Python Programming

by Narendra Allam

Chapter 10

Object Orientation

Topics Covering

- Class
- Abstraction
- Encapsulation
  - Data hiding
  - Data binding
- Accessing data members and member functions explicitely
- Passing paramets to **init**()
- Implementing **repr**(),**eval**()
- Adding a property at run-time
- Inheritence
  - delegating functionality to parent constructor,init
  - Diamond problem
  - MRO
- Using abc module
- Private Memebrs
- Creating inline objects, classes, types
- Static variables, Static Methods and Class Methods
- Funcion Objects (Functor), Callable objects
- Decorator and Context manager
- polymorphism
- Function Overloading
- Operator Overloading
- Sorting Objects

A long time ago when there was no object orientation

With the python concepts, we learned so far (including files and modules), no doubt! we can handle a complete python project. Lets imagine our software development career,...

Mr.Alex, who owns a bank ABX, is our client now. And good news is that, we were choosen to develop a software solution for his bank. Initially he has given two requirements. Each requirement is a banking functionality. We are going to implement them now.

1. Personal Banking
2. Personal Loans

We spent few weeks and completed the application, and ended up with 100 functions and 40 global variables(may contains lists, dictionaries).

We wrote all the code in a single file named 'banking_system.py' using functions. This is procedural style of programming.

There are few limitations to procedural style.

1. Spaghetti code:

Spaghetti code is source code that has a complex and tangled control structure, especially one using many module imports with scattered functionalites across multiple files. It is named such because program flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled. Spaghetti code can be caused by several factors, such as continuous modifications by several people with different programming styles over a long project life cycle.

As developers have freedom to write code any where in the code base, One functinality possibily get scattered among multiple files, which is very difficult to understand for a new programmer and makes scalability almost impossible to achieve.

2. Security - Accidental changes.

It is very hard to maintain code in a single file for entire project which is surely not recommended. Multiple developers whould be implementing multiple functionalities. There will be confilicts, if two developers are simultaniously modifying same code. Developers should sit together and spend hours to resolve the conflicts. seperation of functionalities into multiple modules/files might help to prevent changes, which reduces the possibility of working two developers on same file/module. Cool, lets try that,

1. banking_system.py which contains all personal banking related functions and variables (80 funs and 30 vars)
2. personal_lans.py which contains all personal loans related functions and variables(20 funs and 10 vars)

Still data is open to all developers, we cannot prevent accessing 'Personal Loans' data from 'Personal Banking', because developer can easiy import data and change which leads to unpredictable control flow and hard to debug.

We need stricter boundaries to prevent unwanted changes. We need stricter boundaries to group up all the code related to a functionality at one place. We need stricter boundaries for scalability.

3. Scalability - Replication for Reusability

After few months Mr.Alex decided and came with an aggressive marketing strategy and we came to know that he was going to start 100 branches of ABX bank, exclusively for personal loans.

We were expected to make changes to scale 'Personal Loans' functionality. Now we are going to maintain 100 more units of personal loans functionality. Each unit should maintain its own data set of but functions(actions) are same. How do we achieve this ?

Do we have to create 100 'personal_loans.py' files? or just one file with 100 sets of personal loan variables?

In future, he wants to add few more functionalites like car loans, home loans to the exisiting software system can we make reuse of exisiting code ? a lot of questions in mind!

We started with,

100 funcs and 40 vars (funcs - functions, vars - varaibles)

we seperated them as,

80 funcs + 30 vars - Personal banking 20 funcs + 10 vars - Personal loans

now, we want 100 units of personal loans

20 funcs + 100 * (10 vars for each branch)

Note: Functions are common, only required is, a set of 10 vars for each branch.

We should find an easy way to scale this. Yes there is a way - 'type'

'typing' - Creating a type in programming languages is a powerful technique.

'dict' is a type in python. It is a complex data structure in fact. But creating hundreds of dicts is trouble-free.

d = dict(), here d is a unit of dict functionality. We know that we can create thousands of dicts using this simple dict() function. What is making this possible. Some python developer classified all dicionary functionalities into a type and named it as 'dict'.

That means, if we create 'PersonlLoans' as a type, creating thousands of units is effort less.

Object orientation solves all the above .

