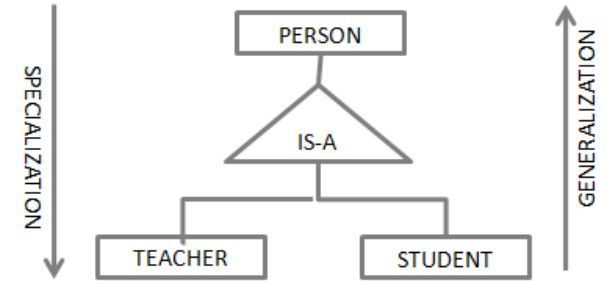
Generalization

The process of extracting common characteristics from two or more classes and combining them into a generalized super class is called Generalization. The common characteristics can be attributes or methods.

Specialization

Specialization is the reverse process of Generalization means creating new subclasses from an existing class.

Generalization and Specialization are represented by “is-a” relationship.



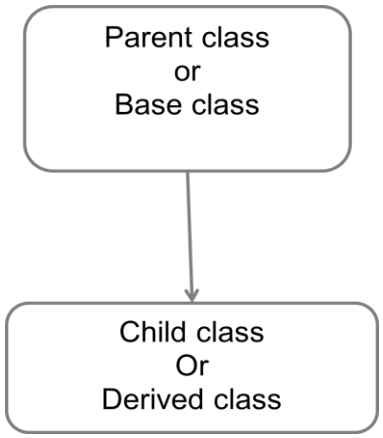
TEACHER and STUDENT class extends the PERSON class
(This is called Inheritance)

PERSON is called base class and TEACHER and STUDENT are called derived class.

Inheritance

The method of inheriting the properties of parent class into a child class is known as inheritance.

It is an OOP concept.



Advantages of Inheritance

1. Code reusability- we do not have to write the same code again and again, we can just inherit the properties from base class to child class.
2. It represents a real world relationship between parent class and child class.
3. It is transitive in nature. If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.

Example:

```
class Parent():
       def fun1(self):
            print('Base Class')

class Child(Parent):
       def fun2(self):
            print('Child Class')

ob = Child()
ob.fun1()
ob.fun2()
```

```
Base Class
Child Class
```

Sub-classing

Calling a constructor of the parent class by mentioning the parent class name in the declaration of the child class is known as sub-classing. A child class identifies its parent class by sub-classing.

__init__( ) Function

The __init__() function is called every time a class is being used to make an object. When we add the __init__() function in child class, the child class will no longer be able to inherit the parent class's __init__() function. The child's class __init__() function overrides the parent class's __init__() function.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show(self):
        print("Name : ",self.name)
        print("Age : ", self.age)

class Teacher(Person):
    def __init__(self, name, age, dept, emp_id):
        Person.__init__(self, name, age)
        self.dept = dept
        self.emp_id = emp_id
    def show(self):
        Person.show(self)
        print(" Department = ",self.dept)
        print("Employee id = ",self.emp_id)
```

super ( ) function

In an inherited subclass, a parent class can be referred to with the use of the super() function. The super function returns a temporary object of the base class that allows access to all of its methods to its child class.

Advantages of super () function

- Need not remember or specify the parent class name to access its methods. This function can be used both in single and multiple inheritances.
- This implements modularity (isolating changes) and code reusability as there is no need to rewrite the entire function.
- Super function in Python is called dynamically because Python is a dynamic language unlike other languages.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show(self):
        print("Name : ",self.name)
        print("Age : ", self.age)

class Teacher(Person):
    def __init__(self, name, age, dept, emp_id):
        super().__init__(name,age)
        self.dept = dept
        self.emp_id = emp_id
    def show(self):
        super().show()
        print(" Department = ",self.dept)
        print("Employee id = ",self.emp_id)
```

Types of inheritance

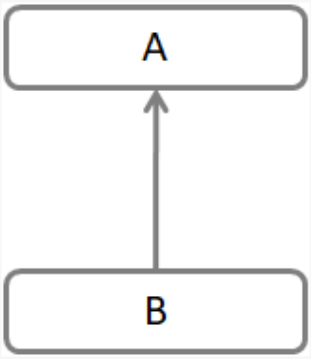
1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

```python
#Single Inheritance
class A:
    def fun1(self):
        print('fun of class A')

class B(A):
    def fun2(self):
        print('fun of class B')

obj = B()
obj.fun1()
obj.fun2()
```




```
fun of class A
fun of class B
```

