and directories without the need to modify them manually. This research will present the basics of dealing with file management and how to control it [1-2].

Python is a great language to learn, so let's get started!

2- INTRODUCTION

Currently, it is noticeable that the data size and its diversity are increasing, and it is difficult to deal with it, and it may sometimes reach the impossibility, especially in the case of the diversity of this information, but there is somewhat similarity in the way it behaves, so collecting and allocating it facilitates sometimes these tasks

Recently, the Python language has released tools through which it is possible to deal with data easily and flexible, in addition to its ability to deal with sub-data.

3- OBJECT-ORAINTED PROGRAMMING

Python is a programming language that is object-oriented. This means that it emphasizes the use of data structures known as objects. In contrast, procedure-oriented languages are concerned with functions. An object can be anything with a name, including functions, integers, strings, floats, classes, methods, and files. It is a collection of data and methods for using data. Objects are adaptable structures that can be used in a variety of ways. Variables, dictionaries, lists, tuples, and sets can all be assigned to them.. They are usable as arguments. A class, like a list, string, dictionary, float, or integer, is a data type [1][2]. When you create an object from the class data type, the object is referred to as a class instance. Everything in Python is an object, including classes and types. The data type 'type' includes both classes and types. The data value stored within an object is referred to as an attribute, and the functions associated with it are referred to as methods. A class is a way to create, organize, and manage objects that have similar attributes and

methods. Planning and decision-making on what the objects will represent and how to group things together are required when designing an object [1].

When you define a class with the keyword 'class,' Python creates a new class object with the same name. When you define a class, you create a namespace that contains the definitions of all the class's attributes, including special attributes that begin with double underscores. This object can then be used to access the class's attributes and to create or instantiate new class objects.

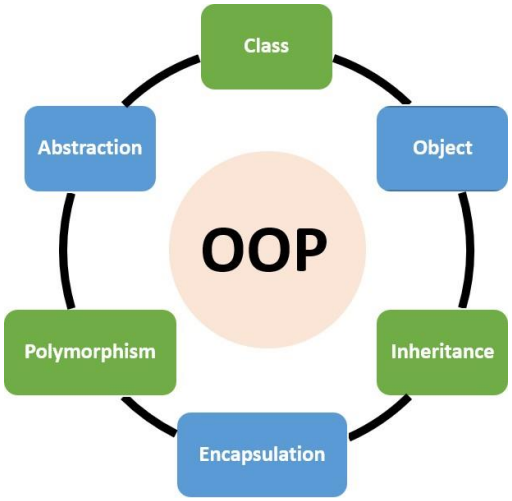


Fig.1 OOP arrangement

3-1 Inheritance

3-1-1 Introduction and Definitions

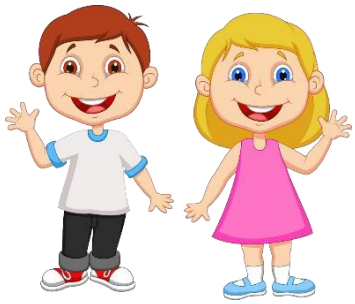
A programming structure that allows a new class to inherit the attributes of an existing class is called inheritance. The new class is referred to as the child class, subclass, or derived class, whereas the class it inherits from is referred to as the parent class, superclass, or base class. Inheritance is a useful feature because it encourages code reuse and efficiency. If you already have a class that does what your program needs, you can create a subclass that does most of what the existing class does, enhances its functionality, or partially overrides some class behavior to make it a perfect fit for your needs[1-3].

The methods or, more broadly, the software inherited by a subclass are considered to be reused in the subclass. A directed graph is formed by the inheritance of objects or classes. The class from which a class inherits is referred to as the parent or superclass. A subclass, also known as an heir class or child class, is a class that inherits from a superclass. Ancestors are another name for superclasses. Classes have a hierarchical relationship with one another. It's comparable to real-

world relationships or categorizations. Consider vehicles, for example. Vehicles include bicycles, automobiles, buses, and trucks. Pick-up trucks, vans, sports cars, convertibles, and estate cars are all cars, and as such, they are also vehicles. In Python, we could create a vehicle class with methods like accelerate and brake. Cars, buses, trucks, and bikes can all be implemented as subclasses that inherit these methods from the vehicle...

