Python Programming

(ESTD-1995)

RGM College of Engineering & Technology (Autonomous)

Department of Computer Science & Engineering

AY:2021-2022

PYTHON'S OBJECT ORIENTED PROGRAMMING - 11

Guido Van Rossum

Agenda:

I. Encapsulation

1. Abstract Method
2. Abstract Class
3. Interface
4. Public, Private and Protected members

Encapsulation

As part of the Encapsulation module, we have to discuss about the following concepts:

1. Abstract Method
2. Abstract class
3. Interface
4. Public, Private and Protected Members

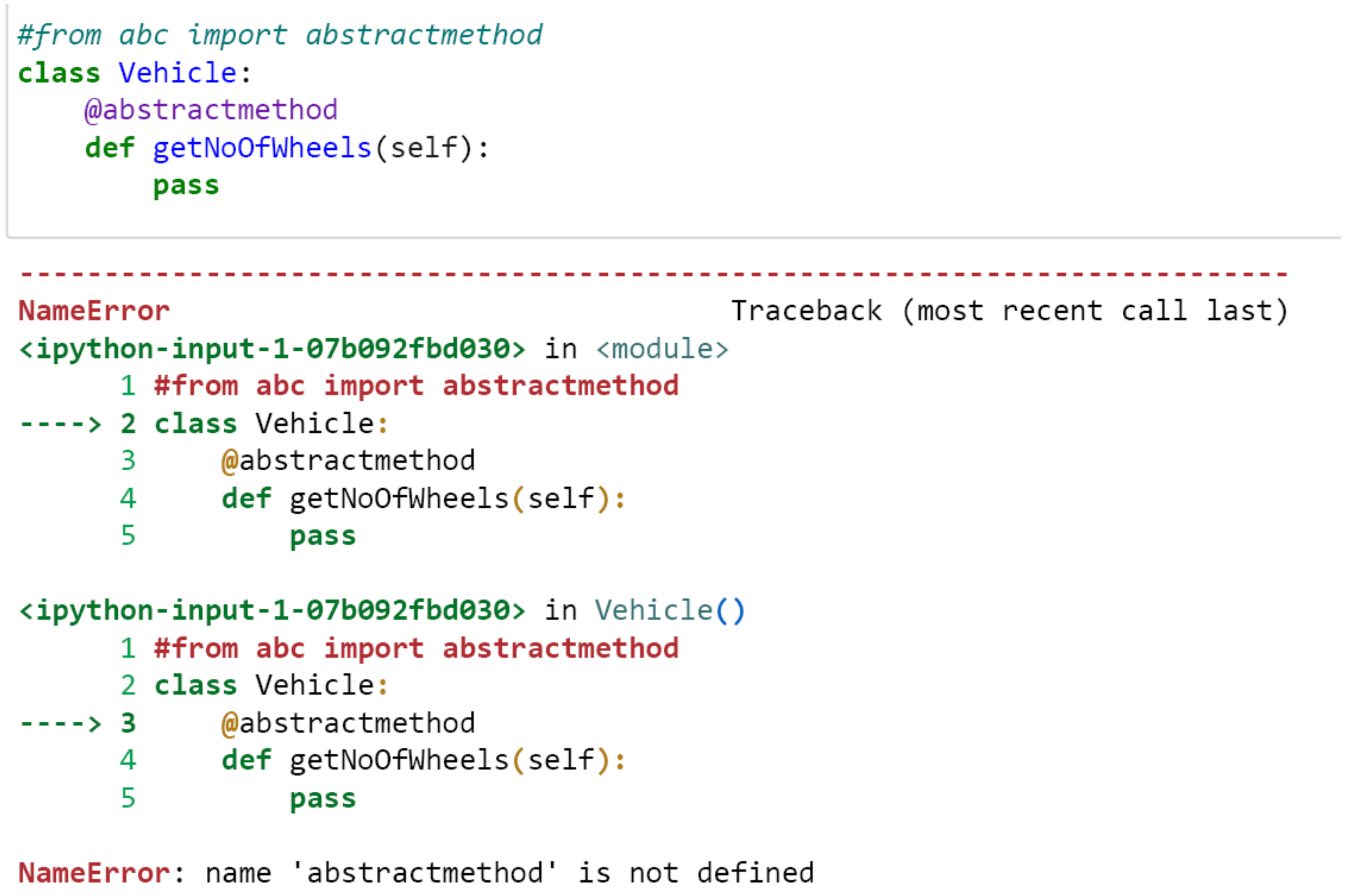
In general,

- Abstraction never talks about internal implementation.
- Abstraction means hiding something.
- Abstraction means which is not complete, just outline.

1. Abstract Method:

- Sometimes we don't know about implementation, still we can declare a method. Such type of methods are called abstract methods. (i.e., abstract method has only declaration but not implementation(i.e., empty implementation)).
- In future, in child classes, these abstract methods are required to implement.
- In Python, we should declare abstract method by using @abstractmethod decorator, which is present in abc module. Hence compulsory we should import abc module.

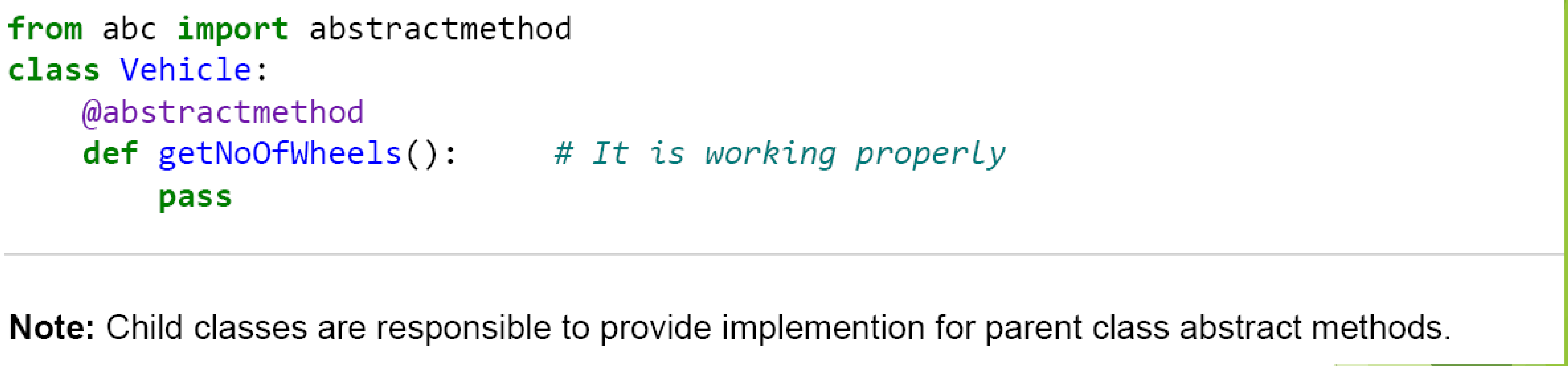
Eg:



```
#from abc import abstractmethod
class Vehicle:
    @abstractmethod
    def getNoOfWheels(self):
        pass
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-07b092fbd030> in <module>
      1 #from abc import abstractmethod
----> 2 class Vehicle:
      3     @abstractmethod
      4     def getNoOfWheels(self):
      5         pass

<ipython-input-1-07b092fbd030> in Vehicle()
      1 #from abc import abstractmethod
      2 class Vehicle:
----> 3     @abstractmethod
      4     def getNoOfWheels(self):
      5         pass

NameError: name 'abstractmethod' is not defined
```

Eg:



```python
from abc import abstractmethod
class Vehicle:
    @abstractmethod
    def getNoOfWheels():      # It is working properly
        pass
```

Note: Child classes are responsible to provide implemention for parent class abstract methods.

Frequently Asked Questions:

Q1. What is an Abstract Method?

Ans: The method which has declaration but not implemented.

Q2. Who is responsible to provide implementation?

Ans: Child class

Q3: How to declare abstract method?

Ans: By using @abstractmethod decorator.

Q4: @abstract method decorator present inside which module?

Ans: abc module

Q5: What is the advantage of declaring the abstract method in parent class?

Ans: By declaring abstract methods in the parent class, we can provide guidelines to the child classes, such that which methods child class has to implement.

Note:

- It is mandatory for the child classes to implement abstract methods of their parent class.

2. Abstract Class:

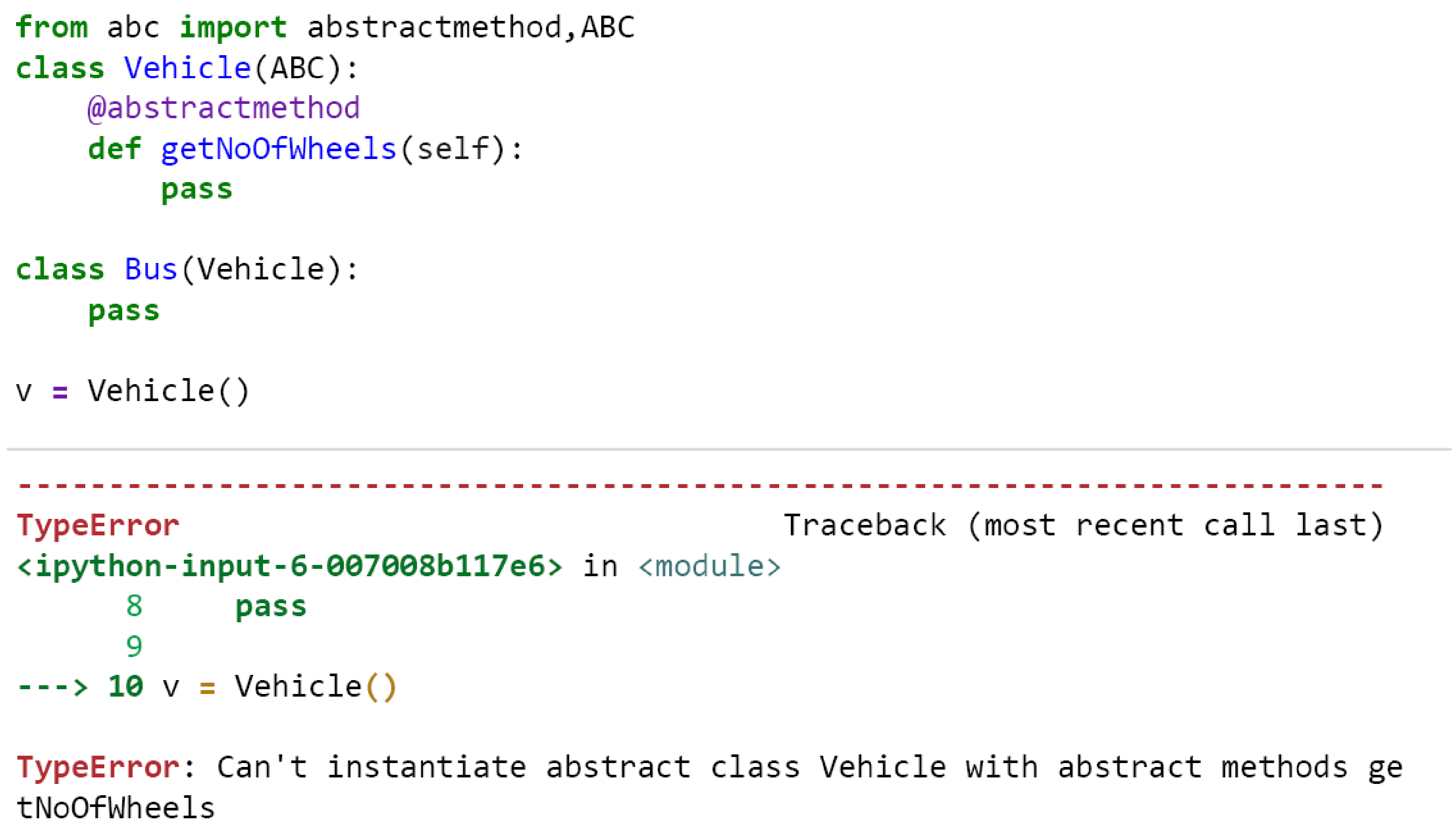
- Some times implementation of a class is not complete, such type of partially implemented classes are called abstract classes.
- Every abstract class in Python is the child class of ABC class.
- ABC means Abstract Base Class.
- Every abstract class in Python should be derived from ABC class which is present in abc module.

```
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):    #Perfectly Valid
        pass
```

Note:

- Child classes are responsible to provide the completeness of abstract classes.

