Department of Computer Science and Engineering (UG Studies)
PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Object Oriented Programming (OOP) in Python

An integer is a set in mathematics. This set has whole numbers. 0 belongs to this set. If x belong to this set, so also x + 1 and x – 1.
On this set, we can do the following operations.

- Add
- Subtract
- Multiply

All these result in integers.

We divide two integers the result may not be integer.

A set like integer specifies what it contains and what we can do with them. Such a set in programming is called a type. A type specifies what it contains and what we can do with its elements.

What is the result of 25 + 36?
It is definitely 61. How do we add?
Table lookup? Add digits from right to left and propagate carry? Add row wise? Use tally marks? Use fingers and toes?
Set of integer which is a type says what the result of an operation is, but does not force how to add.

A pure type specifies 'what' and not 'how'.
'what' specifies an interface.
'how' specifies the implementation.

A language has few types. It cannot provide all possible types we may want to have – like desks, projector, chalkpiece. **A language should provide a mechanism to make our own type. That is called a class. A class is a type and implementation. A variable of the class type is called an object.**

This brief discussion stated above should be sufficient for our course.

We shall look at a few examples and add a few more points about objects as we go along.

Let us look at code from the file 0_intro.py.

```
class Ex0:
      pass
```

```
print(Ex0, type(Ex0))   #<class '__main__.Ex0'> <class 'type'>
```

So, Ex0 is a type in the package __main__.

Note: Class in OOPs allows you to package data and functionality together by *concept*, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine.

A class can have its attributes and behaviour.

```
# a class can have functions(behaviour)

class Ex1:
      def foo():
            print("foo of Ex")

Ex1.foo()
```

The function(behaviour) foo belongs to Ex1 and is invoked using the class name.

```
# a class can have attributes(fields or variables)

class Ex2:
      a = "test"

print(Ex2.a)
```

The attribute *a* with the value “test” belongs to Ex2 and is accessed using the class name.

We can create variables of Ex3 type, that is called as an instance or an object.

```
class Ex3:
      pass

a = Ex3()
print(a, type(a))
```

When an object is created, a special function of the class is called for initializing, that is called a constructor. The name of the constructor in Python is *__init__*.

```
class Ex4:
      def __init__():   # gives an error bcoz self parameter is missing
            print("constructor called")

a = Ex4()         # Ex4.__init__(a)
```

Output:

```
File "C:/Users/cop/Desktop/classdemo.py", line 12, in <module>
   a = Ex4()
TypeError: __init__() takes 0 positional arguments but 1 was given
```

The object creation statement *a = Ex4()* conceptually becomes *Ex4.__init__(a)*.
The class instance(i.e., object) is passed as the first argument to the constructor __init__.
Even though the argument(i.e., class instance or object) is implicit in Python implicit constructor call, we require an explicit appearance of object parameter in constructor. This can be given any name - is normally called self (like *this parameter* in java/C++).

```
class Ex4:
      def __init__(self):
            print("constructor called")
            print('self : ', self)

a = Ex4()
print('a : ', a)
```

Output:

```
constructor called
self :  <__main__.Ex4 object at 0x0000000007B96400>
a :  <__main__.Ex4 object at 0x0000000007B96400>
```

Observe that the both the outputs have the same value indicating ***self*** and ***a*** refer to the same object.

An object can have attributes. These are normally added and initialized in the constructor.

```
class Rect:
def __init__(self, l, b):
      self.length = l
      self.breadth = b
      # print(length) # error
      # all names should be fully qualifed
      print("__init__", self.length, self.breadth)

# initialize a Rect object r1 with length 20 and breadth 10
r1 = Rect(20, 10)
print(r1.length, r1.breadth)
```

Output:

```
__init__ 20 10
20 10
```

Also, observe that these attributes can be accessed anywhere with a fully qualified name. In some languages, we have access control (private, public …).
We do not have them in Python.