Example In this example, we note that the family has the same characteristics, so it can be placed in one class

Hello, I'm George, we're from the Lully family, and I'm short



Hello we are Lulu and John

We are as short as our father

```
class family:
    def_init_(self, first_name,
dad_name="George",last_name="Lully"):
        self.first_name = first_name
        self.dad_name = dad_name
        self.last_name=last_name
    def short(self):
        print("We are as short as our
father")

name=family("lolo")
print(name.first_name +" "+ name.dad_name
+" "+ name.last_name)
name.short()
name_1=family("jon")
print(name_1.first_name +" "+
name_1.dad_name +" "+ name_1.last_name)
name_1.short()
```

Fig.2 Basic Class

Let's apply to this example the principle of inheritance, suppose that each child has a characteristic of his own (such as eye color), but this child still has the characteristics of the family (super class or main class). In this case, we need a new class to solve this problem. This is called child class or subclass. .

- Syntax of Inheritance in Python

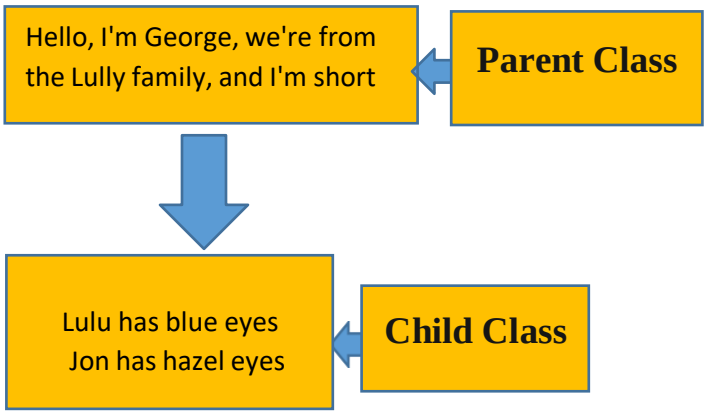
The syntax for a subclass definition looks like this:

```
class Child Class(Parent Class):
    pass
```

There will be methods and attributes instead of the pass statement, as in all other classes. BaseClassName must be defined in the scope that contains the derived class definition.

Now we'll look at a simple inheritance example using Python code.

Fig.3 Inheritance Class




```
class family:
    def_init_(self, first_name, dad_name="George",last_name="Lully"):
        self.first_name = first_name
        self.dad_name = dad_name
```

```
        self.last_name=last_name
    def short(self):
        print("We are as short as our father")
```

```
class boy(family):
    def eyes(self):
        print('I have a hazel eyes')
class girle(family):
    def eyes(self):
        print('I have a blue eyes')
```

```
# John's traits in particular
name=boy("ali")
print(name.first_name +" "+ name.dad_name +" "+ name.last_name)
name.short()
name.eyes()
#lolo's traits in particular
name=girle("lolo")
print(name.first_name +" "+ name.dad_name +" "+ name.last_name)
name.short()
name.eyes()
```

General Example Using different concepts of inheritance theory:

3-1-2 Single Inheratance:

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

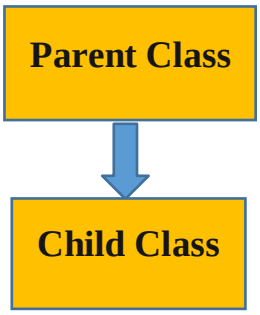


Fig.4 Single Inheritance Class

```
class family:
    def_init_(self, son_name="Ali",daughter_name="tala",
dad_name="Ahmad",mama_name="loma",
                 Grandpa_name="sami"
                ,Grandma_name="suha"):
        self.son_name=son_name
        self.daughter_name=daughter_name
        self.dad_name=dad_name
        self.mama_name=mama_name
        self.Grandpa_name=Grandpa_name
        self.Grandma_name=Grandma_name
    def display(self):
        print(" Hello We are from the Jennifer family")
new_1 = family()
new_1.display()
print(" my name is {} I am the son".format(new_1.son_name))
print(" my name is {} I'm the daughter".format(new_1.daughter_name))
print(" my name is {} I am the dad".format(new_1.dad_name))
print(" my name is {} I'm the mama".format(new_1.mama_name))
print(" my name is {} I am the Grandpa".format(new_1.Grandpa_name))
print(" my name is {} I'm the Grandma".format(new_1.Grandma_name))
```

