Python Polymorphism Concept:

- The word polymorphism means having many forms.
- Yes, Python support polymorphism.
- If one entity shows more than one behavior then it is called as Polymorphism.
- Polymorphism is an important feature of class definition in Python that is utilised when you have commonly named methods across classes or sub classes.
- Polymorphism can be carried out through inheritance, with sub classes making use of base class methods or overriding them.

Polymorphism is classified into two types.

1. Method Overloading
2. Method Overriding

Overloading:

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Synatx:

```
class Calculation:
    def sum(a,b):
        pass
    def sum(a,b,c,d):
        pass
```

Example 1:

```
class  A:
   def  m1(self,a):
            print(a)

   def m1(self,a,b):
      print(a + b)

   def  m1(self,  name , age , a):
```

```
        print('Name is :' , name)
        print('Age is :' , age)
        print(a)

obj = A()
obj.m1('Avinesh',30,50)
```

Output:

```
Name is : Avinesh
Age is : 30
50
```

Example 3: Polymorphism in Class Methods

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age}
years old.")

    def make_sound(self):
        print("Meow")


class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age}
years old.")

    def make_sound(self):
        print("Bark")
```

```
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.info()
    animal.make_sound()
```

Output

```
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Here, we have created two classes `Cat` and `Dog`. They share a similar structure and have the same method names `info()` and `make_sound()`.

However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through it using a common `animal` variable. It is possible due to polymorphism.

Python Overriding Concept :

Overriding means having two methods with the same method name and same parameters (i.e., method signature). One of the method is in the Parent class and the other method is in the Child class.

Syntax:

```
class Person:
   def read(self):
       pass

class Employee(Person):
   def read(self):
       pass
```

Example 1:

```
class Person:
    def read(self):
        print('Person class read() method')

class Employee(Person):
    def read(self):
        super().read()
        print('Employee class read() method')

e = Employee()
e.read()
```

Output:

Person class read() method
Employee class read() method

Operator Overloading Concept:

- Operator overloading refers to the ability to define an operator to work in a different manner depending upon the type of operand it is used with.
- The **operator +** is performing defferent behaviours based on operands values like strings and numeric values.

Example 1: Polymorphism in addition operator

We know that the `+` operator is used extensively in Python programs. But, it does not have a single usage.

For integer data types, `+` operator is used to perform arithmetic addition operation.

```
num1 = 1
num2 = 2
print(num1+num2)
```

Hence, the above program outputs `3`.

Similarly, for string data types, `+` operator is used to perform concatenation.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs `Python Programming`.

Here, we can see that a single operator `+` has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

Function Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types.

One such function is the `len()` function. It can run with many data types in Python. Let's look at some example use cases of the function.

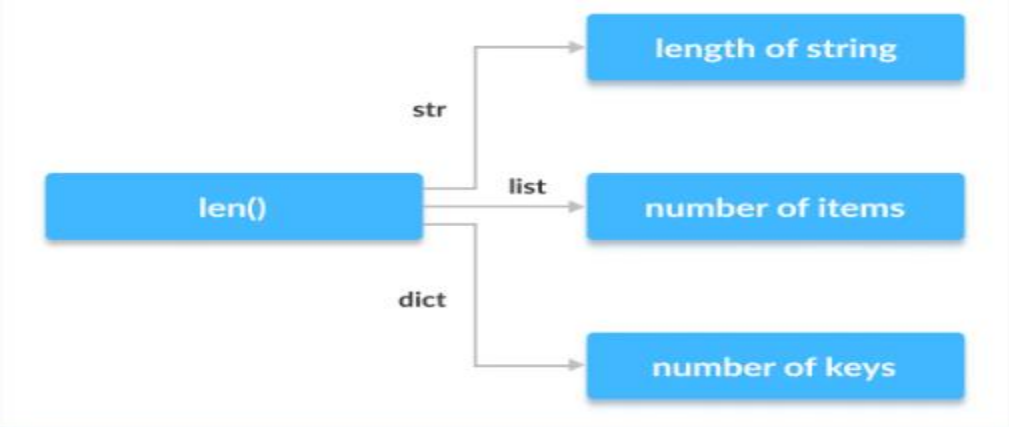
Example 2: Polymorphic len() function

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Output:

```
9
3
2
```

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the `len()` function. However, we can see that it returns specific information about specific data types.