We can create any number of objects - we have to invoke the constructor to initialize them.

```
r1 = Rect(20, 10)
r2 = Rect(40, 30)
r3 = Rect(60, 40)
```

In the below code, all instances data members are initialized with the same values 20 and 10.

```
class Rect:
```

```
    def __init__(self):
        self.length = 20
        self.breadth = 10
        print("__init__", self.length, self.breadth)
r1 = Rect()
print(r1.length, r1.breadth)
r2 = Rect()
print(r2.length, r2.breadth)
```

Output:

```
__init__ 20 10
20 10
__init__ 20 10
20 10
```

There is no constructor overloading in Python.

```
class Rect:
    def __init__(self, l, b):
        self.length = l
        self.breadth = b
        print('First Constructor is used')
    def __init__(self, l, b):
        self.length = l
        self.breadth = b
        print('Second Constructor is used')

r1 = Rect(10,20)
print('l =',r1.length)
print('b =',r1.breadth)
```

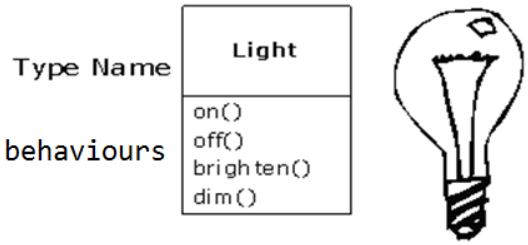
Output:

```
Second Constructor is used
l = 10
b = 20
```

What this means is Python rebinds the name __init__ to the new **method**. This means the first declaration of this method is inaccessible now.

Integers support functions like __add__. Strings support functions like upper. They are called behaviours of that particular type.

Our class can also have behaviour through functions in the class.



In the above code, an object of class Rect contains length and breadth. Given the rectangle, we can extract the length and the breadth to find the area. So, the function area is invoked with an object and does not require any other parameter.

The object encapsulates both attributes and behaviour. **Encapsulation is putting together data and functions (attributes and behaviour).**

```
class Rect:
      def __init__(self, l, b):
            self.length = l
            self.breadth = b

      def area(self):
            return self.length * self.breadth

# initialize a Rect object r1 with length 20 and breadth 10
r1 = Rect(20, 10)
r2 = Rect(40, 30)
print('area of r1 : ', r1.area())   # Rect.area(r1)
print('area of r2 : ', r2.area())   # Rect.area(r2)
```

Output:

```
area of r1 :  200
area of r2 :  1200
```

Demo: Pythontutor.com

Let us look at a few more behaviours.

The change_length method requires the new length apart from the object attributes length and breadth. We require one explicit argument while calling change_length method and two explicit parameters in defining change_length method. The first parameter always refers to the object through which the call is made.

```
# Rect.change_length, Rect.change_breadth
class Rect:
      def __init__(self, l, b):
            self.length = l
            self.breadth = b

      def area(self):
            return self.length * self.breadth

      def change_length(self, l):
            self.length = l

      def change_breadth(self, b):
            self.breadth = b

      def disp(self):
            print('length : ', self.length)
            print('breadth : ', self.breadth)

r1 = Rect(20, 10)       #Rect.__init__(r1,20,10)

print("before change length ")
```

```
r1.disp()               #Rect.disp(r1)
r1.change_length(40)    #Rect.change_length(r1,40)
print("after change length ")
r1.disp()

print("before change breadth ")
r1.disp()

r1.change_breadth(30)
print("after change breadth ")
r1.disp()
```

Output:

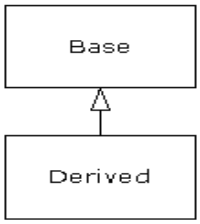
```
before change length
length  :  20
breadth :  10
after change length
length  :  40
breadth :  10
before change breadth
length  :  40
breadth :  10
after change breadth
length  :  40
breadth :  30
```

Inheritance

Inheritance is a mechanism wherein a new class is derived from an existing class. In Python, classes may inherit or acquire the **attributes** (data) and **behaviors** (functions) of other classes.
A class derived from another class is called a **subclass**, whereas the class from which a subclass is derived is called a **superclass**.
Inheritance is the process wherein characteristics are inherited from ancestors. Similarly, in Python, a subclass inherits the **attributes** and **behaviors** of its superclass (ancestor).

For example, a vehicle is a superclass and a car is a subclass. The car (subclass) inherits all of the vehicle's properties. **The inheritance mechanism is very useful in code reuse**.

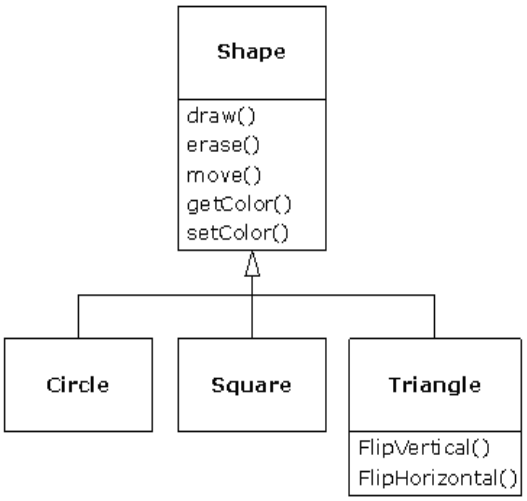
Inheritance is a mechanism to capture the commonality between classes. This is used to create class or type hierarchy. This indicates the relation of "ISA" between classes.