Here we define animals in a class alone because they share in other characteristics that are different from family members but are still members of the family

```
class Animal(family):
    def name_ani(self):
        print("my name bosy i am a cat")
    def name_ani_2(self):
        print("my name is hopy i am a dog")
```

Create one object of each class and Now pass each object, one at a time, to the name_ani() and name_ani_2() function and see the

```
h=Animal()
h.name_ani()
print(h.son_name+' '+"my frind")
h.name_ani_2()
print(h.daughter_name+" "+"my frind")
h.display()
```

Let us get back to our new **Animal** class. Imagine now that an instance of a **Animal** should say hi in a different way. In this case, we have to redefine the method display inside of the subclass **Animal**, this called Overriding as shown below we define the display inside animal class

```
class Animal(family):
    def name_ani(self):
        print("my name bosy i am a cat")
    def name_ani_2(self):
        print("my name is hopy i am a dog")
    def display(self):
        print("meo meo hoho")
```

Create one object of each class and Now pass to the new display :

```
h=Animal()
h.name_ani()
print(h.son_name+' '+"my frind")
h.name_ani_2()
print(h.daughter_name+" "+"my frind")
h.display()
```

Output:

```
✓ h=Animal() ···

my name bosy i am a cat
Ali  my frind
my name is hopy i am a dog
tala my frind
meo meo hoho
```

- **Multiple Inheritance:**

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class**.**

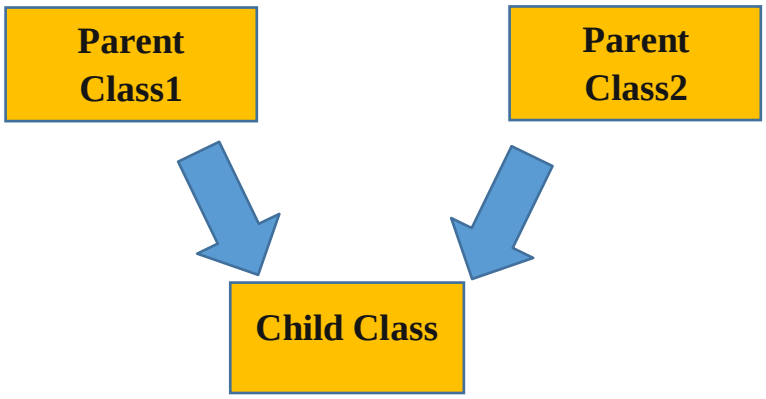


Fig.5 Multiple Inheritance Class flowchart

Here Suppose that a child from another family (a new class different from the first class) and a request to join the current class under the so-called adoption, in this case, this child will bear the characteristics of both families, both classes