Eg:



```
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):
        pass

class Bus(Vehicle):
    pass

v = Vehicle()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-007008b117e6> in <module>
      8     pass
      9 
---> 10 v = Vehicle()

TypeError: Can't instantiate abstract class Vehicle with abstract methods ge
tNoOfWheels
```

Eg:

```
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):
        pass

class Bus(Vehicle):
    pass

b = Bus()
```

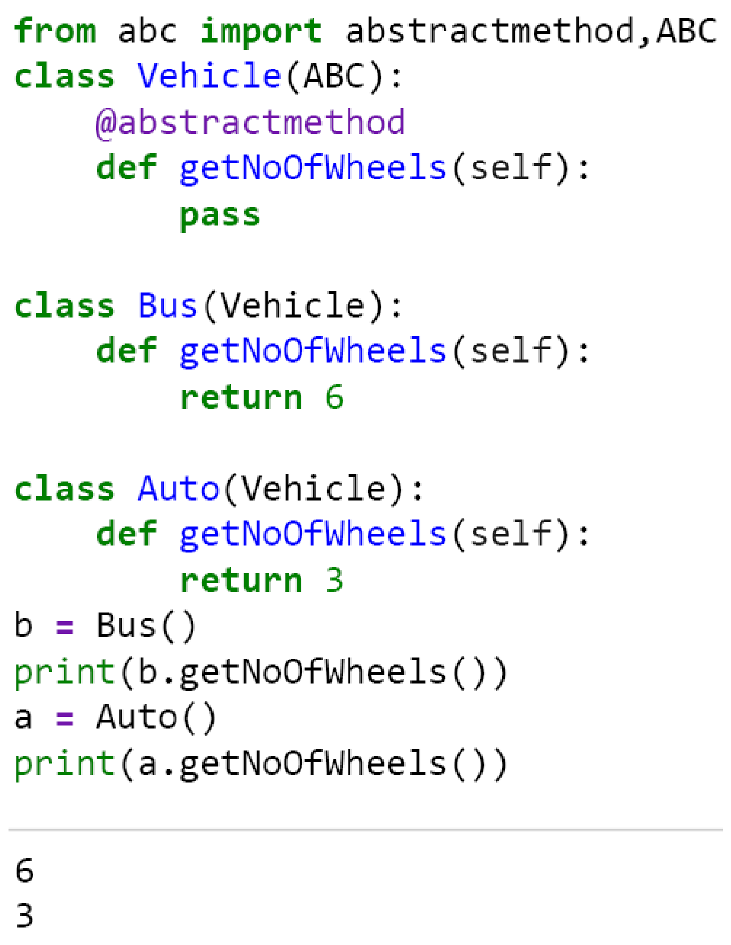
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-537a4072d511> in <module>
      8     pass
      9 
---> 10 b = Bus()

TypeError: Can't instantiate abstract class Bus with abstract methods getNoO
fWheels
```

Q. Is a class with abstract method will by default considered as abstract class?

Ans:

- No, compulsory abstract class should be declared as the child class of ABC.
- An Abstract class should be the child class of ABC and it should contain at least one abstract method. Whenever these both conditions are satisfied then only we will considered it as an abstract class.

Eg:

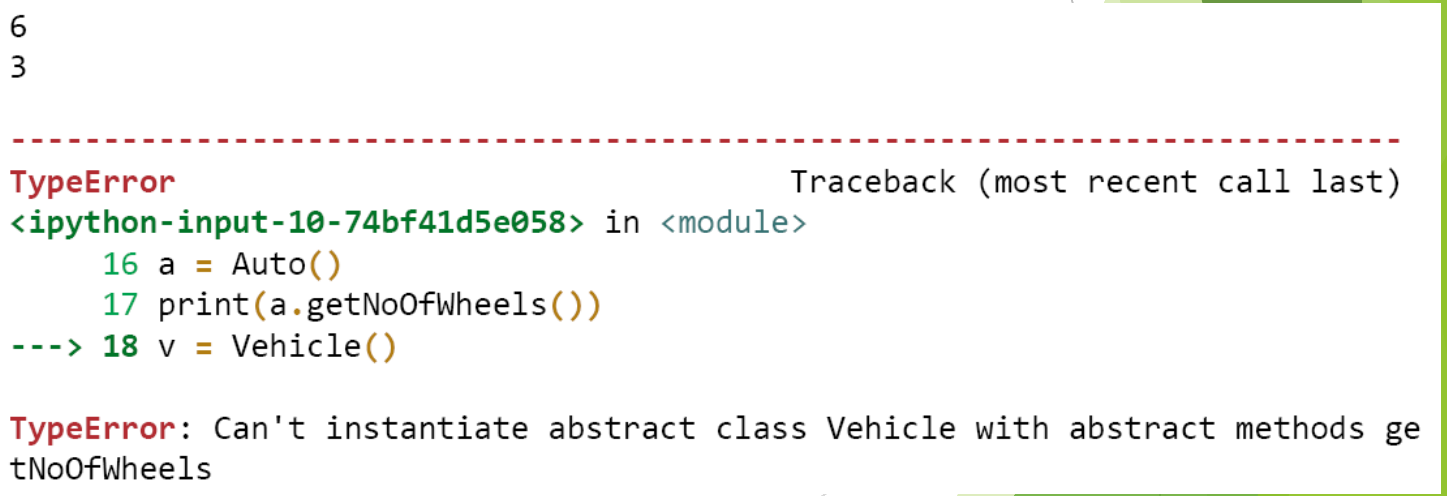


```python
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):
        pass

class Bus(Vehicle):
    def getNoOfWheels(self):
        return 6

class Auto(Vehicle):
    def getNoOfWheels(self):
        return 3
b = Bus()
print(b.getNoOfWheels())
a = Auto()
print(a.getNoOfWheels())
```

```
6
3
```

Eg:

```python
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):
        pass

class Bus(Vehicle):
    def getNoOfWheels(self):
        return 6

class Auto(Vehicle):
    def getNoOfWheels(self):
        return 3
b = Bus()
print(b.getNoOfWheels())
a = Auto()
print(a.getNoOfWheels())
v = Vehicle()
```



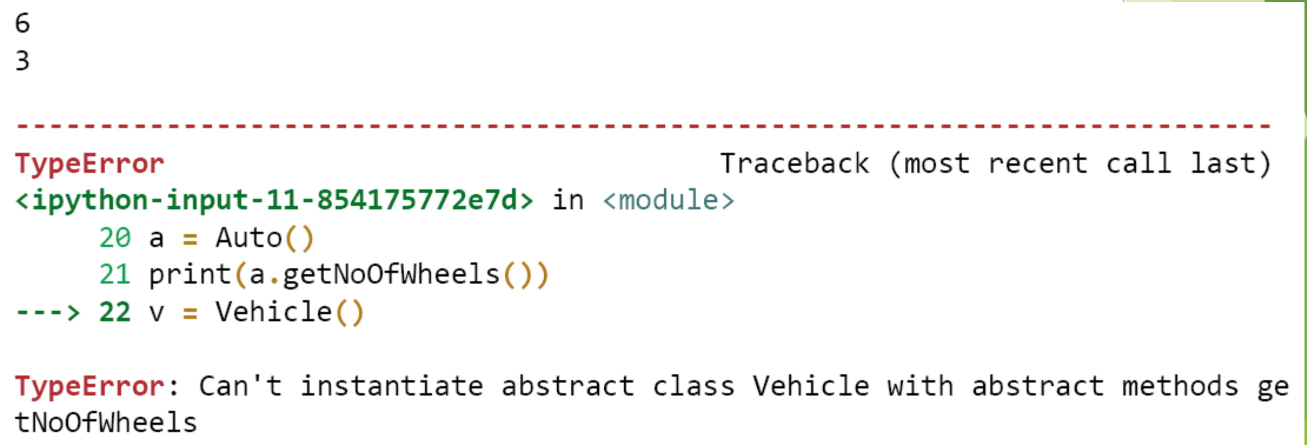
```
6
3

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-74bf41d5e058> in <module>
     16 a = Auto()
     17 print(a.getNoOfWheels())
---> 18 v = Vehicle()

TypeError: Can't instantiate abstract class Vehicle with abstract methods ge
tNoOfWheels
```

Q. In the previous example, suppose Vehicle class has few fully implemented methods, then, can we instantiate [i.e., can we create an object] vehicle class?

Ans: No

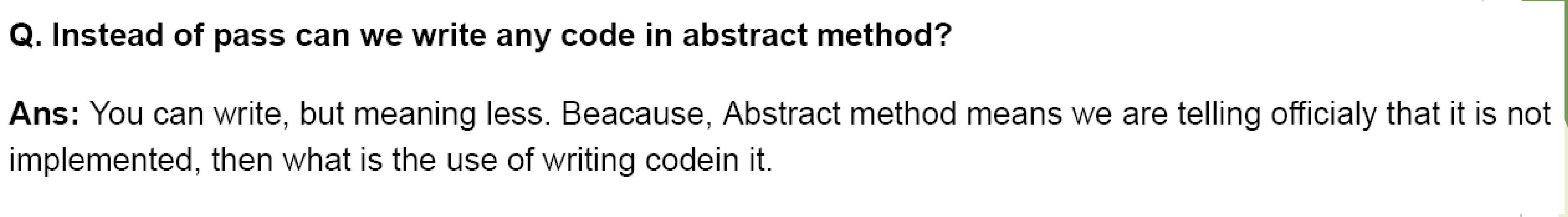
Eg:

```python
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):
        pass
    def m1(self):
        print('Vehicle class object created')
    def m2(self):
        print('Vehicle class object created')

class Bus(Vehicle):
    def getNoOfWheels(self):
        return 6

class Auto(Vehicle):
    def getNoOfWheels(self):
        return 3
b = Bus()
print(b.getNoOfWheels())
a = Auto()
print(a.getNoOfWheels())
v = Vehicle()
```



```
6
3

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-854175772e7d> in <module>
     20 a = Auto()
     21 print(a.getNoOfWheels())
---> 22 v = Vehicle()

TypeError: Can't instantiate abstract class Vehicle with abstract methods ge
tNoOfWheels
```

Key Point:



Q. Instead of pass can we write any code in abstract method?

Ans: You can write, but meaning less. Beacause, Abstract method means we are telling officialy that it is not implemented, then what is the use of writing codein it.

Eg:

```
from abc import abstractmethod,ABC
class Vehicle(ABC):
    @abstractmethod
    def getNoOfWheels(self):
        print('Hai')
    def m1(self):
        print('Vehicle class object created')
    def m2(self):
        print('Vehicle class object created')

class Bus(Vehicle):
    def getNoOfWheels(self):
        return 6

class Auto(Vehicle):
    def getNoOfWheels(self):
        return 3
b = Bus()
print(b.getNoOfWheels())
a = Auto()
print(a.getNoOfWheels())
```