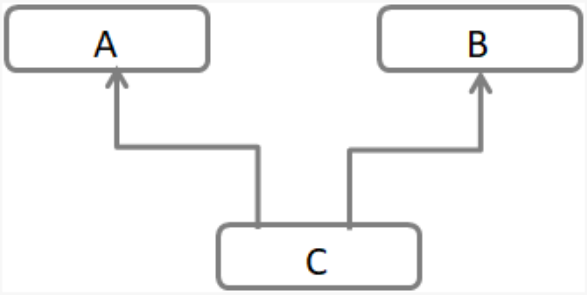
2. Multiple Inheritance

```python
#Multiple Inheritance
class A:
    def fun1(self):
        print('fun of class A')

class B:
    def fun2(self):
        print('fun of class B')

class C(A,B):
    def fun3(self):
        print('fun of class C')

obj = C()
obj.fun1()
obj.fun2()
obj.fun3()
```




```
fun of class A
fun of class B
fun of class C
```

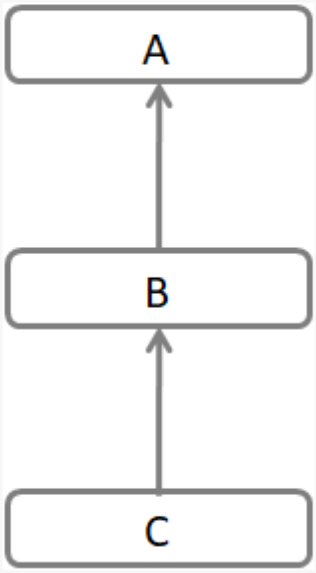
3. Multilevel Inheritance

```python
#Multilevel Inheritance
class A:
    def fun1(self):
        print('fun of class A')

class B(A):
    def fun2(self):
        print('fun of class B')

class C(B):
    def fun3(self):
        print('fun of class C')

obj = C()
obj.fun1()
obj.fun2()
obj.fun3()
```

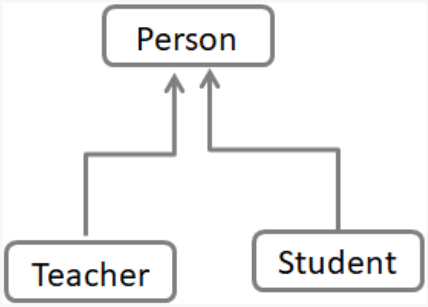



```
fun of class A
fun of class B
fun of class C
```

4. Hierarchical Inheritance

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show(self):
        print("Name : ",self.name)
        print("Age : ", self.age)

class Teacher(Person):
    def __init__(self, name, age, dept, emp_id):
        Person.__init__(self, name, age)
        self.dept = dept
        self.emp_id = emp_id
    def show(self):
        Person.show(self)
        print("Department = ",self.dept)
        print("Employee id = ",self.emp_id)
```




```
class Student(Person):
    def __init__(self, name, age, branch, rollno):
        Person.__init__(self,name,age)
        self.branch = branch
        self.rollno = rollno

    def show(self):
        Person.show(self)
        print("Branch: ",self.branch)
        print("Roll Number: ",self.rollno)

t1 = Teacher("Abhishek", 34, "Data Science", "M2015558")
t1.show()
```

```
Name :  Abhishek
Age :  34
Department =  Data Science
Employee id =  M2015558
```

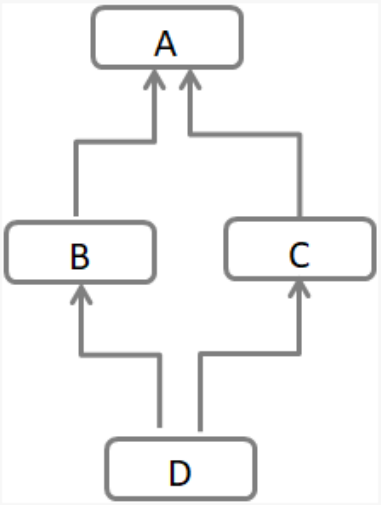
5. Hybrid Inheritance

```python
#Hybrid Inheritance
class A:
    def fun1(self):
        print("fun1 of class A")

class B(A):
    def fun2(self):
        print("fun2 of class B")

class C(A):
    def fun3(self):
        print("fun3 of class C")

class D(B,C):
    def fun4(self):
        print("fun4 of class D")
obj = D()
obj.fun1()
obj.fun2()
obj.fun3()
obj.fun4()
```



Method overriding

Implementing a **method** in child class which is already present in parent class is known as **method overriding**. **Overriding** is done so that a child class can give its own implementation to a **method** which is already provided by the parent class.

To override a method in the base class, sub class needs to define a method of same signature. (i.e same method name and same number of parameters as method in base class).

EXAMPLE:

In the code below, the method fun( ) of base class A is being overridden by derived class B.