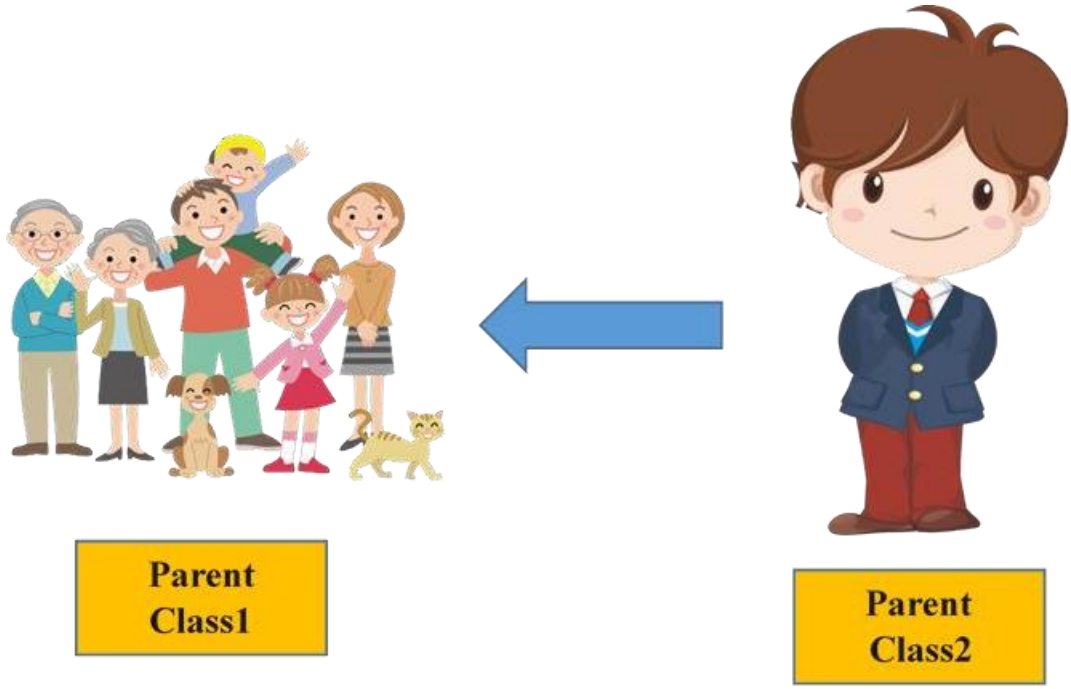


Fig.6 Multiple Inheritance Class

Here we define a two different classes and we combine them in one classes derived(class1, class2)

```
class family_2:
    def adopted (self):
        print(" hello my name is loi i am from roti family ")

class derived(family, family_2):
    pass
```

```
multi_class=derived()
multi_class.adopted()
print ("my new dad is "+" "+multi_class.dad_name)
print("my new is mama"+" "+multi_class.mama_name)
```

Output:

```
 hello my name is loi i am from roti family (old family)
my new dad is  Ahmad
my new  mama is loma
```

- **Multilevel Inheritance :**

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather**.**

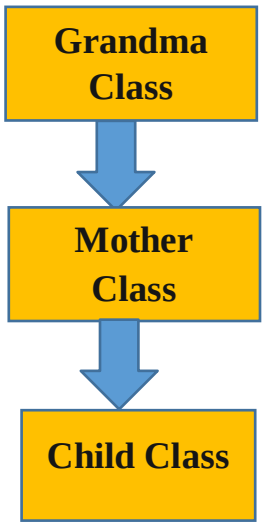


Fig.7 Multilevel Inheritance Class flow chart

Here the family names will be called successively without the need to identify every name in the family

```
class Grandpa (object):#The base class
    def __init__(self,Grandpa_name):
        self.Grandpa_name=Grandpa_name
class dad(Grandpa): # Middle class
    def _init_(self,dad_name, Grandpa_name):
        self.dad_name=dad_name
        Grandpa._init_(self,Grandpa_name)
class son (dad):# last class
    def _init_(self,son_name,dad_name, Grandpa_name):
        self.son_name=son_name
        dad._init_(self, dad_name,Grandpa_name)
    def son_dad(self):
        return " I have a sister called tala she has a class also"
class dut (dad):# last class
    def _init_(self,dut_name,dad_name, Grandpa_name):
        self.dut_name=dut_name
        dad.__init__(self, dad_name,Grandpa_name)
```

```
family=dad("Ahmad","Sami")
print(family.dad_name,family.Grandpa_name)
family_G=son("Ali","Ahamd","Sami")
print(family_G.son_name,family_G.dad_name,family_G.Grandpa_name)
print(family_G.son_dad())
family_b=dut("tala","Ahamd","Sami")
print(family_b.dut_name,family_b.dad_name,family_b.Grandpa_name)
```

Output:

```
Ahmad Sami
Ali Ahamd Sami
 I have a sister called tala she has a class also
tala Ahamd Sami
```

General Example:

```
 Inherited or Sub class (Note Person in bracket)
class Child(Base):
    # Constructor
    def _init_(self, name, age):
        Base._init_(self, name)
        self.age = age
    # To get name
    def getAge(self):
        return self.age
# Inherited or Sub class (Note Person in bracket)
class GrandChild:
    # Constructor
    def _init_(self, name, age, address):
        Child._init_(self, name, age)
        self.address = address
    # To get address
    def getAddress(self):
        return self.address
# Driver code
g = GrandChild("Geek1", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())
```