A base type contains all of the attributes and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that this core can be realized.

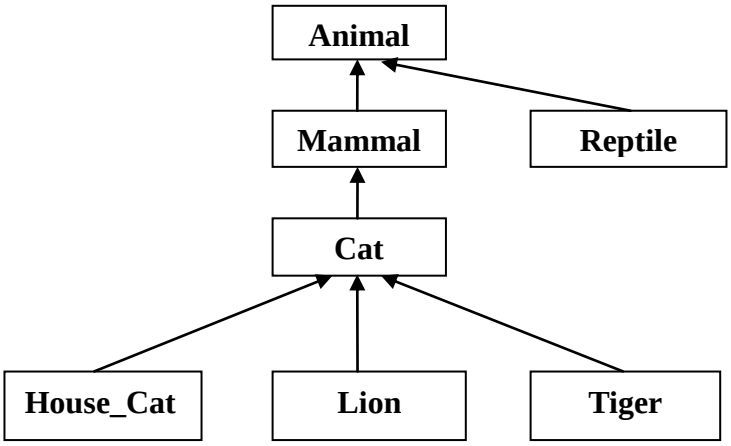
It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. **It's nicer if we can take the existing class, clone it, and then make additions and modifications to the clone. This is effectively what you get with *inheritance*.**

Example 1:



The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc.
From this, specific types of shapes are derived (inherited)—circle, square, triangle, and so on—each of which may have additional characteristics and behaviors.

Example 2:



Now, based on the above example, In Object Oriented terms, the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Cat is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

- Mammal **IS-A** Animal
- Reptile **IS-A** Animal
- Cat **IS-A** Mammal
- Hence : Cat **IS-A** Animal as well

A cat is a mammal. Tiger is a cat. House cat is a cat. Lion is a cat. All these have common characteristics. We can address all of them as cats. They walk on their toes. They climb trees. There could be differences in the way they walk or they climb. **Because of common characteristics, we can consider a tiger or a lion as a cat**. This helps in maintenance of programs.
We call this **"is a" relationship** between classes as inheritance.

Example 3:

```
class P2D:
      def __init__(self, x, y):
            self.x = x
            self.y = y
      def disp(self):
            print ("x : ", self.x)
            print ("y : ", self.y)

class P3D(P2D):      # P3D inherits from P2D. An object of P3D gets everything P2D has.
      def __init__(self, x, y, z):
            P2D.__init__(self, x, y)
            self.z = z

p3 = P3D(11, 12, 13)
p3.disp()     # no function in the derived class; calls the base class function by default
print(isinstance(p3, P2D))
```

Output:

```
x :  11
y :  12
True
```

A point in three dimensions(class P3D) is also a point in two dimensions(P2D).

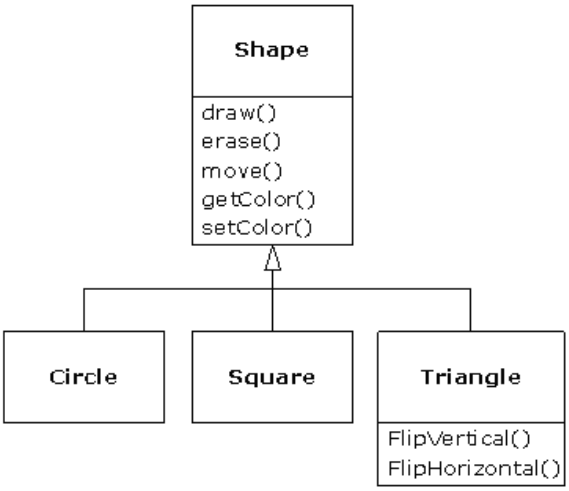
`class P3D(P2D):`

The above statement indicates that relationship. We say that P3D is a derived class and P2D is the base class.
An object of derived class with a reference to an embedded object of base class – normally referred to as the base class sub object.
When we create an object of the derived class, we invoke the constructor of the derived class which in turn calls the base class constructor on the base class sub object.

The base class provides a default implementation of the behaviour for the derived class.
`p3.disp() # no function in the derived class; calls the base class function by default`
As there is no function called disp in the derived class, the base class function gets called.