```
# method overriding
class A:
    def fun(self):
        print("fun of class A")

class B(A):
    def fun(self):
        print("fun of class B")

obj = B()
obj.fun()
```

```
fun of class B
```

Invoking the parent class method

Parent class methods can also be called within the overridden methods. This can generally be achieved by two ways.

i. **Using Classname:** Parent's class methods can be called by using the *Parent classname.method* inside the overridden method.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show(self):
        print("Name : ",self.name)
        print("Age : ", self.age)

class Teacher(Person):
    def __init__(self, name, age, dept, emp_id):
        Person.__init__(self,name, age)
        self.dept = dept
        self.emp_id = emp_id
    def show(self):
        Person.show(self)
        print("Department = ",self.dept)
        print("Employee id = ",self.emp_id)

t = Teacher("Amit",55,"CSE","S7676")
t.show()
```

```
Name :  Amit
Age :  55
Department =  CSE
Employee id =  S7676
```

ii. **Using super( ):** Python super() function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call super-class functions. It returns the proxy object that allows us to refer parent class by 'super'.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show(self):
        print("Name : ",self.name)
        print("Age : ", self.age)

class Teacher(Person):
    def __init__(self, name, age, dept, emp_id):
        super().__init__(name, age)
        self.dept = dept
        self.emp_id = emp_id
    def show(self):
        super().show()
        print("Department = ",self.dept)
        print("Employee id = ",self.emp_id)

t = Teacher("Amit",55,"CSE","S7676")
t.show()
```

```
Name :  Amit
Age :  55
Department =  CSE
Employee id =  S7676
```

Abstract class

A class is called an Abstract class if it contains one or more abstract methods.

An **abstract method** is a method that is declared, but contains no implementation. Abstract classes cannot be instantiated, and its abstract methods must be implemented by its subclasses.

If a method is implemented in Abstract class itself then it is called a **Concrete method**.

How Abstract Base classes work:

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is **abc**. ***ABC*** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword **@abstractmethod**.

To create an abstract class we need to import:

```
from abc import ABC, abstractmethod
```

Here, abc is module, ABC class is known as Metaclass (a class that defines the behavior of another class), abstractmethod is a function.

We can also say that a class that is derived from Metaclass ABC is called an Abstract class.

Example: Abstract Method and Concrete Method

```
from abc import ABC, abstractmethod

class Shapes(ABC):

    @abstractmethod
    def fun(self):
        pass
    def show(self):                    #Concrete Method
        print("I am Concrete Method")
```

Here, fun() is abstract method and show() concrete method.

Example: Program to illustrate abstract class and abstract method.

```python
# Program to demonstrate abstract class
from abc import ABC, abstractmethod

class Shapes(ABC):

    @abstractmethod
    def fun(self):
        pass

class Triangle(Shapes):
    def fun(self):
        print("I am Triangle")

class Rectangle(Shapes):
    def fun(self):
        print("I am Rectangle")

class Hexagon(Shapes):
    def fun(self):
        print("I am Hexagon")

a = Triangle()
a.fun()

b = Rectangle()
b.fun()

c = Hexagon()
c.fun()
```

```
I am Triangle
I am Rectangle
I am Hexagon
```

MRO (Method Resolution Order)

MRO is a concept used in inheritance.

It is the order in which Python looks for a method in a hierarchy of classes.

It is especially useful in Python because Python supports multiple inheritance.

In Python, the MRO is from bottom to top and left to right. To understand the concept of MRO and its need, let's discuss few cases.

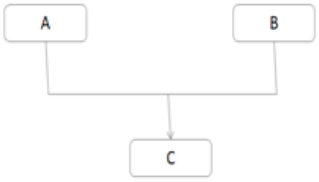
Case 1:

```
class A:
    def fun(self):
        print('fun of class A')

class B:
    pass

class C(A, B):
    pass


obj = C()
obj.fun()
print(C.mro())   # print MRO for class C
```




```
fun of class A
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

MRO is **C → A → B → object**

From MRO of class C, we get to know that Python looks for a method first in class C. Then it goes to A and then to B. So, first it goes to super class given first in the list then second super class, from left to right order. Then finally object class, which is a super class for all classes.