```
6
3
```

Q. Abstract class (i.e., Vehicle class in the above example) should extend ABC class, but if Bus class doesn't implement the abstract method, then Bus class also become abstract class, then it should extend ABC class also?

Ans:

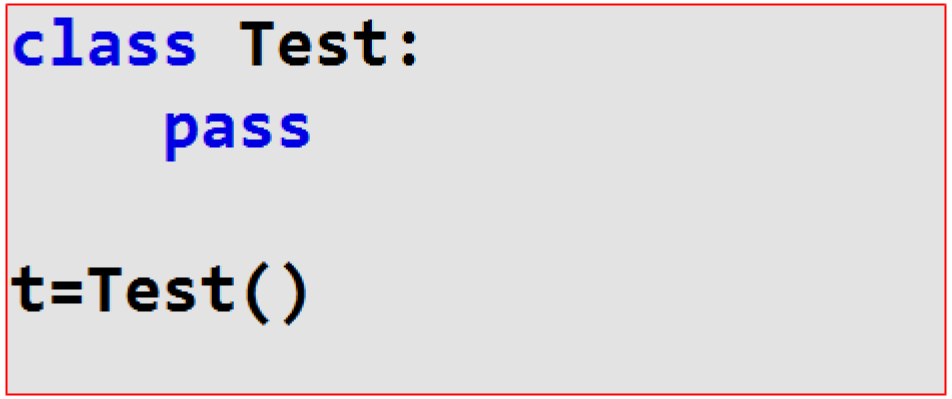
Yes. It should extend ABC class either directly or indirectly.

Important Conclusions of Abstract Method & Abstract Class

Conclusion 1:

- If a class contains at least one abstract method and if we are extending ABC class then it’s instantiation is not possible. In simple words, "For Abstract class with abstract methods, instantiation is not possible".
- Now, with the help of the following scenarios, we will try to understand more information about syntactical loopholes of abstract methods and abstract classes.

Case 1:



```
class Test:
    pass

t=Test()
```

- We can create Test class object, because it is concrete class (not an abstract class) and does not contain any abstract method. So, it is perfectly acceptable.

Eg:

```python
class Test:
    pass

t = Test()        # It is valid
```

Case 2:

```
from abc import *  #In this case it is not required
class Test:    # concrete class (Not an abstract class)
    pass
t =Test()        #Perfectly valid code
```

Analysis:

- In the above code, we can create object for Test class because it is concrete class and it does not contain any abstract method.

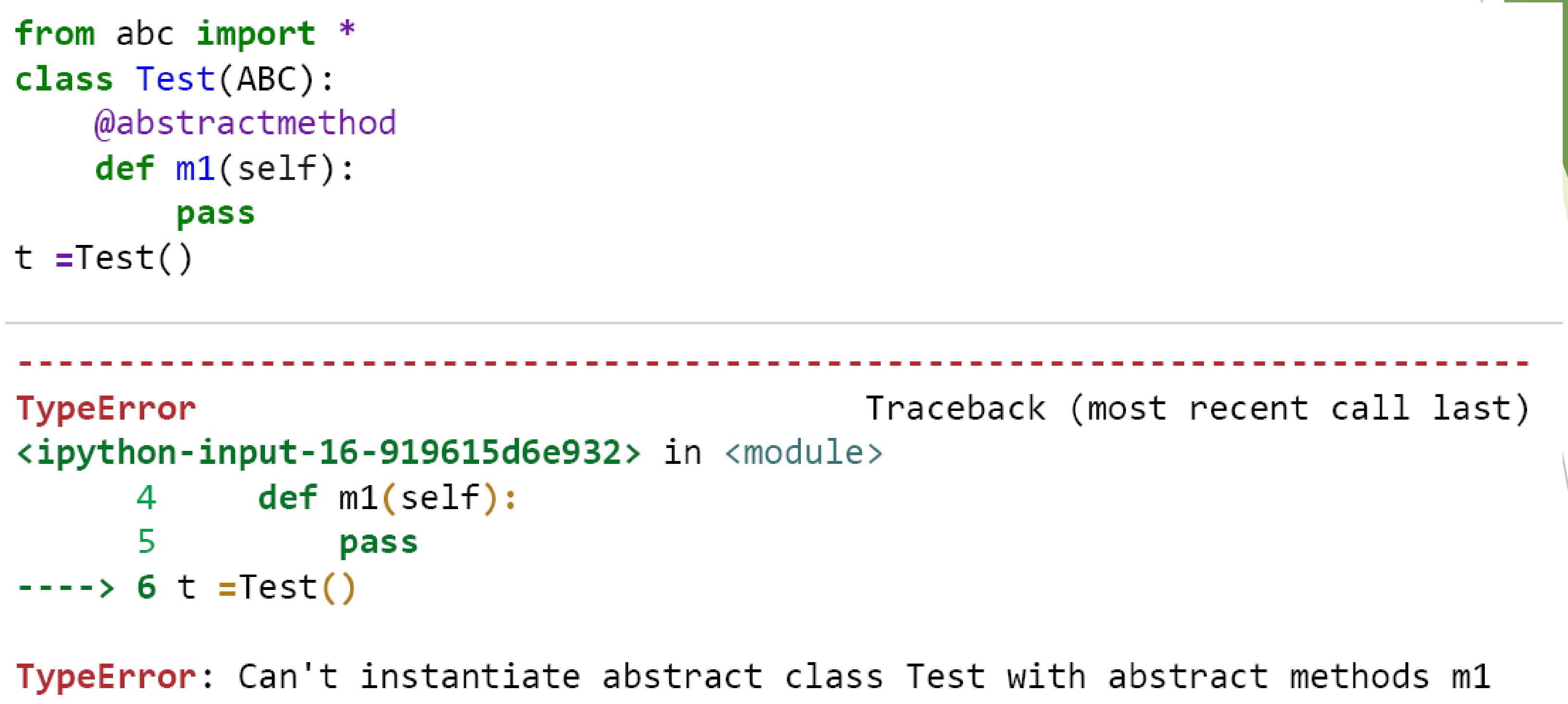
Case 3:

```
from abc import *
class Test(ABC):
    pass
t =Test()          #It is meaning less, but Perfectly valid code
```

Analysis:

- An Abstract class contain zero or more abstract methods. If the abstract class does not contain any abstract method, then we can create an object.

Case 4:



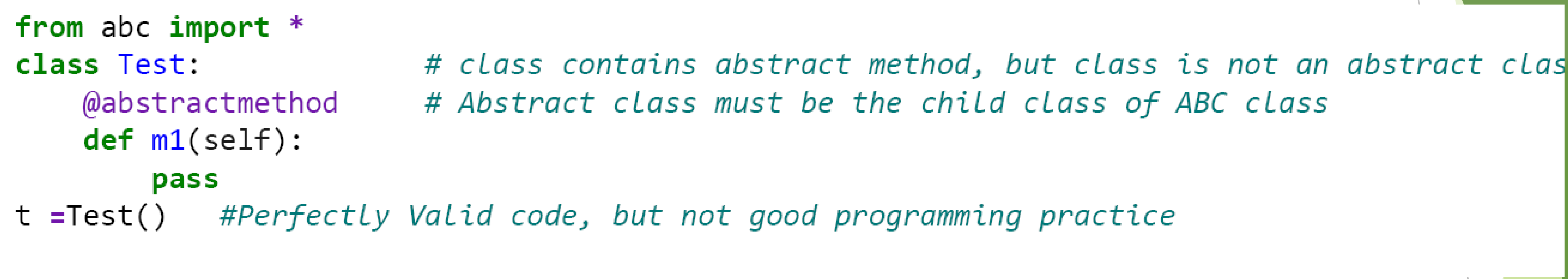
```python
from abc import *
class Test(ABC):
    @abstractmethod
    def m1(self):
        pass
t =Test()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-919615d6e932> in <module>
      4     def m1(self):
      5         pass
----> 6 t =Test()

TypeError: Can't instantiate abstract class Test with abstract methods m1
```

Analysis:

- If the class is the child class of ABC and contain at least one abstract method, then it is not possible to create an object. For abstract classes with at least one abstract method, instantiation is not possible.

Case 5:

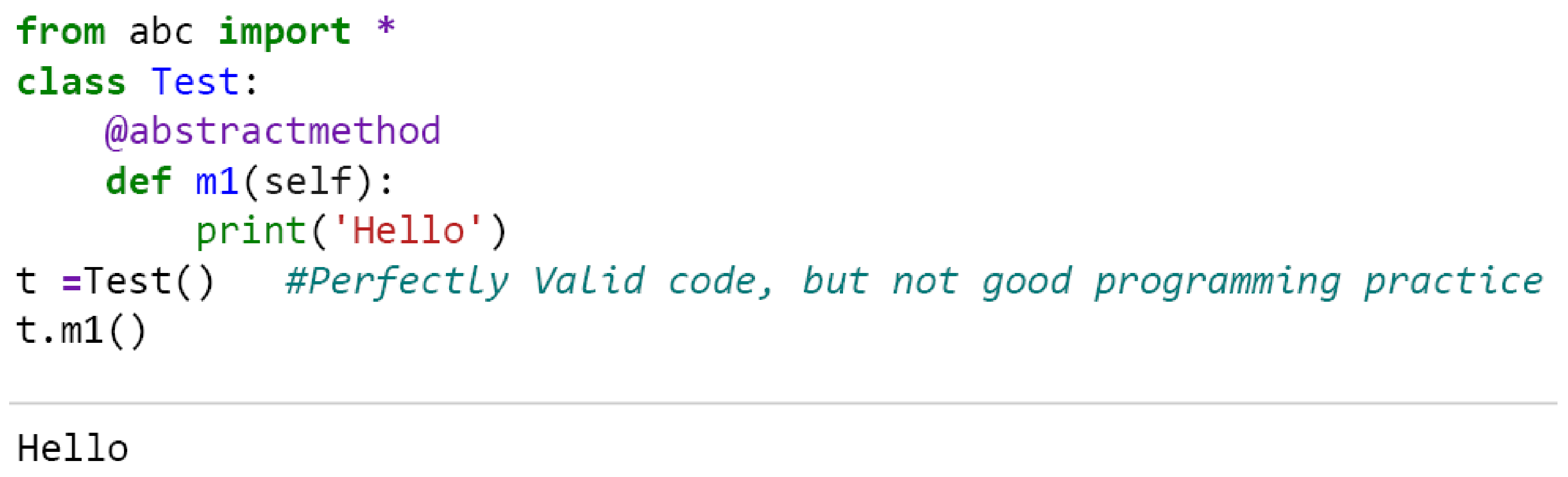


```
from abc import *
class Test:             # class contains abstract method, but class is not an abstract clas
    @abstractmethod     # Abstract class must be the child class of ABC class
    def m1(self):
        pass
t =Test()   #Perfectly Valid code, but not good programming practice
```

Analysis:

- We can create object even class contains abstract method because we are not extending ABC class.

Case 6:



```python
from abc import *
class Test:
    @abstractmethod
    def m1(self):
        print('Hello')
t =Test()   #Perfectly Valid code, but not good programming practice
t.m1()
```