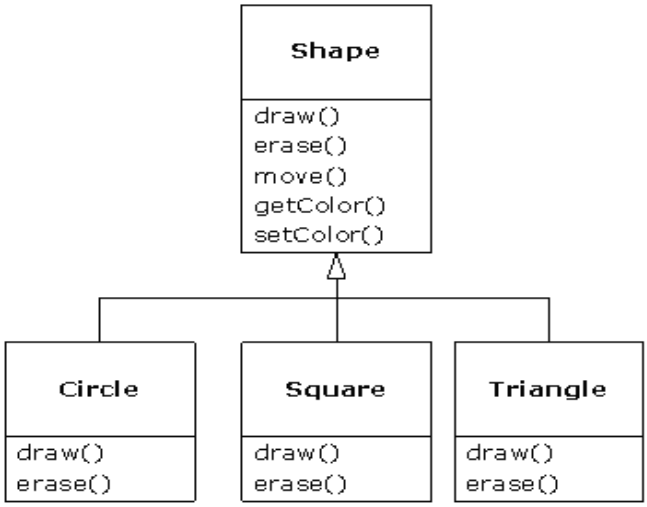
You have **two ways to differentiate your new derived class from the original base class**.

The first is quite straightforward: You simply add brand new functions to the derived class. These new functions are not part of the base-class interface.



Overriding:

The second and more important way to differentiate your new class is to *change* the behavior of an existing base-class function. This is referred to as ***overriding*** that function.



To override a function, you simply create a new definition for the function in the derived class. You're saying, " I'm using the same function interface(or signature) here, but I want it to do something different for my new type."

Example:

```
class P2D:
      def __init__(self, x, y):
            self.x = x
            self.y = y

      def disp(self):
            print ("x : ", self.x)
            print ("y : ", self.y)

class P3D(P2D) :
      def __init__(self, x, y, z):
            P2D.__init__(self, x, y)
            self.z = z
```

```
        #overriding the base class function
        def disp(self):
              P2D.disp(self) # delegate work to the base class
              print("z : ", self.z)

p3 = P3D(11, 12, 13)
p3.disp()
```

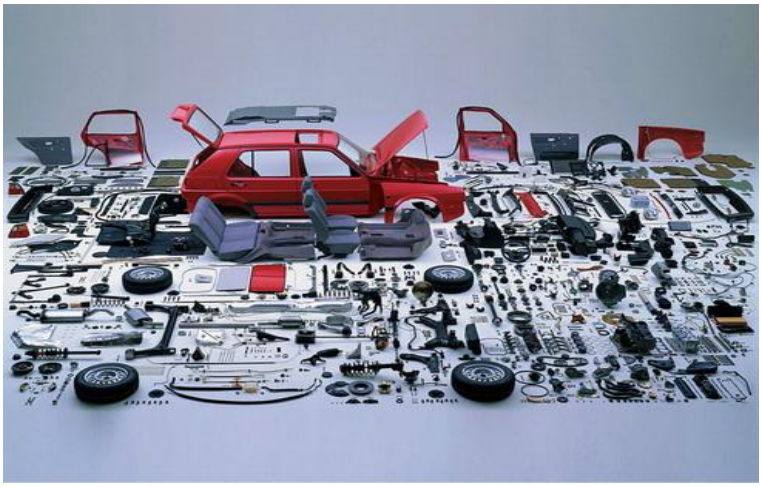
Output:

```
x :  11
y :  12
z :  13
```

It is possible for the derived class to provide its own function disp. This concept is called overriding. The derived class modifies the function of the base class. In many such cases, this overriding function may in turn call the base class function.

Composition:

Composing a new class from existing classes is called **composition**. Composition is often referred to as a **"has-a"** relationship, as in "A car has an engine."



Example 1: A cat has paws, tail, whiskers, claws, teeth, … .

Example 2:

In the below code, MyEvent object has MyDate object as part of it. An Event occurs on a date. An event is not a date. A date is not an event. **An event has a date as part of it**. So observe that the constructor of MyEvent class initializes a field called date by creating and initializing an object of MyDate class.

All functions of the MyEvent class in turn invoke the functions of MyDate class through the attribute date – this is called delegation or forwarding.