Case 2:

```
class A:
    def fun(self):
        print('fun of class A')

class B:
    def fun(self):
        print('fun of class B')

class C(A, B):
    pass


obj = C()
obj.fun()
print(C.mro())   # print MRO for class C
```

```
fun of class A
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

MRO is **C → A → B → object**

Python calls fun() method in class A. According to MRO, it searches A first and then B. So if method is found in A then it calls that method.

However, if we remove fun() method from class A then fun() method in class B will be called as it is the next class to be searched according to MRO.

The ambiguity that arises from multiple inheritance is handled by Python using MRO.

Case 3:

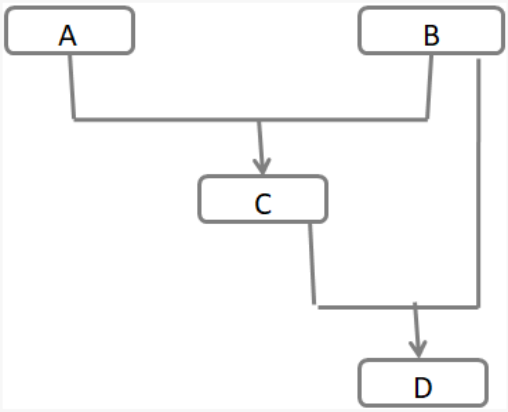
```
class A:
    def fun(self):
        print('fun of class A')

class B:
    def fun(self):
        print('fun of class B')

class C(A, B):
    def fun(self):
        print('fun of class c')

class D(C, B):
    pass

obj = D()
obj.fun()
print(D.mro())   # print MRO for class D
```



fun of class c
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]

MRO is **D → C → A → B →object**

Case 4:

Now, we create B and C from A and then D from B and C. Method fun() is present in both A and C.

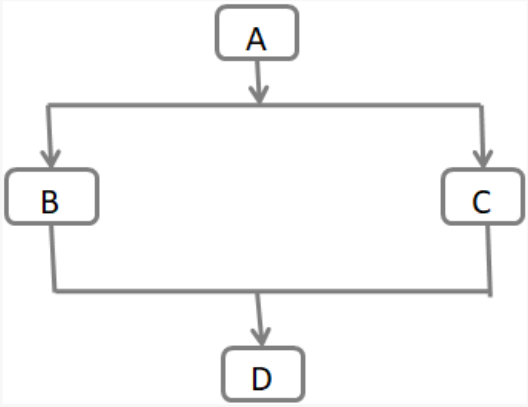
```
class A:
    def fun(self):
        print('fun of class A')

class B(A):
    pass

class C(A):
    def fun(self):
        print('fun of class C')

class D(B,C):
    pass

obj = D()
obj.fun()
print(D.mro())   # print MRO for class D
```



fun of class C
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]

When we call fun() with an object of class D, it should start with first Super class – B (and its super classes) and then second super class – C (and its super classes). If that is the case then we will call fun() method from class A as B doesn't have it and A is super class for B.

Here, MRO should be **D →B → A →object → C → A → object**.

However, that is contradictory to rule of inheritance, as most specific version must be taken first and then least specific (generic) version. So, calling fun() from A, which is super class of C, is not correct as C is a direct super class of D. That means C is more specific than A. So method must come from C and not from A.

This is where Python applies a simple rule that says : **when in MRO we have a super class before subclass then it must be removed from that position in MRO.**

But as A is super class of C, it cannot be before C in MRO. Similarly object is super class of A so it cannot be before A. So, Python removes A from that position, which results in new MRO as follows:

D →B → C → A → object .

Case 5:

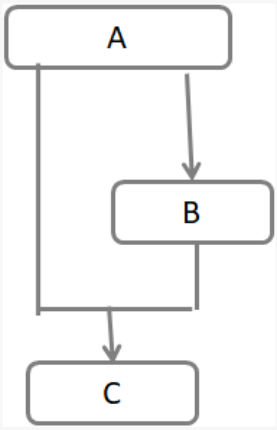
There are cases when Python cannot construct MRO due to complexity of hierarchy. In such cases it will throw an error as demonstrated by the following code.

