ECE-122

Chapter-3: **Object Oriented Programming and Applications**

3.6 Encapsulation and Properties

Class Anatomy: Review

- Attributes
  - class attributes
  - instance attributes
  - instance attributes can be made public, private (__), protected (_)

```
class MyClass:
    #constructor
    def __init__(self):
        self.a="I am public"
        self._b="I am protected"
        self.__c="I am private"

obj=MyClass()
print(obj.a)           # public - accessible
print(obj._b)          # protected – accessible, use at your own risk
print(obj.__c)         # private- not accessible >>Python error <<
```

Class Anatomy: Review

- Methods
  - constructor __init__
  - magic methods (so far): __str__, __del__
  - instance methods
  - static methods
  - methods can also be made public, private (__), protected (_)

```
class Rectangle:
    #constructor
    def __init__(self,width=0,height=0):
        self.w=width     #w public
        self.h=height    #h  public

    #Instance methods
    def  __compute_area(self):    # private
         return self.w*self.h
    def  compute_area_x2(self):  # public
         return self.__compute_area()*2
```

```
from Rectangle import Rectangle

box1=Rectangle(10,10)


print(box1.compute_area_x2())
print(box1.__compute_area()) # error
```

Encapsulation in OOP

- Encapsulation consists of using getter and setter methods to access instance attributes. Example:

line1.length=100	becomes	line1.set_length(100)
print(line1.length)	becomes	print(line1.get_length())

- Set and get methods could do more than just “setting” and “getting” values. You could explicitly *encapsulate* some statement within these methods. Example: print some info, read/write a file, send an email, etc.
- More importantly, the setter methods can be used to set some properties for the attributes. Example: restrict to a range of possible values, etc.

```
#set method for length
def  set_length(self,length):
     if length>50:
        length=50
     self.__length=length
```

Data Abstraction

- Some data must be "hidden", so that it can't be accidentally changed
- **Data encapsulation** is often associated with the use of private instance attributes (**information hiding**), we talk then about **data abstraction**
- Complete example.

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.length=length
```

→

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.set_length(length)

    #getter-setter methods
    def  set_length(self,length):
         if length>50:
            length=50
         self.__length=length
    def  get_length(self):
         return self.__length
```

Encapsulation-the python way

- In OOP, it is customary to make all the instance attributes private and to access them via setter-getter methods.
- This is the Java way that works fine in Python
- One drawback is that the main code (interface) becomes cumbersome

```
from Line import Line

l1,l2,l3=Line(),Line(),Line()

l1.length=30
l2.length=20
l3.length=l1.length+l2.length
```

$\Rightarrow$

```
from Line import Line

l1,l2,l3=Line(),Line(),Line()

l1.set_length(30)
l2.set_length(20)
l3.set_length(l1.get_length()+l2.get_length())
```

- The Python(ic) way: keep things simple.
- <u>It is possible to achieve encapsulation without data abstraction!</u>
  (stated otherwise here: l1.length=60 should assign 50 to attribute length)

Encapsulation-the python way

- 1st approach:
  - by adding the function **property** inside the class. Example for attribute **name**=property(**get_name**,**set_name**)

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.set_length(length)
        # it is possible to use instead
        self.length=length
    #getter-setter methods
    def  set_length(self,length):
          if length>50:
            length=50
          self.__length=length
    def  get_length(self):
          return self.__length

    length=property(get_length,set_length)
```

*With this line, you will be able to directly access the attribute (example: l1.length) but you are actually calling the getter/setter method.

*The instance attribute is still private (__length is private), but you define a new attribute length which is now public.

*There is no data abstraction here (no need to call explicitly get and set methods)

*There is still encapsulation since the get and set methods are called implicitly

Encapsulation-the python way

- 2nd approach (python syntax—far from traditional OOP syntax):
  - using the decorators: @**property** for get and @**name.setter** for set

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.length=length # direct assignment

    @property
    def  length(self):
         return self.__length

    @length.setter
    def  length(self,length):
         if length>50:
            length=50
         self.__length=length
```

*two new functions with the same names (different number of arguments), and with two different decorators

*the direct assignment in the __init__ method will call the setter decorator method

Encapsulation-summary

```
from Line import Line
l1,l2,l3=Line(),Line(),Line()

l1.length,l2.length=60,20
l3.length=l1.length+l2.length
print(l1.length,l2.length,l3.length)
```

No Encapsulation

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.length=length
```

```
60 20 80
```

Encapsulation 1st approach

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.set_length(length)

    #getter-setter methods
    def  set_length(self,length):
        if length>50:
            length=50
        self.__length=length
    def  get_length(self):
        return self.__length

    length=property(get_length,set_length)
```

```
50 20 50
```

Encapsulation 2nd approach

```
class Line:
    #constructor
    def __init__(self,length=0):
        self.length=length

    @property
    def  length(self):
        return self.__length
    @length.setter
    def  length(self,length):
        if length>50:
            length=50
        self.__length=length
```

@property

- The property decorator can be used in a more general way than a simple get method-like property for a given attribute. A property can be deduced from the values of more than one attribute. Example:

```
class Line:
    #constructor
    def __init__(self,length=0,shape="continuous"):
        self.length=length
        self.__shape=shape
    @property
    def  length(self):
          return self.__length
    @length.setter
    def  length(self,length):
          if length>50: length=50
          self.__length=length
    @property
    def  condition(self):
          if self.__shape=="continuous" and self.length>10:
               return "this is a regular line"
          else:
               return "this is a not a regular line"
```

```
from Line import Line
l1=Line(60,"dotted")
l2=Line(1,"continuous")
l3=Line(30,"continuous")

print("l1:",l1.condition)
print("l2:",l2.condition)
print("l3:",l3.condition)
```

```
l1: this is a not a regular line
l2: this is a not a regular line
l3: this is a regular line
```