We have been using the function str on different types like int. When we invoke this function, a special function __str__ of that class will be called. We can also provide such a function in our class to convert an object of our class to a string. In the print context, this function gets called automatically.

```
# filename : 2_composition.py

class MyDate:
      def __init__(self, dd, mm, yy):
            self.dd = dd
            self.mm = mm
            self.yy = yy

      def __str__(self):
            return str(self.dd) + "-" + str(self.mm) + "-" + str(self.yy)

      def key(self):
            return self.yy * 365 + self.mm * 30 + self.dd

d = MyDate(15, 8, 1947)
print(d)
print(d.key())
```

Output:

```
15-8-1947
710910
```

```
class MyEvent:
      def __init__(self, dd, mm, yy, detail):
            self.date = MyDate(dd, mm, yy)
            self.detail = detail

      def __str__(self):
            return str(self.date) + " => " + self.detail

      def key(self):
            #return self.detail
            return self.date.key()  # return MyDate.key(self.date)

e = MyEvent(15, 8, 1947, "Independence Day")
print(e)

print(e.key()) # print(MyEvent.key(e))

print(MyEvent.key(MyEvent(15, 8, 1947, "Independence Day")))
```

Output:

```
15-8-1947 => Independence Day
710910
710910
```

The rest of the code creates a list of user defined objects – creates a list of events. Then the list is sorted and displayed using for statement.

We would like sort these dates based on the date. We provide keyword parameter key which converts the date into a single number.

The callback MyEvent.key in turn delegates the work to MyDate.key. This function converts a given date to a single integer by using some approximate formula to count the number of days from the beginning of the era.

The sorted function converts each event into this number and sorts the events based on this number.

```
mylist = [
            MyEvent(26, 1, 1956, "Republic Day"),
```

```
            MyEvent(1, 11, 1973, "Karnataka born"),
            MyEvent(2, 10, 1868, "Gandhi jayanthi"),
            MyEvent(15, 8, 1947, "Independence Day"),
            MyEvent(16, 12, 1971, "Amar sonar bangla")
        ]

for e in sorted(mylist, key = MyEvent.key):
print(e)
```

Output:

```
2-10-1868 => Gandhi jayanthi
15-8-1947 => Independence Day
26-1-1956 => Republic Day
16-12-1971 => Amar sonar bangla
1-11-1973 => Karnataka born
```

Class or Static Variables in Python

Class or static variables are shared by all objects. Instance or non-static variables are different for different objects (every object has a copy of it).

In python, all variables which are assigned a value in class declaration are class variables and variables which are assigned values inside class methods are instance variables.

Example 1: Class for Computer Science Student

```
class CSStudent:

    branch = 'CSE'    # Class Variable (static variable)

    def __init__(self,name,roll):
        self.name = name            # Instance Variable
        self.roll = roll            # Instance Variable

# Objects of CSStudent class
s1 = CSStudent('Ram', 11)
s2 = CSStudent('Kishan', 21)

print('Branch =',s1.branch)
print('Name =',s1.name)
print('Roll No. =',s1.roll)
print()
print('Branch =',s2.branch)
print('Name =',s2.name)
print('Roll No. =',s2.roll)

# Class variables can be accessed using class name also
print('Branch name accessed using class name:',CSStudent.branch)
```

Output:

```
Branch = CSE
Name = Ram
Roll No. = 11

Branch = CSE
```

```
Name = Kishan
Roll No. = 21
Branch name accessed using class name: CSE
```

Destructors in Python:

The `__del__()` method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e **when the reference count becomes zero.**

We know that the constructor is always called when an object is created. Similarly when an object dies, a destructor function will be called.

The destructor is called

- when an object dies or
- when the object type is changed or
- when *del* is called on the object explicitly.

But remember that objects are also reference counted like any other entity in Python. The destructor gets called only when the reference count becomes zero.

Example 2:

The number of students in a class is not an attribute of any one particular student. It is an attribute of the class itself. In such cases, **we create attributes outside of the functions within the class. These are called as static fields or variables of the class**.
We can access static variable using the classname or **an object name as long as the object does not have a field by the same name.**