1. Spaghetti code - Object oriented programming is structured programming, very less scope for tangled code
2. Preventing accidental changes - Encapsulation decides what to hide and what to expose
3. Scalability - Class is a type, we can create multiple units of same functionality by instantiation

Thinking in object orientation:

1. We found a relation between funcs and vars for Personal Loan functionality and we modularized them, which is called - **data binding**
2. lets bind these 20 funcs and 10 vars and isolate(hide) inside a container - **data hiding**
3. The container is - **class**
4. We should not restrict everything inside the container, as funcs are social, they should interact with external funcs. Lets expose few funcs to interact with external functionalities - **abstraction**
5. Whe should have a protocol to control data hiding and abstraction. We should care fully think about, what needs to be hidden? what needs to be exposed to the external components? and draw a boundary in between - **encapulation**
6. How do we resuse existing code? - **inheritance**
7. How do we incorporate new changes into a complex project? - **overriding**, **overloading** which is **polymorphism**

Object orientation is all about - in-advance planning of a project design by anticipating future changes

Class

- Class is a model of any real-world entity, process or an idea.
- A class is an extensible program-code-template for reusablity.
- Class contains data (member variables) and actions(member functions or methods)
- Class is a blue-print of structure and behaviour, more importantly a class is a 'type', so that, we can create mutiple copies (instances) of the same structure and behaviour.
- class instances or called objects.

- object is the physical existance of a class

Syntax:

```
class ClassName(object):
    """
    All attributes are mostly written in side __init__ method
    """

    def __init__(self, args, ...):
        self.attribute1 = some_val
        self.attribute2 = some_val
        self.attribute3 = some_val

    def method1(self, args, ...):
        # code
    def method2(self, args, ...):
        # code
```

Upgrading Personal Loans sytem with Object Orientation ...

```
# personal_loans.py
# -------------------

class PersonalLoans(object):
        # HIDDEN DATA
        def __init__(self):
            self.__cusomerDetails = []
            self.__loanTypes = []
        ...

        # HIDDEN FUNCTIONS
        def __utility1(self):
            ...
        def __utility1(self):
            ...

        # PUBLIC FUNCTIONS/INTERFACES
        def get_customer_details():
            ...
        def get_loan_details():
            ...
```

Abstraction:

Hiding Complex details, providing simple interface.
Abstractions allow us to think of complex things in a simpler way.
e.g., a Car is an abstraction of details such as a Chassis, Motor, Wheels, etc.

Encapsulation:

Encapsulation is how we decide the level of detail of the elements comprising our abstractions.
Good encapsulation applies information hiding, to enforce limits of details.

Data hiding:

Limiting access to details of an implementation(Data or functions).

Data binding:

Establishing a connection between data and the functions which depend and makes use of that data is called Data binding.
Note: In functional style of programming there is no relation between data and functions, because funtions don't depend on data.

Inheritance:

It is a technique of reusing code, by extending or modifying the existing code.

Polymorphism:

Single interface multiple functionalities.
(or) Polymorphism is the ability of doing different things by using the same name.
(or) Plymorphism is conditional and contextual execution of a functionality.

Modeling an employee

In [113]:

```
class Employee(object):
    def __init__(self):
        self.num = 0
        self.name = ''
        self.salary = 0.0

    def get_salary(self):
        return self.salary

    def get_name(self):
        return self.name

    def print_employee(self):
        print ('num=', self.num, ' name=', self.name, ' sal=', self.salary)
```

Creating an object for class **Employee**

Note: Object creation is also called **instantiation**

In [114]:

```
e1 = Employee() # Employee.__new__().__init__()
```

In [116]:

```
e2 = Employee()
```

here e1 and e2 are objects or instances

init_()

*_init_*() is a builtin function for a class, which is called for each object at the time of object creation. *_init_*() is used for iniitializing an object with data members

Use '.' operator top access properties of a class

In [118]:

```
1 e1.salary
```

Out[118]:

```
0.0
```

In [119]:

```
1 e1.get_salary()
```

Out[119]:

```
0.0
```

In [120]:

```
1 print (e1.num, e1.name, e1.salary)
```

```
0  0.0
```

Accessing data members and member functions explicitely

In [121]:

```
1 e1.num = 1234
2 e1.name = 'John'
3 e1.salary = 23000
4
5 print (e1.num, e1.name, e1.salary)
```

```
1234 John 23000
```