```
class A:
    def fun(self):
        print('fun of class A')

class B(A):
    def fun(self):
        print('fun of class B')

class C(A, B):
    pass


obj = C()
obj.fun()
print(C.mro())   # print MRO for class D
```



***TypeError**: Cannot create a consistent method resolution order (MRO) for bases A, B*

The problem comes from the fact that class A is a super class for both C and B. If you construct MRO then it should be like this:

$$C \rightarrow A \rightarrow B \rightarrow A$$

Then according to the rule A should NOT be ahead of B as A is super class of B. So new MRO must be like this

$$C \rightarrow B \rightarrow A$$

But A is also direct super class of C. So, if a method is in both A and B classes then which version should class C call? According to new MRO, the version in B is called first ahead of A and that is not according to inheritance rules (specific to generic) resulting in Python to throw error.

Polymorphism

Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types. This makes programming more intuitive and easier.

In python we can find the same operator or function taking multiple forms.

We have different ways to define polymorphism.

Polymorphism in operators

The + operator can take two inputs and give us the result depending on the datatypes of the inputs.

```
a = 20
b = 30
c = 20.45
s1 = "Greater "
s2 = "Noida"
print(a + b)
print(type(a + b))
print(b + c)
print(type (b + c))
print(s1 + s2)
print(type(s1 + s2))
```

```
50
<class 'int'>
50.45
<class 'float'>
Greater Noida
<class 'str'>
```

Polymorphism in built-in functions

When we supply a string value to **len( )** it counts every letter in it. But if we supply tuple or a dictionary as an input, it processes them differently.

```
str = 'NIET, Greater Noida'
lst = ['Mon','Tue','wed','Thu','Fri']
tup = ('Red','Green','Blue','Yellow')
dict = {'1D':'Line','2D':'Rectangle','3D':'Cuboid'}
print(len(str))
print(len(lst))
print(len(tup))
print(len(dict))
```

```
19
5
4
3
```

Polymorphism with Function and Objects

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

```
from math import pi

class Rectangle:
    def __init__(self, length, breadth):
        self.l = length
        self.b = breadth
    def perimeter(self):
        return 2*(self.l + self.b)
    def area(self):
        return self.l * self.b

class Circle:
    def __init__(self, radius):
        self.r = radius
    def perimeter(self):
        return 2 * pi * self.r
    def area(self):
        return pi * self.r ** 2

rec = Rectangle(10,7)
cr = Circle(5)
print("Perimter of rectangel: ",rec.perimeter())
print("Area of rectangel: ",rec.area())

print("Perimter of Circle: ",cr.perimeter())
print("Area of Circle: ",cr.area())
```

```
Perimter of rectangel:  34
Area of rectangel:  70
Perimter of Circle:  31.41592653589793
Area of Circle:  78.53981633974483
```

Polymorphism with Inheritance (Method Overriding)

We can define functions in the derived class that has the same name as the functions in the base class. Here, we re-implement the functions in the derived class. The phenomenon of re-implementing a function in the derived class is known as Method Overriding.

```
# method overriding
class A:
    def fun(self):
        print("fun of class A")

class B(A):
    def fun(self):
        print("fun of class B")

obj = B()
obj.fun()
```

```
fun of class B
```

Compile-Time Polymorphism or Method Overloading (NOT Supported)

Unlike many other popular object-oriented programming languages such as Java, Python doesn't support compile-time polymorphism or method overloading. If a class or Python script has multiple methods with the same name, the method defined in the last will override the earlier one.

Python doesn't use function arguments for method signature, that's why method overloading is not supported in Python.

Operator Overloading

To perform operator overloading, Python provides some special function or *magic function* that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

Example 1:

```
# Program to overload an binary + operator

class A:
    def __init__(self, a):
        self.a = a

    def __add__(self, o):
        return self.a + o.a

ob1 = A(10)
ob2 = A(15)
ob3 = A("Greater ")
ob4 = A("Noida")

print(ob1 + ob2)
print(ob3 + ob4)
```

```
25
Greater Noida
```

Example 2: Write a program to overload + operator so that it can add two fractions.

```
# overloading + operator for addition of two fractions
def gcd(x,y):
        if y == 0:
            return x
        else:
            return gcd(y,x%y)

class fraction:
    def __init__(self,nr,dr=1):
        self.nr = nr
        self.dr = dr

    def __add__(self,obj):
        nr = self.nr*obj.dr + obj.nr*self.dr
        dr = self.dr*obj.dr
        return fraction(nr,dr)

    def display(self):
        x = gcd(self.nr, self.dr)
        self.nr //= x
        self.dr //= x
        print(f"Result = {self.nr}/{self.dr}")