We may want to invoke a function to display or play with a static attribute. Such a function should be callable using the class name or an object without passing object as an implicit parameter. So, the function should have no parameters. We achieve by using the following magic.

```
@staticmethod           #@staticmethod decorator to flag it as a static method.
def disp_count():       # there is no parameter like self
      print("disp_count : count : ", MyDate.date_count)
```

```
# filename: 3_static.py

# count the number of objects
class MyDate:
      date_count = 0   # class attribute
      def __init__(self, dd, mm, yy):
            self.dd = dd
            self.mm = mm
            self.yy = yy
            MyDate.date_count += 1
      def __str__(self):
            return str(self.dd) + "-" + str(self.mm) + "-" + str(self.yy)
      def __del__(self):
```

```
            MyDate.date_count -= 1

        """@staticmethod notation is used to indicate that the function should be changed
        from an objectfunction to a static function - this does not have any parameter – no
        self. """
        @staticmethod
        def disp_count(): # A static method does not receive an implicit first argument.
            print("disp_count : count : ", MyDate.date_count)

"""Counting dates attribute date_count is a property of the class and not any date object,
such an attribute is called a class attribute. """

print("count : ", MyDate.date_count)
d1 = MyDate(11, 11, 11)
print("count : ", MyDate.date_count)
d2 = MyDate(12, 12, 12)
print("count : ", MyDate.date_count)

# should decrease when the object is destroyed.
# a special function called destructor __del__ will be called
# we can take care of resources in this function
del d2

print("Object count info accessed using classname Mydate:")
print("count : ", MyDate.date_count)
"""We could have a function to display static attributes called static function; there do
not depend on any object of the class. This is a class behaviour and not object behavior
"""
MyDate.disp_count()

print("Object count info accessed using objectname d1:")
print("count : ", d1.date_count)
d1.disp_count()
```

Output:

```
count :  0
count :  1
count :  2
Object count info accessed using classname Mydate:
count :  1
disp_count : count :  1
Object count info accessed using objectname d1:
count :  1
disp_count : count :  1
```

Python property() function

Traditional object-oriented languages like Java and C# use properties in a class to encapsulate data. **Property includes the getter and setter method to access encapsulated data.** A class in Python can also include properties by using the property() function.

In Python, the main purpose of **property()** function is to create property of a class.

Syntax: property(getter, setter, deleter, docstring)

Parameters:

getter() – used to get the value of attribute
setter() – used to set the value of attribute
deleter() – used to delete the attribute value
docstring – string that contains the documentation (docstring) for the attribute

Return: Returns a property attribute from the given getter, setter and deleter.

Note:

- If no arguments are given, **property()** method returns a base property attribute that doesn't contain any getter, setter or deleter.
- If doc isn't provided, property() method takes the docstring of the getter function.

Example 1:

```
class Alphabet:
    def __init__(self, value):
        self._value = value     # _value is a private instance variable

    # getting the values
    def getValue(self):
        print('Getting value')
        return self._value

    # setting the values
    def setValue(self, value):
        print('Setting value to ' + value)
        self._value = value

    # deleting the values
    def delValue(self):
        print('Deleting value')
        del self._value

    # property method returns a property attribute from the given getter, setter and deleter
    value = property(getValue, setValue, delValue, )

d1 = Alphabet('PES University')
print(d1.value)    # print(d1.getValue())
d1.value = 'PESU'  # d1.setValue('PESU')
del d1.value       # print(d1.getValue())
```

Output:

```
Getting value
```

```
PES University
Setting value to PESU
Deleting value
```

Example 2:

A rectangle has length and breadth. We can compute the area given an object of rectangle. The user may want to look upon area as an attribute. We can support it using the concept called **property**.

d1 is an object of MyDate class.

The functions get_dd gets us the date of MyDate object and set_dd changes the date of MyDate object.

We create a property called DD and associate in our example with a pair of functions. We can use expression DD of MyDate object either to the right of assignment or to the left of assignment. These are called rvalue and lvalue usages of that field.

When we use the expression d1.DD as an rvalue, the first function in the pair is called. So `d1.DD` becomes `d1.get_dd()`.

`d1.DD = 22`

When we use the expression d1.DD as a lvalue, the second function in the pair is called. So this statement above becomes `d1.set_dd(22)`.

A property allows the user to consider an entity as an attributes of a class. This is the way we support the user's view.

```
# get and set
class MyDate:
      date_count = 0
      def __init__(self, dd, mm, yy):
            self.dd = dd
            self.mm = mm
            self.yy = yy

      def get_dd(self):
            return self.dd

      def set_dd(self, dd) :
            self.dd = dd

      DD = property(get_dd, set_dd)

d1 = MyDate(11, 11, 11)
print(d1.DD)      # print(d1.get_dd())
d1.DD = 22        # d1.set_dd(22)
print(d1.DD)      # print(d1.get_dd())
```

Output:

```
11
22
```

This is a brief introduction to Object Oriented Programming in Python.

References:

1. oo.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://docs.python.org/
3. https://realpython.com/instance-class-and-static-methods-demystified/