In [122]:

```
1 e1.print_employee()
```

```
num= 1234  name= John  sal= 23000
```

In [123]:

```
1 e2.print_employee()
```

```
num= 0  name=   sal= 0.0
```

Passing parameters to *_init_*()

In [124]:

```
class Employee(object):
    def __init__(self, _num=0, _name='', _salary=0.0):
        self.num = _num
        self.name = _name
        self.salary = _salary

    def print_data(self):
        print ('EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.num,
                                                              self.name,
                                                              self.salary))
    def calculate_tax(self):
        print ('Processing tax for :....')
        self.print_data()
        slab = (self.salary * 12) - 300000
        tax = 0
        if slab > 0:
            tax = slab * 0.1
        print ("tax:", tax)


e1 = Employee(1234, 'John', 23600.0) # e1.__init__(1234, 'John', 23500)
e2 = Employee(1235, 'Samanta', 45000.0) # e2.__init__(1235, 'Samanta', 45000.0)

e1.print_data()
e2.print_data()
```

```
EmpId: 1234, EmpName: John, EmpSalary: 23600.0
EmpId: 1235, EmpName: Samanta, EmpSalary: 45000.0
```

In [125]:

```
e1.calculate_tax()
```

```
Processing tax for :....
EmpId: 1234, EmpName: John, EmpSalary: 23600.0
tax: 0
```

In [126]:

```
e2.calculate_tax()
```

```
Processing tax for :....
EmpId: 1235, EmpName: Samanta, EmpSalary: 45000.0
tax: 24000.0
```

Without __init():__

In [127]:

```
class Employee(object):
    def set_data(self, _num=0, _name='', _salary=0.0):
        self.num = _num
        self.name = _name
        self.salary = _salary

    def print_data(self):
        print ('EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.num,
                                                              self.name,
                                                              self.salary))
    def calculate_tax(self):
        print ('Processing tax for :....')
        self.print_data()
        slab = (self.salary * 12) - 300000
        tax = 0
        if slab > 0:
            tax = slab * 0.1
        print ("tax:", tax)


e1 = Employee()
e1.set_data(1234, 'John', 23600.0)
e2 = Employee()
e2.set_data(1235, 'Samanta', 45000.0)

e1.print_data()
e2.print_data()
```

```
EmpId: 1234, EmpName: John, EmpSalary: 23600.0
EmpId: 1235, EmpName: Samanta, EmpSalary: 45000.0
```

Adding a property at run-time

In [128]:

```
class Example(object):
    def __init__(self):
        self.x = 20
        self.y = 30

    def fun(self):
        self.p = 999

e1 = Example()
e2 = Example()
```

In [129]:

```
e1.x
```

Out[129]:

```
20
```

In [130]:

```
e1.X = 50
```

In [131]:

```
e1.X
```

Out[131]:

```
50
```

Though attribute 'p' is not existing python adds property p to object e1, not to class Example

In [132]:

```
e1.p = 100
```

In [133]:

```
e1.p
```

Out[133]:

```
100
```

fun() also adds 'p' through 'self.p' statement, if 'p' is not existing else it updates with new value, after all self.p equivalent of e1.p inside 'fun'

In [134]:

```
e2.fun() # fun adds a poperty to e1
```

In [135]:

```
e2.p
```

Out[135]:

```
999
```

In [136]:

```
hasattr(e2, 'p')
```

Out[136]:

```
True
```

In [137]:

```
e3 = Example()
```

In [138]:

```
hasattr(e3, 'p')
```

Out[138]:

```
False
```

In [139]:

```
1 isinstance(e1, Example)
```

Out[139]:

```
True
```

In [140]:

```
1 isinstance(e1, object)
```

Out[140]:

```
True
```

Inheritance

Game

In [141]:

```
class Tree(object):

    def __init__(self, leaf_count=0, stem_count=0, trunck_size=0, root_count=0):
        self.leafCount = leaf_count
        self.stemCount = stem_count
        self.trunckSize = trunck_size
        self.rootCount = root_count

    def swing_left(self):
        print('<<<<<<<<<<<<<<')

    def swing_right(self):
        print('>>>>>>>>>>>>>>')


class Human(object):

    def __init__(self, shirt, trouser, shoe):

        self.shirt = shirt
        self.trouser = trouser
        self.shoe = shoe


    def walk(self, direction):

        print("Moving ->" + direction)
        return True


    def run(self, direction):

        print("Running ->" + direction)
        return True


    def jump(self, direction):
        print("Jump ->" + direction)
        return True;


    def action(self):
        self.walk("West")
        self.walk("North")
        self.run("East")
        self.jump("Up")



class InHuman(Human):

    def __init__(self, shirt, trouser, shoe, powers):

        super(InHuman, self).__init__(shirt, trouser, shoe)
        self.powers = powers


    def fly(self, direction):