```
Hello
```

Analysis:

- If the class is child class of ABC and if it contain at least one abstract method, then only instantiation is not possible.
- If the class is not the child class of ABC (or) It doesn't contain any abstract method, then instantiation is always possible.

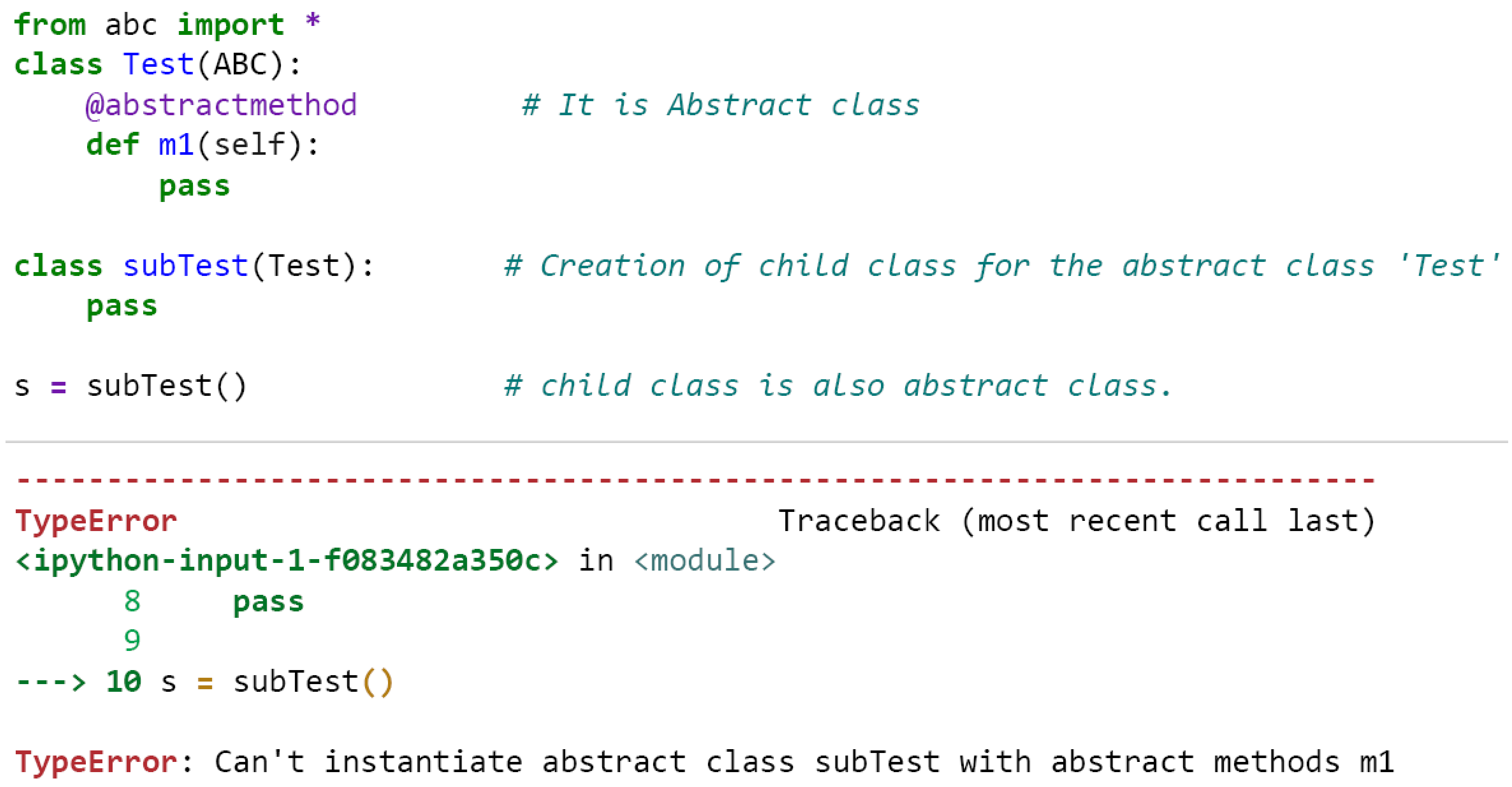
Note:

- Instantiation means Object creation.
- In the previous example, We are explicitly saying that the method 'm1()' is an abstract method. So code is valid, but not good programming practice. Because we are calling the method, if you feel that it's implementation is complete, then why we are explicitly saying that it is an abstract method.

Conclusion 2:

- If we are creating child class for abstract class, then for every abstract method of parent class compulsory we should provide implementation in the child class, otherwise child class is also abstract and we cannot create object for child class.

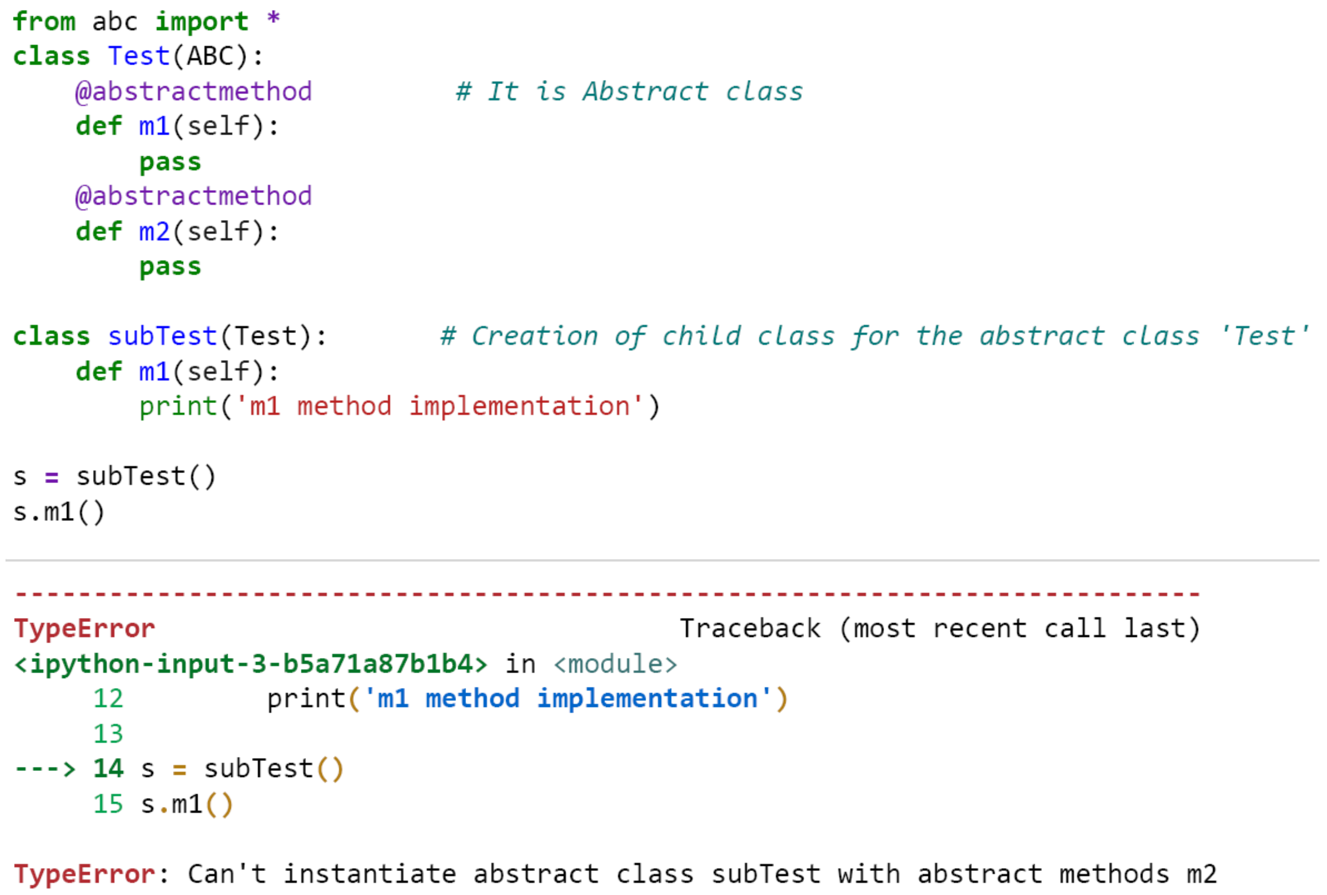
Case 1:



```
from abc import *
class Test(ABC):
    @abstractmethod           # It is Abstract class
    def m1(self):
        pass

class subTest(Test):         # Creation of child class for the abstract class 'Test'
    pass

s = subTest()                # child class is also abstract class.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-f083482a350c> in <module>
      8     pass
      9 
---> 10 s = subTest()

TypeError: Can't instantiate abstract class subTest with abstract methods m1
```

Case 2:



```python
from abc import *
class Test(ABC):
    @abstractmethod         # It is Abstract class
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass

class subTest(Test):       # Creation of child class for the abstract class 'Test'
    def m1(self):
        print('m1 method implementation')

s = subTest()
s.m1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-b5a71a87b1b4> in <module>
     12         print('m1 method implementation')
     13 
---> 14 s = subTest()
     15 s.m1()

TypeError: Can't instantiate abstract class subTest with abstract methods m2
```

Case 3:

```
from abc import *
class Test(ABC):
    @abstractmethod         # It is Abstract class
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass

class subTest(Test):       # Creation of child class for the abstract class 'Test'
    def m1(self):
        print('m1 method implementation')
    def m2(self):
        print('m2 method implementation')
s = subTest()
s.m1()
s.m2()
```

```
m1 method implementation
m2 method implementation
```

Case 4:

- An Abstract class can contain both abstract and non-abstract methods also.

```
from abc import *
class Test(ABC):

    def m1(self):
        print('It is Non-Abstract Method')
    @abstractmethod
    def m2(self):
        pass

class subTest(Test):

    def m2(self):
        print('m2 method implementation')
s = subTest()
s.m1()
s.m2()
```

```
It is Non-Abstract Method
m2 method implementation
```

Another Example:

```
from abc import *
class P(ABC):
    def m1(self):
        print('Hello')
    @abstractmethod
    def m2(self):
        pass
    def m3(self):
        pass
class C(P):
    def m2(self):
        print('m2 method implementation')
c=C()
c.m1()
c.m2()
c.m3()
```

```
Hello
m2 method implementation
```

Analysis:

- If we are creating child class for any abstract class, compulsory we should provide implementation for every abstract method of parent class.
- If we are not implementing atleast one method, then child class also will become abstract class. So, we can't create an object to the child class.

Q. Is a class can contain both abstract and non-abstract methods?

Ans: YES

Note:

- If a class contains both abstract and non-abstract methods then child class is responsible to provide implementation only for abstract methods.
- In Java, these many loop holes are not there. Once if you explicitly state that a class is abstract class, then you can't create objects irrespective of whether that class contains abstract method or not.

3. Interfaces in Python

Key Points about Interface:

- We can implement interface concept in Python with abstract class only.
- An abstract class can contains both abstract and non-abstract methods.
- If an abstract class contains only abstract methods, such type of abstract class is nothing but interface.
- 100% pure abstract class is nothing but interface.
- Interface simply acts as Service Requirement Specification(SRS).

Eg:

```
class Test(ABC):

    def m1(self):
        print('Hi')             # Non abstract method

    @abstractmethod
    def m2(self):              # Abstract method
        pass
```

Analysis:

The above class is an abstract class, but not considered as an Interface. Because it contains a non abstract method.

Consider another example,

```
class Test(ABC):

    @abstractmethod
    def m1(self):
        print('Hi')              # Abstract method

    @abstractmethod
    def m2(self):               # Abstract method
        pass
```

Analysis:

The above class is an abstract class, and also considered as an Interface. Because it contains only abstract methods.

What is the purpose of Interface ???

- Interfaces are used to represent service requirements specification.

Let us assume that, a client requires a College Automation System with the following required services.

1. getStudentsAttendance()
2. updateStudentsAttendance()
3. getMarks()
4. updateMarks()
5. getFeeInfo()
6. updateFeeInfo()

- In the previous scenario, College Automation System is an abstract class and every method (i.e., required service) is considered as an abstract method. Representing this type of requirements specification itself is called as an Interface.

Executable code for the previous case:

```
from abc import *

class CollegeAutomation(ABC):

