__repr__()

__repr__ is a special method in Python used to represent an object as a string. It is called by the built-in repr() function and is intended to provide an unambiguous representation of an object that can be used to recreate the object, if possible, using eval()

To provide a string representation of an object that is primarily intended for developers. It should be detailed and unambiguous, aiming to recreate the object if possible.

It is also instance method of object class.

Example:

```
class A:
    def __str__(self):
        return "inside str method"
    def __repr__(self):
        return "inside repr method"

obj1=A()
print(obj1)

class B:
    def __repr__(self):
        return "inside repr method"

obj2=B()
print(obj2)
```

Output

```
inside str method
inside repr method
```

Example

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
	Output
	100,200
	Point(x=100, y=200)

```
        return f'{self.x},{self.y}'
    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

point1=Point(100,200)
print(point1)
print(repr(point1))
```

Example:

```
class Integer:
    def __init__(self,n):
        self.__value=n

    def __str__(self):
        return f'{self.__value}'
    def __repr__(self):
        return f'Integer({self.__value})'

a=Integer(30)
b=str(a)
print(type(a),type(b))
c=repr(a)
print(c)
```

Output

```
<class '__main__.Integer'> <class 'str'>
Integer(30)
```

Operator Overloading

Existing operators does not work with user defined data types, it works with predefined data types. In order operator on user defined data type we use operator overloading.

Operator overloading in Python allows you to redefine the behavior of built-in operators like +, -, *, /, ==, <, >, etc., for objects of your custom classes. This is achieved by implementing special methods (also known as "dunder" methods or magic methods) in your class that correspond to the operators you want to overload.

Operator	**Magic Methods (Operator Methods)**
+	__add__
-	__sub__
*	__multiply__

/	__floatdiv__
//	__floordiv__
%	__mod__
>	__gt__
>=	__ge__
<	__lt__
<=	__le__
==	__eq__
!=	__nq__

Example:

```
class Employee:
    def __init__(self,eno,en,s):
        self.__empno=eno
        self.__ename=en
        self.__salary=s
    def __gt__(self,other):
        return self.__salary>other.__salary
    def __eq__(self,other):
        return self.__salary==other.__salary

emp1=Employee(101,"naresh",45000)
emp2=Employee(102,"suresh",54000)
if emp1>emp2:
    print("emp1 salary> emp2")
else:
    print("emp1 salary<emp2")

print(emp1==emp2)
emp3=Employee(103,"kishore",45000)
print(emp1==emp3)
```

Output

```
emp1 salary<emp2
False
True
```

Example:

```
class Complex:
    def __init__(self,real=0.0,imag=0.0):
        self.__real=real
        self.__imag=imag
    def __str__(self):
        return f'{self.__real},{self.__imag}'
    def __add__(self,other): # Operator Overloading
        r=Complex()
        r.__real=self.__real+other.__real
        r.__imag=self.__imag+other.__imag
        return r
c1=Complex(1.2,1.5)
print(c1)
c2=Complex(1.3,1.5)
print(c2)
c3=c1+c2
print(c3)
```

Output

```
1.2,1.5
1.3,1.5
2.5,3.0
```

Abstract classes and abstract methods (abc module)

Abstracts define set of rules and regulations

What is abstract class?

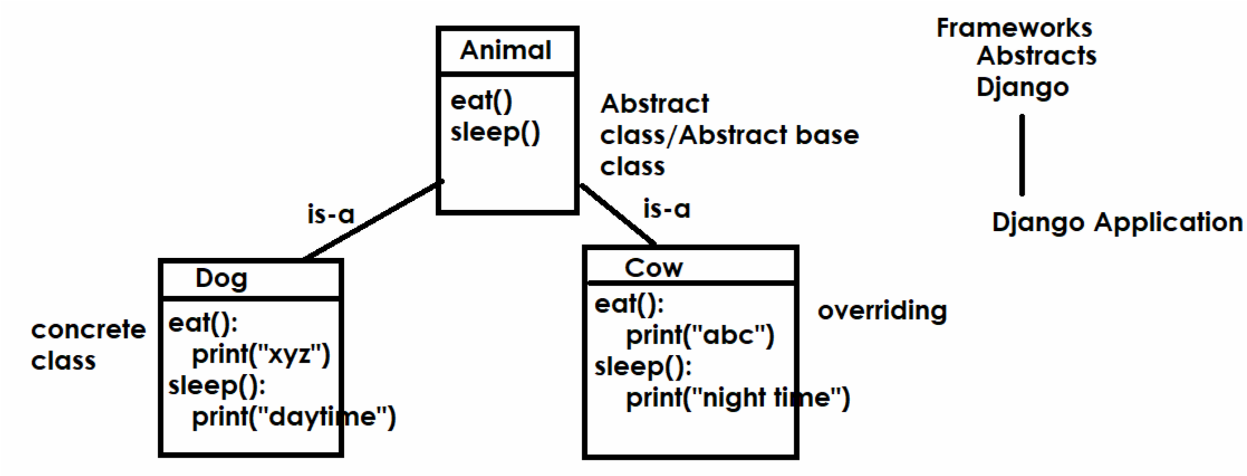
Abstract class defines set of rules and regulations to develop class.
Abstract class is an abstract data type (ADT). A data type which allows you builds similar data types.