4-1 POLYMORPHISM

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Polymorphism without inheritance in the form of duck typing as available in Python due to its dynamic typing system. This means that as long as the classes contain the same methods the Python interpreter does not distinguish between them, as the only checking of the calls occurs at run-time

Example:

Imagine that you entered a farm to spot ducks that were shouting **Quaaaaaack** and a white and gray feather, but by chance you found a man wearing the same robe as the duck and shouting **Quaaaak** the solution??

```
class Duck:
    def quack(self):
        print("Quaaaaaack!") #Duck
    def feathers(self):
        print("The duck has white and gray feathers.")
class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")
def in_the_forest(obj):
    obj.quack()
    obj.feathers()
donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

Output:

```
Quaaaaaack!
The duck has white and gray feathers.
The person imitates a duck.
The person takes a feather from the ground and shows it.
```

- Basic Polymorphism

Polymorphism is the ability to perform an action on an object regardless of its type. This is generally implemented by creating a base class and having two or more subclasses that all implement methods with the same signature. Any other function or method that manipulates these objects can call the same methods regardless of which type of object it is operating on, without needing to do a type check first. In object-oriented terminology when class X extend class Y , then Y is called super class or base class and X is called subclass or derived class.

Example:

This is a parent class that is intended to be inherited by other classes, This method is intended to be overridden in subclasses. If a subclass doesn't implement it but it is called, **NotImplemented** will be raised.

```
class Shape:
    def calculate_area(self):
        return NotImplemented
class Square(Shape):
    side_length = 2 # in this example, the sides are 2 units long
    def calculate_area(self):
This method overrides Shape.calculate_area(). When an object of type
Square has its calculate_area() method called, this is the method that
will be called, rather than the parent class' version.
It performs the calculation necessary for this shape, a square, and
returns the result.
        return self.side_length * 2
class Triangle(Shape):
This is also a subclass of the Shape class, and it represents a triangle
    base_length = 4
    height = 3
    def calculate_area(self):
This method also overrides Shape.calculate_area() and performs the area
calculation for a triangle, returning the result.
        return 0.5 * self.base_length * self.height
def get_area(input_obj):
This function accepts an input object, and will call that object's
calculate_area() method. Note that the object type is not specified. It
could be a Square, Triangle, or Shape object.
"""
    print(input_obj.calculate_area())
# Create one object of each class
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()
# Now pass each object, one at a time, to the get_area() function and see the
```

```
# result.
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)
```

Without Polymorphism the the last function calculate_area() will be executed

:

```
class Shape:
    def calculate_area(self):
       return NotImplemented
class Square(Shape):
    side_length = 2 # in this example, the sides are 2 units long
    def calculate_area(self):
        return self.side_length * 2
class Triangle(Shape):
    base_length = 4
    height = 3
    def calculate_area(self):
        return 0.5 * self.base_length * self.height

# Create one object of each class
f=Shape()
f.calculate_area()
namew=Square()
namew.calculate_area()
l=Triangle()
l.calculate_area()
```

Output:

6.0

4- REFERANCES

[1] Matthes, E. (2019). *Python crash course: A hands-on, project-based introduction to programming*. No Starch Press.

[2] Kuhlman, D. (2009). *A python book: Beginning python, advanced python, and python exercises* (pp. 1-227). Lutz: Dave Kuhlman.

[3] *Python Directory and Files Management*. Programiz. (n.d.). Retrieved October 22, 2022,from https://www.programiz.com/python-programming/directory

[4] *Python directory*. Python Tutorial - Master Python Programming For Beginners from Scratch. (2022, September 15). Retrieved October 22, 2022, from https://www.pythontutorial.net/python-basics/python-directory/

[5] YouTube. (2022, June 5). *File organizing with python: Rename, Move, copy & delete files and Folders*. YouTube. Retrieved October 22, 2022, from https://www.youtube.com/watch?v=NOvFZamGXXo