    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass
    @abstractmethod
    def m3(self):
        pass
    @abstractmethod         # It is an Interface
    def m4(self):
        pass
    @abstractmethod
    def m5(self):
        pass
    @abstractmethod
    def m6(self):
        pass
```

The client requirements are representing by using Intrface.
A single interface may contain multiple implementations by multiple people (or compaies) in their way.

For example, see the below code:

```
from abc import *

class CollegeAutomation(ABC):

    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass
    @abstractmethod
    def m3(self):
        pass
    @abstractmethod
    def m4(self):
        pass
    @abstractmethod
    def m5(self):
        pass
    @abstractmethod
    def m6(self):
        pass
```

```python
class RGMimpl(CollegeAutomation):
    def m1(self):
        print('Get Attendance Information')
    def m2(self):
        print('Update Attendance Information')
    def m3(self):
        print('Get marks Information')
    def m4(self):
        print('Update Marks Information')
    def m5(self):
        print('Get Fee Information')
    def m6(self):
        print('Update Fee Information')
```

```
r = RGMimpl()
r.m1()
r.m2()
r.m3()
r.m4()
r.m5()
r.m6()
```

```
Get Attendance Information
Update Attendance Information
Get marks Information
Update Marks Information
Get Fee Information
Update Fee Information
```

Note:

- Interface acts as Prototype to implement a particular software.

Interface vs Abstract class vs Concrete class

1. If we don't know anything about implementation and just we have requirement specification then we should go for interface (SRS-Service Requirement Specification).

2. If we are talking about implementation but not completely then we should go for abstract class (Partially implemented class).

3. If we are talking about implementation completely and ready to provide service then we should go for concrete class (Fully Implemented class).

Note:

- In Python, by using abstract class concept, we can implement both abstract classes and Interfaces concepts.
- In Java, both abstract classes and interfaces are separate concepts.

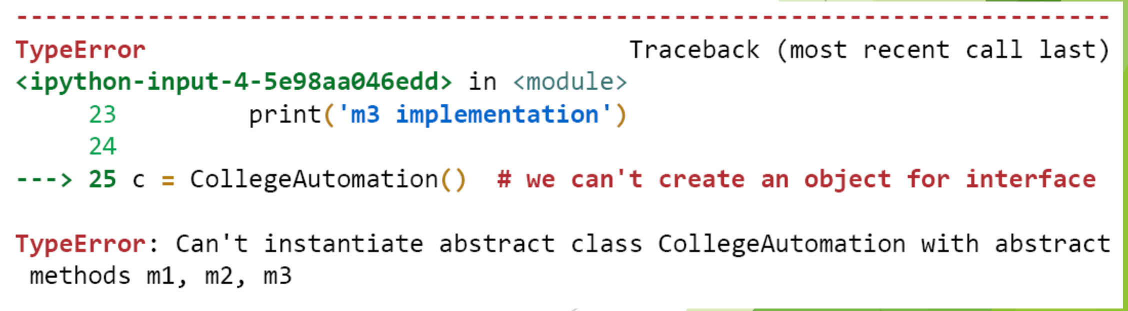
Eg:

```python
from abc import *

class CollegeAutomation(ABC):

    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):                       # Interface
        pass
    @abstractmethod
    def m3(self):
        pass

class AbsClass(CollegeAutomation):
    def m1(self):
        print('m1 implementation')      # Abstract Class
    def m2(self):
        print('m2 implementation')

class ConcreteClass(AbsClass):
    def m3(self):                       # Concrete Class
        print('m3 implementation')

c = ConcreteClass()
c.m1()
c.m2()
c.m3()
```

```
m1 implementation
m2 implementation
m3 implementation
```

Eg:

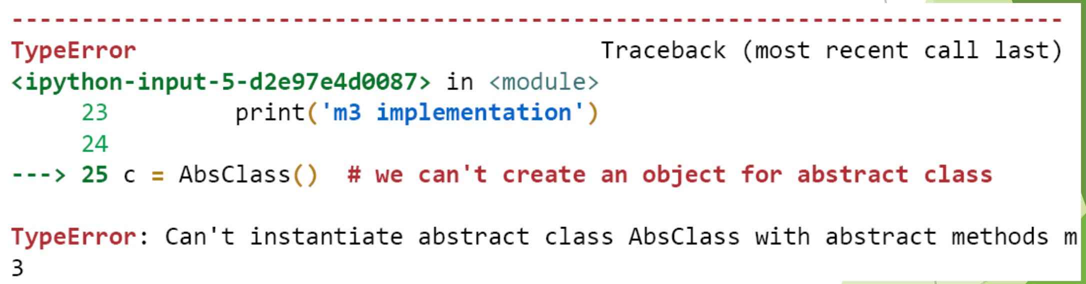
```python
from abc import *

class CollegeAutomation(ABC):

    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):                       # Interface
        pass
    @abstractmethod
    def m3(self):
        pass

class AbsClass(CollegeAutomation):
    def m1(self):
        print('m1 implementation')      # Abstract Class
    def m2(self):
        print('m2 implementation')

class ConcreteClass(AbsClass):
    def m3(self):                       # Concrete Class
        print('m3 implementation')

c = CollegeAutomation()  # we can't create an object for interface
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-5e98aa046edd> in <module>
     23         print('m3 implementation')
     24 
---> 25 c = CollegeAutomation()  # we can't create an object for interface

TypeError: Can't instantiate abstract class CollegeAutomation with abstract
 methods m1, m2, m3
```

Eg:

```python
from abc import *

class CollegeAutomation(ABC):

    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):                           # Interface
        pass
    @abstractmethod
    def m3(self):
        pass

class AbsClass(CollegeAutomation):
    def m1(self):
        print('m1 implementation')      # Abstract Class
    def m2(self):
        print('m2 implementation')

class ConcreteClass(AbsClass):
    def m3(self):                           # Concrete Class
        print('m3 implementation')

c = AbsClass()  # we can't create an object for abstract class
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-d2e97e4d0087> in <module>
     23         print('m3 implementation')
     24 
---> 25 c = AbsClass()  # we can't create an object for abstract class

TypeError: Can't instantiate abstract class AbsClass with abstract methods m
3
```

Note:

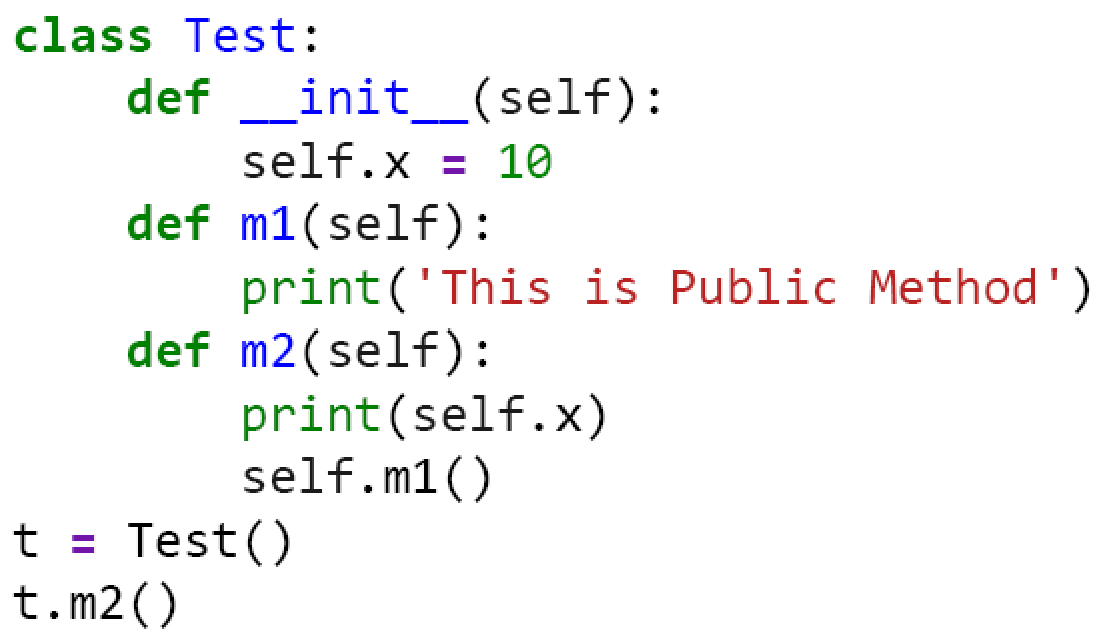
- You can create object for only concrete classes.

4. Public, Private and Protected Members

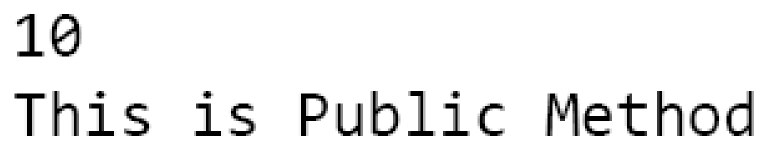
i. Public Members:

- If a member(either method or variable) of a class is public, then we can access that member from anywhere either within the class or from outside of the class.
- By default every member present in python class is public. Hence we can access from anywhere either within the class or from outside of the class.

Demo Program on Accessing the public members from within the class:



```
class Test:
    def __init__(self):
        self.x = 10
    def m1(self):
        print('This is Public Method')
    def m2(self):
        print(self.x)
        self.m1()
t = Test()
t.m2()
```



```
10
This is Public Method
```

Demo Program on Accessing the public members from outside the class:

```
class Test:
    def __init__(self):
        self.x = 10
    def m1(self):
        print('This is Public Method')
    def m2(self):
        print(self.x)
        self.m1()
t = Test()
t.m2()
print(t.x)
t.m1()
```

```
10
This is Public Method
10
This is Public Method
```

ii. Private Members:

- If a member is private, then we can access that member only with in the class and from outside of the class we cannot access.
- We can declare a member as private explicitly by prefixing with two underscore symbols.

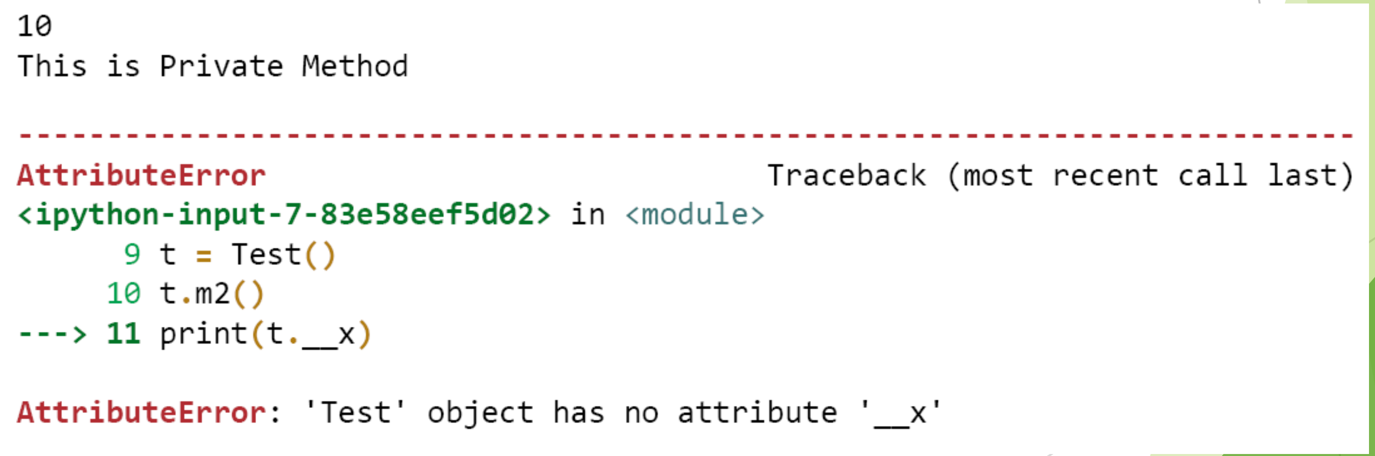
Demo Program on accessing the private members from within the class:

```
class Test:
    def __init__(self):
        self.__x = 10      # Private variable
    def __m1(self):        # Private method
        print('This is Private Method')
    def m2(self):
        print(self.__x)
        self.__m1()
t = Test()
t.m2()
```

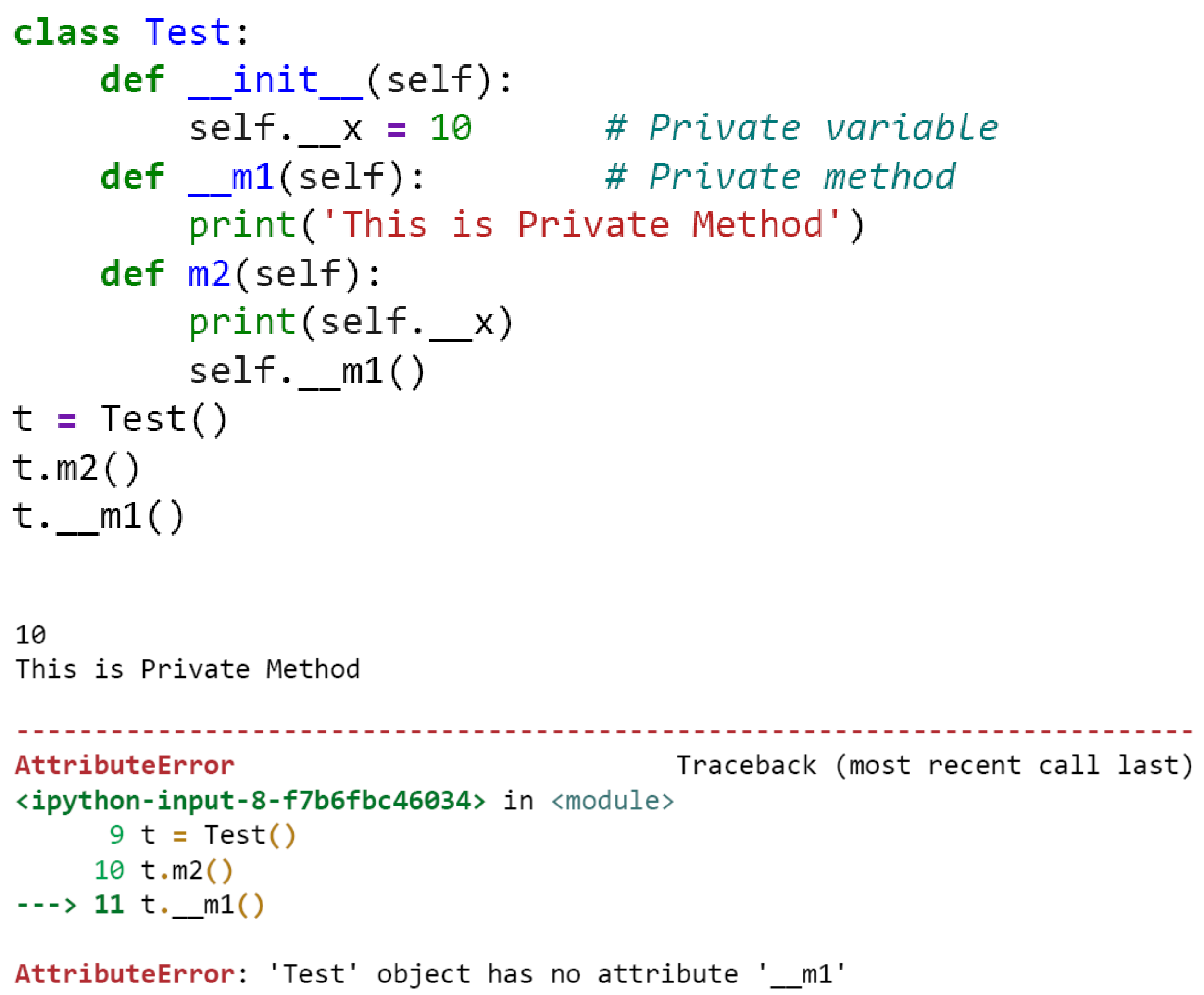
```
10
This is Private Method
```

Demo Program on accessing the private members from outside the class:

```python
class Test:
    def __init__(self):
        self.__x = 10       # Private variable
    def __m1(self):         # Private method
        print('This is Private Method')
    def m2(self):
        print(self.__x)
        self.__m1()
t = Test()
t.m2()
print(t.__x)
```



```
10
This is Private Method

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-83e58eef5d02> in <module>
      9 t = Test()
     10 t.m2()
---> 11 print(t.__x)

AttributeError: 'Test' object has no attribute '__x'
```



```
class Test:
    def __init__(self):
        self.__x = 10       # Private variable
    def __m1(self):         # Private method
        print('This is Private Method')
    def m2(self):
        print(self.__x)
        self.__m1()
t = Test()
t.m2()
t.__m1()
```

```
10
This is Private Method

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-f7b6fbc46034> in <module>
      9 t = Test()
     10 t.m2()
---> 11 t.__m1()

AttributeError: 'Test' object has no attribute '__m1'
```

Name Mangling and Accessing private members from outside of the class

- We cannot access private members directly from outside of the class. But we can access indirectly as follows.
- Name Mangling will be happened for the private variables. Hence every private variable name will be changed to new name.

_ _VariableName ===> _ClassName _ _VariableName

Hence, we can access private variable from outside of the class as follows:

print(objectreference._classname__variablename)

Eg:

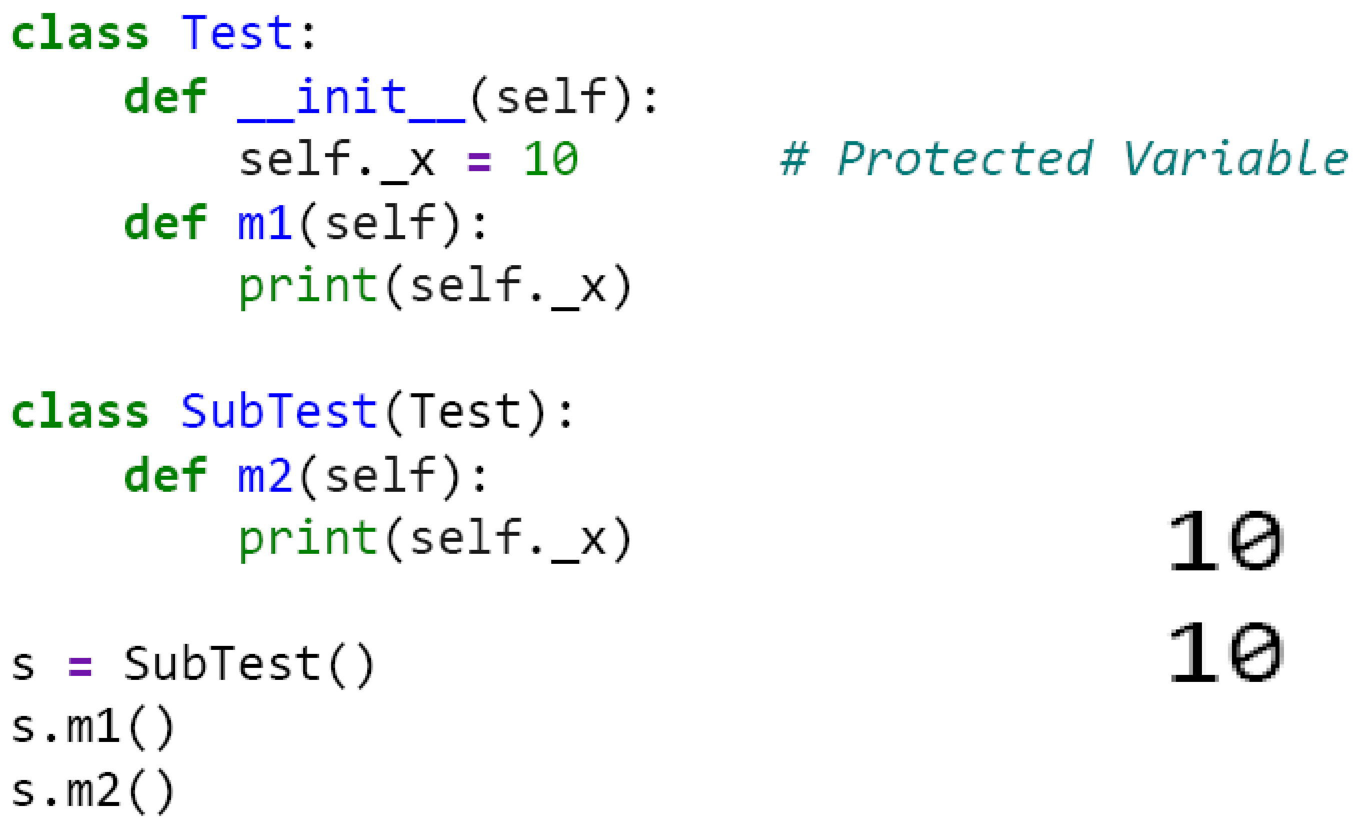
```
class Test:
    def __init__(self):
        self.__x = 10       # Private variable
    def __m1(self):         # Private method
        print('This is Private Method')
    def m2(self):
        print(self.__x)
        self.__m1()
t = Test()
print(t._Test__x) # Name Mangling concept
t._Test__m1()
```

```
10
This is Private Method
```

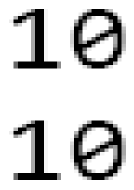
iii. Protected Members:

- Protected members can access within the class from anywhere but from outside of the class we can access only in child classes.
- We can declare members as protected explicitly by prefixing with one underscore symbol.

Demo Program on accessing the protected members:



```python
class Test:
    def __init__(self):
        self._x = 10       # Protected Variable
    def m1(self):
        print(self._x)

class SubTest(Test):
    def m2(self):
        print(self._x)

s = SubTest()
s.m1()
s.m2()
```



```
10
10
```

Eg:

```
class Test:
    def __init__(self):
        self._x = 10        # Protected Variable
    def m1(self):
        print(self._x)

class SubTest(Test):
    def m2(self):
        print(self._x)

s = SubTest()
s.m1()
s.m2()
print(s._x)             # Note: We can access from outside child class also.
```

```
10
10
10
```

Note:

These 3 access specifiers are just naming conventions and it is not strictly implemented in python (may be for the future versions purpose).

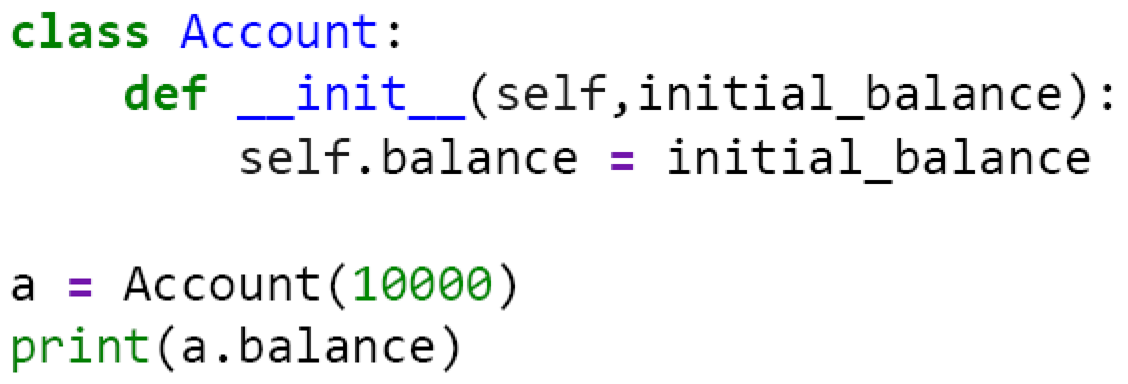
You saw already violation in the above example.

Key Terms related to Objet Oriented Programing

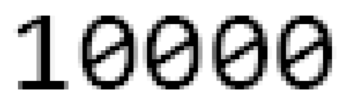
1. Data Hiding:

- Our internal data should not go out directly. (i.e., outside person should not access our internal data directly).
- This OOP feature is nothing but data hiding.
- By declaring data members as private, we can implement Data Hiding.

Eg:



```
class Account:
    def __init__(self,initial_balance):
        self.balance = initial_balance

a = Account(10000)
print(a.balance)
```

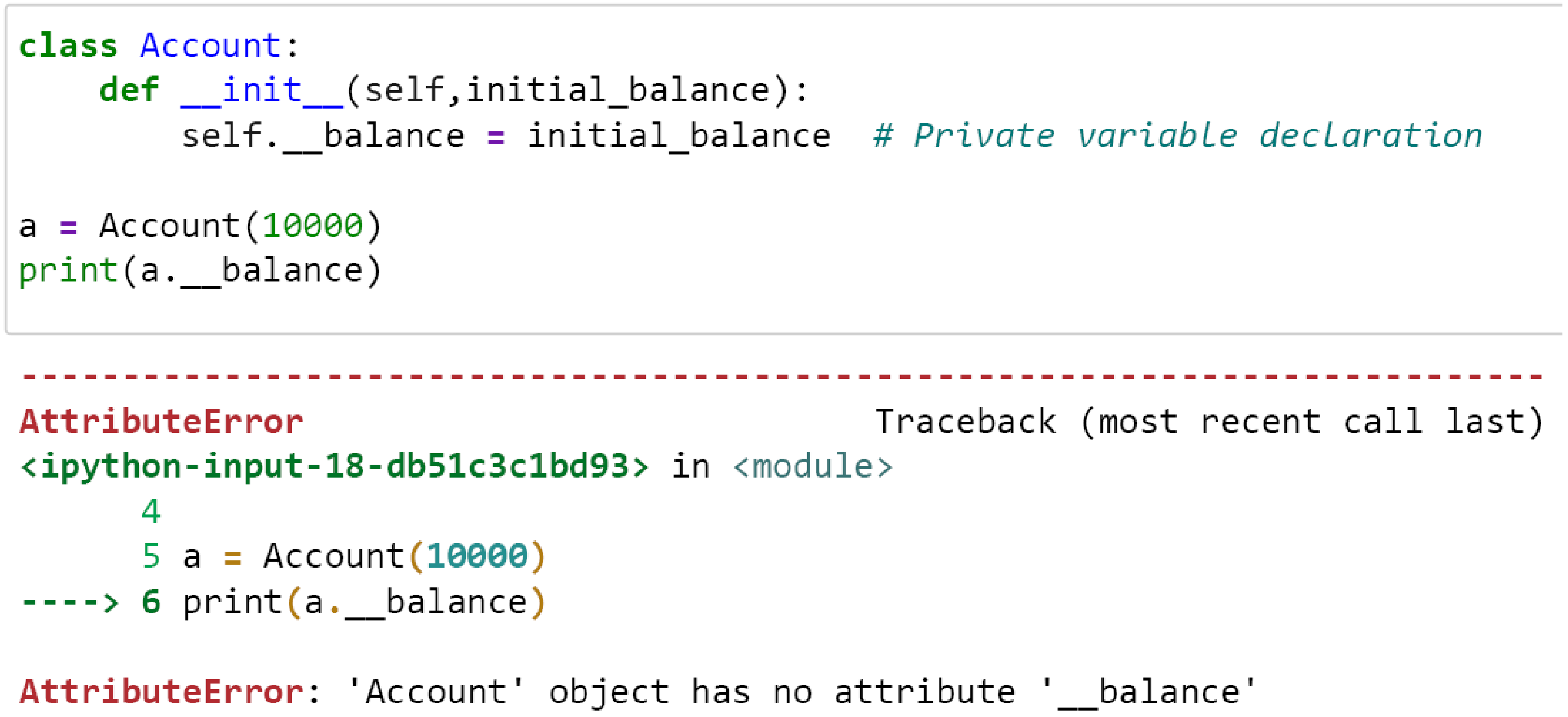


```
10000
```

Analysis:

- In the above code, data hiding is not there, any object directly access the data (i.e.,initial_balance). So, security issues may rise.
- This type of programming is not a good programming practice, because outside person (i.e.,any object) can't access data directly.
- By declaring the variable (i.e.,initial_balance) as private variable we can prevent this directly accessing of data.

Eg:



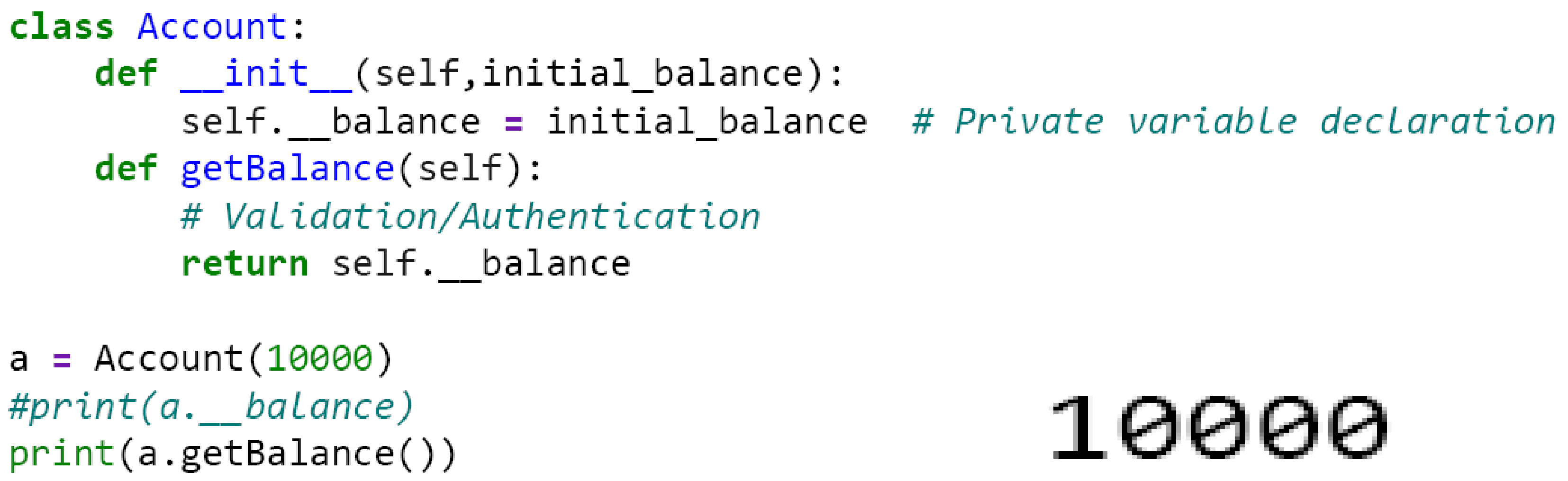
```
class Account:
    def __init__(self,initial_balance):
        self.__balance = initial_balance  # Private variable declaration

a = Account(10000)
print(a.__balance)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-18-db51c3c1bd93> in <module>
      4
      5 a = Account(10000)
----> 6 print(a.__balance)

AttributeError: 'Account' object has no attribute '__balance'
```

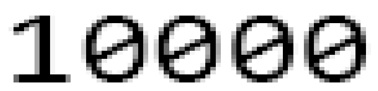
Now the Question is, How can we access the data?

- By using method, we can access the data. You have to call a method, through that method, you can get access to our data.

Eg:



```
class Account:
    def __init__(self,initial_balance):
        self.__balance = initial_balance  # Private variable declaration
    def getBalance(self):
        # Validation/Authentication
        return self.__balance