a = fraction(5,10)
b = fraction(4,20)
c = a + b
c.display()
```

```
Result = 7/10
```

Introspection and Reflection

- Introspection is the ability to find out information about an object at runtime.

  Since, everything in Python is an object so it is also called object introspection or code Introspection.

  Code **Introspection** is used for examining the classes, methods, objects, modules, keywords to get information about them.

Python, being a dynamic, object-oriented programming language, provides tremendous introspection support.

- Reflection is the ability to modify the object at run time.
  Python supports both introspection and reflection.

Introspection

Built-in functions that are used for code introspection

Python provides some built-in functions that are used for code introspection. These are following:

type(obj)	Returns the type of an object
dir(obj)	Return list of methods and attributes associated with that object
str(obj)	Converts everything into a string
id(obj)	Returns a unique id of an object
isinstance(obj,type)	Returns True if a certain object is an instance of the specified class
hasattr(obj,'a')	This returns True if obj has an attribute a
callable(obj)	Returns True if obj is callable
vars(obj)	Returns all the instance variables of an object in a dictionary
issubclass(B,A)	Returns True if B is a derived class of class A
globals()	returns you a dictionary of variables declared in the global scope
locals()	returns you a dictionary of variables declared in the local scope

Example:

```python
def funct():
    print("This is a function")

lst = [2,3,4,6]
x = 5
class MyClass:
    def __init__(self):
        self.a = 10
        self.b = 20
    def fun(self):
        print("fun of my class")

obj = MyClass()

print("type: ", type(obj))
print("dir: ", dir(obj))
str(lst)
print("id: ", id(obj))
print("isinstance: ", isinstance(x,int))
print("hasattr: ", hasattr(obj,'a'))
print("callable: ",callable(funct))
print("vars ", vars(obj))
```

```
type:  <class '__main__.MyClass'>
dir:  ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'a', 'b', 'fun']
id:  2605543426568
isinstance:  True
hasattr:  True
callable:  True
vars  {'a': 10, 'b': 20}
```

inspect module

The inspect module provides functions for introspecting on live objects, including modules, classes, instances, functions, and methods and their source code.

Most commonly used methods of inspect module are:

isclass().	Returns True if that object is a class or false otherwise
ismodule()	Returns True if the given argument is an imported module
isfunction()	Returns True if the given argument is a function name
ismethod()	Returns True if the argument passed is the name of a method
isbuiltin()	Return True if the object is a built-in function or a bound built-in method
isabstract()	Return True if the object is an abstract base class
getclasstree()	The getclasstree() method will help in getting and inspecting class hierarchy. It returns a tuple of that class and that of its preceding base classes.
getmembers()	Return all the members of an object in a list of (name, value) pairs sorted by name.
getsource()	Returns the source code of a module, class, method, or a function
getmodule(obj)	Returns the module name of the obj
getdoc()	Get the documentation string for an object
signature()	Returns arguments to be passed in a callable

Example: getclasstree( )

```
import inspect
import pprint
class A(object):
    pass

class B(A):
    pass

class C(B):
    pass

class D(C, A):
    pass

pprint.pprint(inspect.getclasstree([A, B, C, D]))
```

```
[(<class 'object'>, ()),
 [(<class '__main__.A'>, (<class 'object'>,)),
  [(<class '__main__.B'>, (<class '__main__.A'>,)),
   [(<class '__main__.C'>, (<class '__main__.B'>,)),
    [(<class '__main__.D'>, (<class '__main__.C'>, <class '__main__.A'>))]],
   (<class '__main__.D'>, (<class '__main__.C'>, <class '__main__.A'>))]]]
```

Example: getmembers( )

```
import inspect
class MyClass:
    def fun1():
        pass
    def fun2():
        pass

c = MyClass()

for name, value in inspect.getmembers(c):
    if name.startswith('__')==False:
        print(name)
```

```
fun1
fun2
```

Example: Get a list of classes within the current module

```
import sys
print(inspect.getmembers(sys.modules[__name__], inspect.isclass))
```

Example: getsource( )

```
import inspect

def funct(a,b):
    z = a+b
    return z

print(inspect.getsource(funct))
```

```
def funct(a,b):
    z = a+b
    return z
```

Example: getmodule( )

```
from math import sqrt
print(inspect.getmodule(sqrt))
```

```
<module 'math' (built-in)>
```

Example: getdoc( )

```
import inspect
def fun(x, y):
    """This function adds two numbers"""
    return x + y

print(inspect.getdoc(fun))
```

```
This function adds two numbers
```

Example: signature( )

```
import inspect
def fun(x, y):
    """This function adds two numbers"""
    return x + y

print(inspect.signature(fun))
```

```
(x, y)
```