Polymorphism in len() function in Python

Example:

```
# function overloading
def f1(a,b):
    print(a*b)
def f1(a,b,c):
    print(a+b-c)
f1(10,20,5)
```

Output:

```
The result is: 25
```

Example:

```
def add(datatype,*args):
    if datatype=='int':
        result = 0
    if datatype=='str':
        result = ' '

    for x in args:
        result = result + x
    print(result)

add('int',5,6)
add('str', 'Srinivas', 'hello')
add('int',20,30,40,50)
```

Output:

```
11
Srinivashello
140
```

"dispatch" Decorator in Python:

A Dispatch decorator is used to select between different implementations of the same abstract method based on the signature, or list of types.

If you want to use dispatch() decorator then we need to install "multipledispatch" third party module using pip command.

cmd> pip install multipledispatch

Note: After installing "multipledispatch" module then we need to import "dispatch" decorator to executing perticular method with different parameters.

Example:

```
from multipledispatch import dispatch
@dispatch(int,int)
def product(a,b):
    result = a * b
    print(result)

@dispatch(int,int,int)
def product(a,b,c):
    result = a * b * c
    print(result)

@dispatch(str,str)
def product(a,b):
    restlt = a + '  '+ b
    print(restlt)

product(10,20)
product(10,20,10)
product('Hello','Rumi')
```

How to send class object as a input to the function calling?

Overriding:

Example:

```
class Tomato:
    def type(self):
        print('vegetable')
```

```
    def color(self):
        print('It looks like Red')

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

    def color(self):
        print('It looks like Red and green')

def func(obj):
    obj.type()
    obj.color()

t = Tomato()
a = Apple()
func(t)
func(a)
```

 # Python program to demonstrate dispatch decorator

```
from multipledispatch import dispatch
@dispatch(int)
def func(x):
    return x * 2

@dispatch(float)
def func(x):
    return x / 2

# Driver code
print(func(2))
print(func(2.0))
```

Output:

4
1.0

Practice Examples:

Example 1:

```
class Person:
   c = 40
   def result(self):
      self.a = 10
      self.b = 20
      print('Person class result() method')
      print("The result is :", self.a + self.b )
      print("The result is :", p.a + p.b )
      # print("The result is :", Person.a + Person.b ) # instance variable

class Employee(Person):
   def result(self):
      print('Employee class result() 1st time')
      # super().result()
      print("The result is :",p.a * p.b )
      # print("The result is :",self.a * self.b )
      # print("The result is :",Person.a * Person.b ) # instance variables
      print('Employee class result() method 2nd time')

p = Person()
p.result()
e = Employee()
e.result()
print()
print(p.a) # instance variables
print(p.b) # instance variables
#print(Person.a) --->> error
#print(Person.b) --->> error
```

```
# class variables
print(Person.c)
print(p.c)
```

Output:

```
Person class result() method
The result is : 30
The result is : 30
Employee class result() 1st time
The result is : 200
Employee class result() method 2nd time

10
20
40
40
```

Example 2:

```
# Identifying the Object and display value.

from multipledispatch import dispatch
@dispatch(object)
def add(x):
    if type(x)==list:
        print("List Object is :",x)

    elif type(x)==tuple:
        print("Tuple Object is :",x)

    elif type(x)==set:
        print("Set Object is :",x)
    else:
        print(x)
add((10,20,30))
```

Output:

```
Tuple Object is : (10, 20, 30)
```