What is abstract method?

An abstract method is method without implementation
Abstract method is an empty method.

Abstract method defines protocol or rules which has to be implemented by every derived class.
Abstract method must be override in derived class.

The class which inherits abstract base class and provides implementation of abstract methods is called concrete class.



To work with abstract classes and abstract method python provides a predefined module “abc”. “abc” stands for abstract base class module, this modules provides classes and methods to work with abstract classes and methods.

Syntax of abstract class
class class-name(abc.ABC):
abstract methods
non abstract methods/concrete methods

abstract base class must inherit ABC class of abc module.
Abstract class is inherited but we cannot used to create object.

Syntax of abstract methods
Abstract method must be defined inside abstract class.

```
class class-name(abc.ABC):
    @abc.abstractmethod
```

```
      def abstract-method-name(self,…):
         pass
```

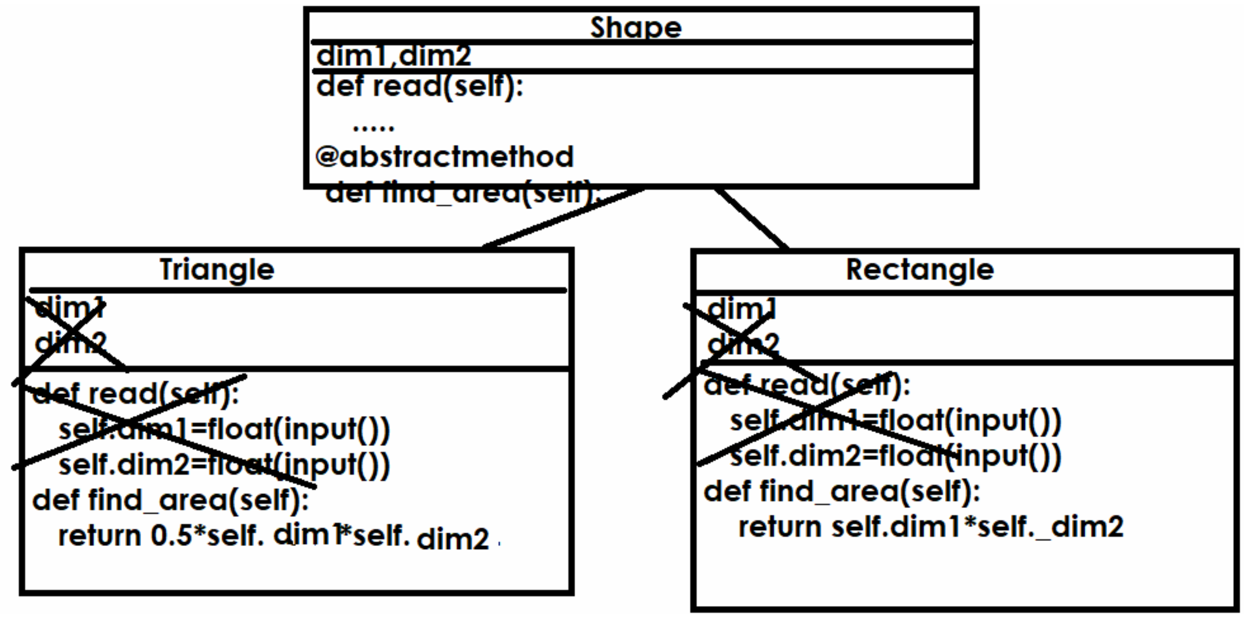
Example:

```
import abc
class Animal(abc.ABC): # abstract class
    @abc.abstractmethod
    def sleep(self):
        pass
class Dog(Animal): # concrete class
    def sleep(self): # Overriding
        print("Dog Sleep")
class Cow(Animal): # concrete class
    def sleep(self): # Overriding
        print("Cow Sleep")
class Cat(Animal):
    def eat(self):
        print("Cat Eat")
    def sleep(self):
        print("Cat Sleep")

#a1=Animal()
dog1=Dog()
dog1.sleep()
cow1=Cow()
cow1.sleep()
cat1=Cat()
cat1.sleep()
```

Output

```
Dog Sleep
Cow Sleep
Cat Sleep
```



When more than one sub class having same role with different implementation it is declared as abstract method.

Example:

```
import abc

class Shape(abc.ABC):
    def __init__(self):
        self._dim1=None
        self._dim2=None
    def read(self):
        self._dim1=float(input("Dim1 :"))
        self._dim2=float(input("Dim2 :"))
    @abc.abstractmethod
    def find_area(self):
        pass

class Triangle(Shape):
    def find_area(self):
        return self._dim1*self._dim2*0.5
class Rectangle(Shape):
    def find_area(self):
        return self._dim1*self._dim2
t1=Triangle()
r1=Rectangle()
t1.read()
```

```
area1=t1.find_area()
r1.read()
area2=r1.find_area()
print(area1)
print(area2)
```

Output

```
Dim1 :1.2
Dim2 :1.5
Dim1 :1.6
Dim2 :1.7
0.8999999999999999
2.72
```