a = Account(10000)
#print(a.__balance)
print(a.getBalance())
```



```
10000
```

What is the advantage of Data Hiding?

- Security is the biggest advantage of Data hiding. Because, no one is allowed to modify our data. Authorized person only can access the data.

2. Abstraction:

Def: Hiding internal implementation and just highlight the set of services is the concept of abstraction.

Eg:

- Through Bank ATM GUI Screen, Bank people are highlighting the set of services what they are offering without highlighting internal implementation. This is nothing but abstraction.

How to implement Abstraction?

- By using GUI Screens, Application Programming Interfaces (APIs),Abstract classes and Interfaces, we can implement abstraction.

Advantages of Abstraction:

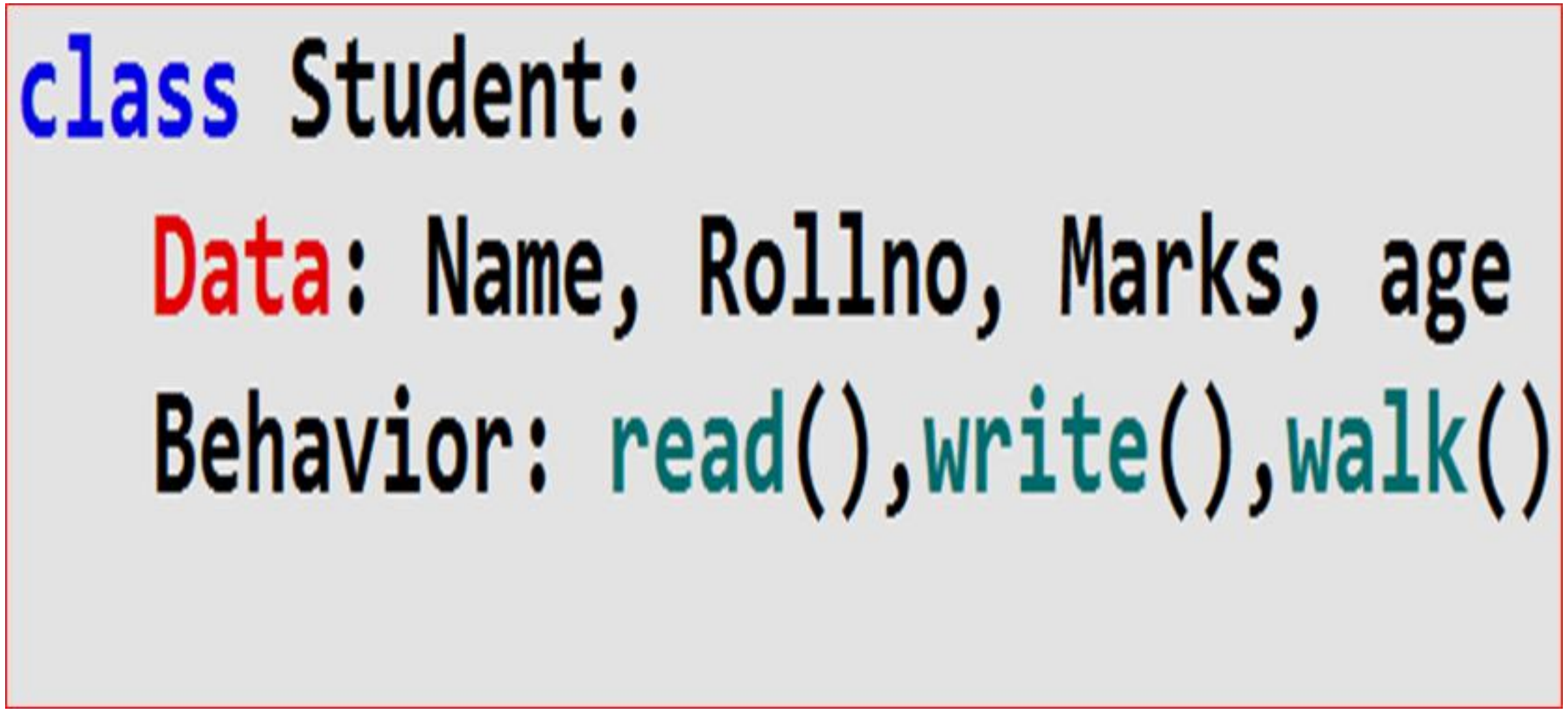
1. Security.
2. Enhancement will become very easy.
3. Maintainability and Modularity of the application.

3. Encapsulation:

Def: The process of binding (or) grouping of related things into a single unit is called as Encapsulation.

For example, consider a Medical Capsule,

For example, consider a Programming Capsule,



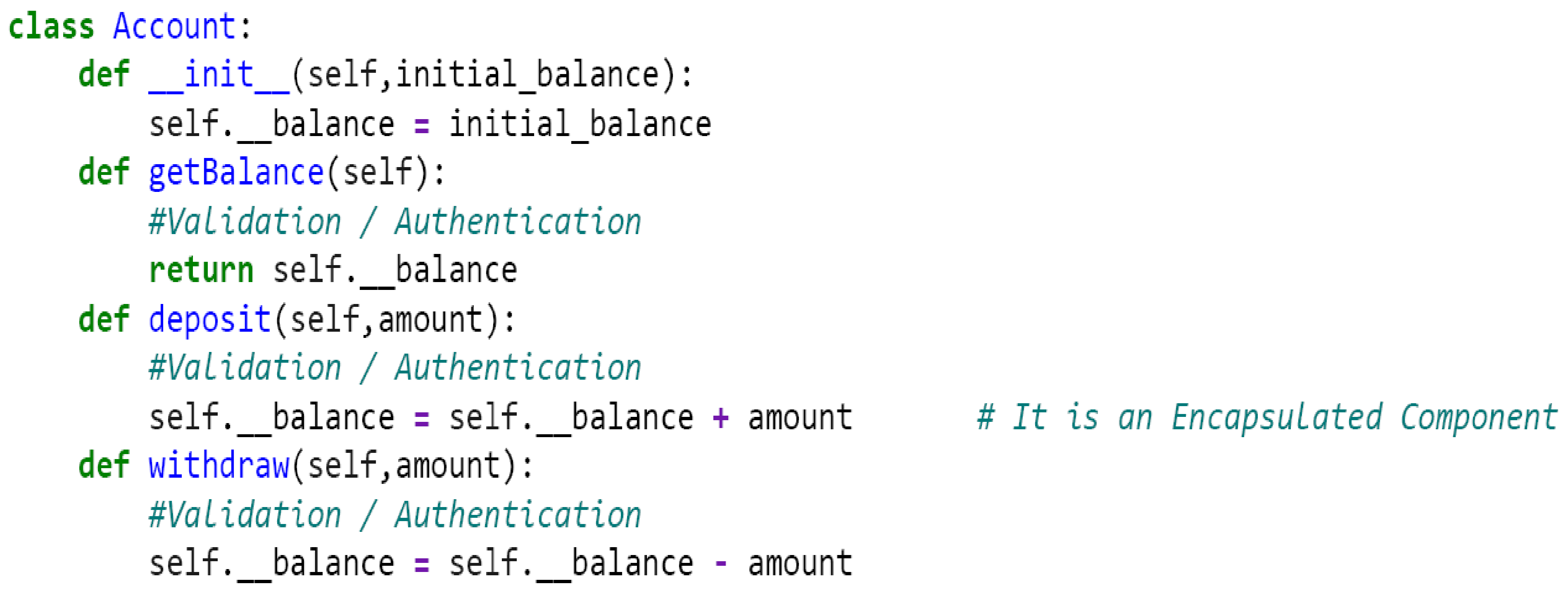
```
class Student:
    Data: Name, Rollno, Marks, age
    Behavior: read(),write(),walk()
```

Key Points:

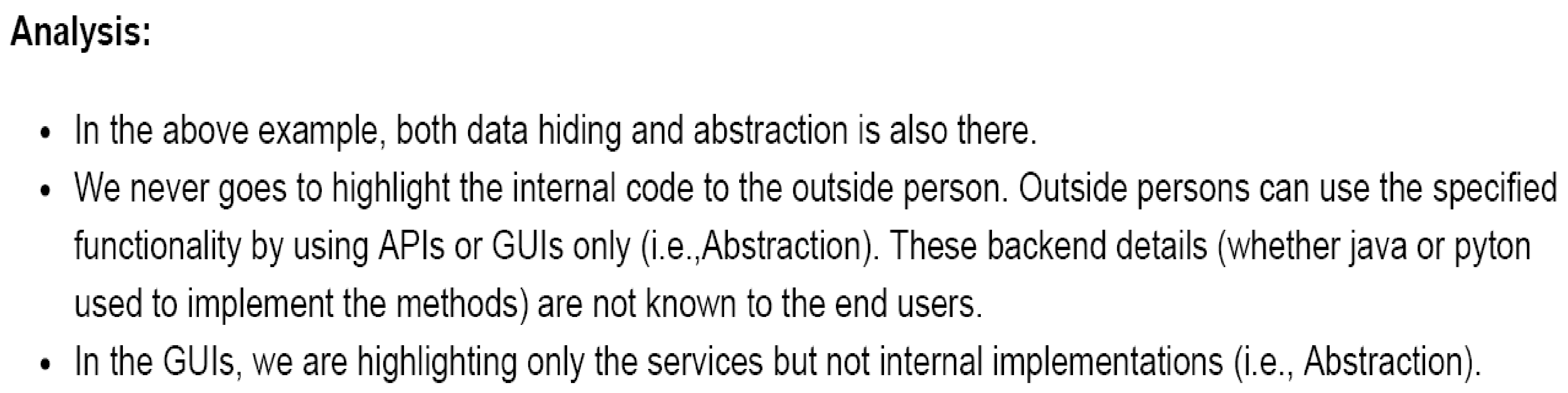
- The process of Binding/ Grouping/ encapsulating data and corresponding behavior(methods) into a single unit is nothing but encapsulation.
- Every Python class is an example of encapsulation.
- If any component follows data hiding and abstraction, such component is said to be encapsulated component.

Encapsulation = Data Hiding + Abstraction

Eg:



```
class Account:
    def __init__(self,initial_balance):
        self.__balance = initial_balance
    def getBalance(self):
        #Validation / Authentication
        return self.__balance
    def deposit(self,amount):
        #Validation / Authentication
        self.__balance = self.__balance + amount       # It is an Encapsulated Component
    def withdraw(self,amount):
        #Validation / Authentication
        self.__balance = self.__balance - amount
```



Analysis:

- In the above example, both data hiding and abstraction is also there.
- We never goes to highlight the internal code to the outside person. Outside persons can use the specified functionality by using APIs or GUIs only (i.e.,Abstraction). These backend details (whether java or pyton used to implement the methods) are not known to the end users.
- In the GUIs, we are highlighting only the services but not internal implementations (i.e., Abstraction).

Key Point:

- ❑ Hiding data behind methods is the central concept of encapsulation.

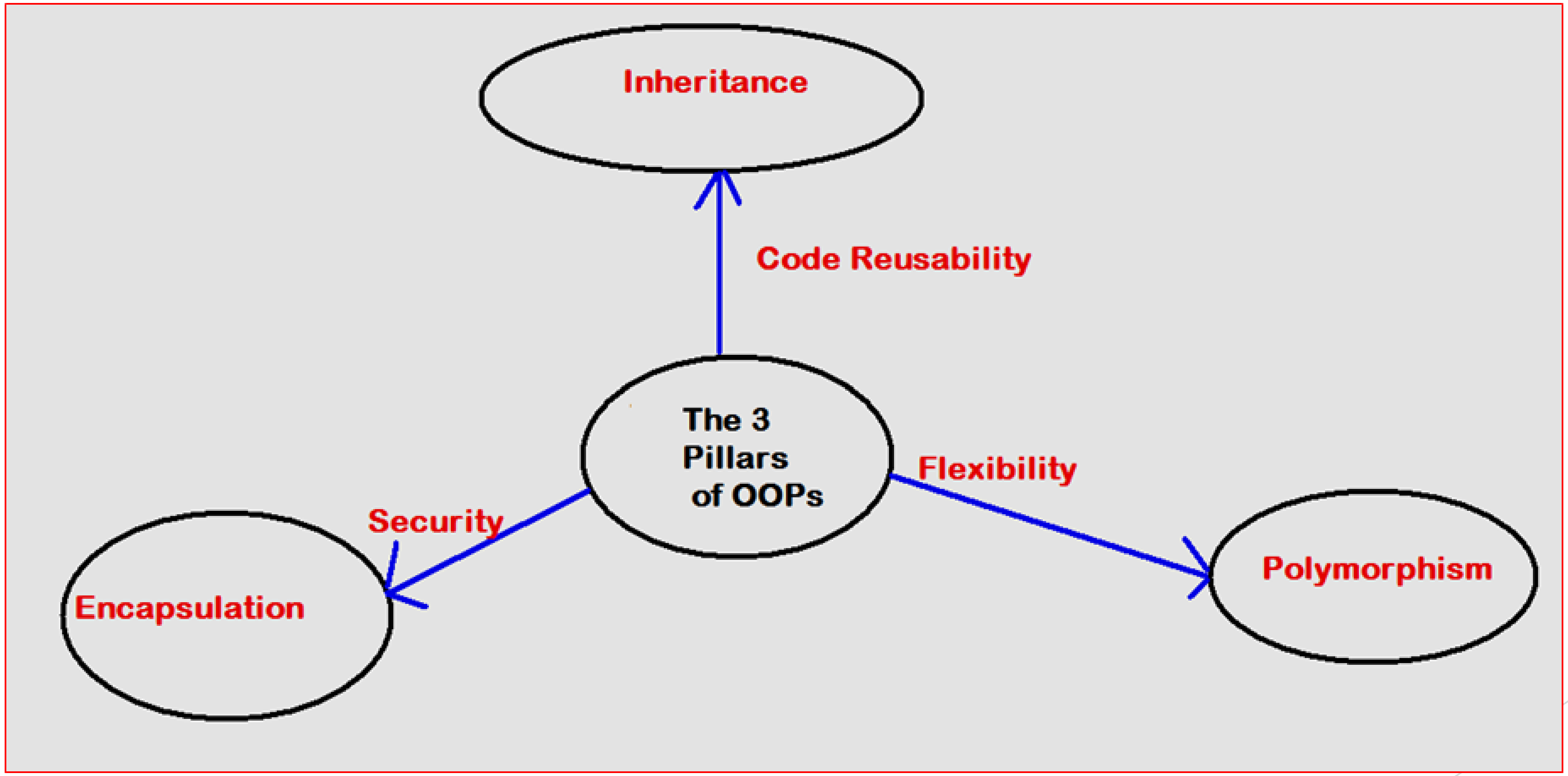
Advantages of Encapsulation:

1. Security.
2. Enhancement will become easy.
3. Maintainability and Modularity will be improved.

Points to Ponder about Encapsulation:

- The main advantage of encapsulation is Security.
- The main limitation of encapsulation is it increases length of the code and slows down execution. We should compromise with performance.
- If we want security we should compromise with Performance.
- If we want Performance we should compromise with Security.

The Three Pillars of Object Oriented Programming



Any question?

If you try to practice programs yourself, then you will learn many things automatically

Spend few minutes and then enjoy the study

Thank You