```

```
        print("fly ->" + direction)
        return True

    # Overriding
    def action(self):

        self.walk("East");
        self.run("North");
        self.fly("High");
        self.fly("Low");


if __name__ == '__main__':
    h = InHuman(1, 2, 3, 4);
    h.action()
```

```
Moving ->East
Running ->North
fly ->High
fly ->Low
```

Employee

In [142]:

```
class EmployeeTax(object):
    def __init__(self, _id, _name, _sal):
        self.eId = _id
        self.eName = _name
        self.eSal = _sal

    def professional_tax(self):
        return 200

    def income_tax(self):
        return self.eSal * 0.3

    def net_salary(self):
        return self.eSal - self.income_tax() - self.professional_tax()

obj = EmployeeTax(1234, 'Jhon', 25000)
obj.net_salary()
```

Out[142]:

```
17300.0
```

Inheritance

Syntax:

```
class <class_name>(<base_Class1>, <base_Class2>, ...):
    statements...

e.g,
class NRIEMployeeTax(EmployeeTax):
    pass
```

In [143]:

```
class NRIEmployeeTax(EmployeeTax):
    pass
```

In [144]:

```
emp = EmployeeTax(1234, 'John', 25000)
nri_emp = NRIEmployeeTax(1234, 'John', 25000)

print (emp.net_salary(),nri_emp.net_salary())
```

```
17300.0 17300.0
```

In [145]:

```
class NRIEmployeeTax(EmployeeTax):
    def __init__(self, _id, _name, _sal, _citizenship):
        self.eId = _id
        self.eName = _name
        self.eSal = _sal
        # -------------
        self.citizenship = _citizenship

    def income_tax(self):
        return self.eSal * 0.4

    def is_us_citizen(self):
        return 'United States' == self.citizenship

    def professional_tax(self):
        if self.is_us_citizen():
            return 2000
        return 200

nri_emp = NRIEmployeeTax(1234, 'John', 25000, 'United States')
```

In [146]:

```
nri_emp.income_tax()
```

Out[146]:

```
10000.0
```

In [147]:

```
nri_emp.net_salary()
```

Out[147]:

```
13000.0
```

delegating functionality to parent constructor, init

In [148]:

```
class NRIEmployeeTax(EmployeeTax):
    def __init__(self, _id, _name, _sal, _citizenship):
        super(NRIEmployeeTax, self).__init__(_id, _name, _sal)
        # -------------
        self.citizenship = _citizenship

    def income_tax(self):
        return self.eSal * 0.4

    def is_us_citizen(self):
        return 'United States' == self.citizenship

    def professional_tax(self):
        if self.is_us_citizen():
            return 2000
        return 200

nri_emp = NRIEmployeeTax(1234, 'John', 25000, 'United States')
nri_emp.net_salary()
```

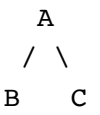
Out[148]:

```
13000.0
```

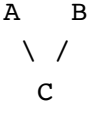
Types of Inheritance

1. Single

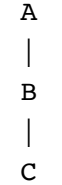
2. Hierarchical



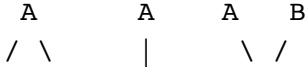
3. Multiple

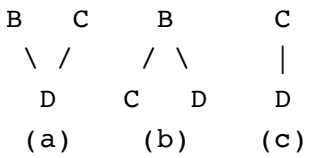


4. Multi-level



5. Hybrid





Diamond problem:
This is a welll known problem in multiple inheritance. When two classes are having an attribute with same name, a conflict ariases when inheriting both of them in a multiple inheritance.
Python has a technique to solve this issue, which is MRO(Method resolution Order).
Python considers attribute of the first class in the inheritance order.

In the below example class D is inheriting A, B and C classes, we can see a conflict for function 'f()'.
As per the MRO in python B's f() is considered for inheritance.

In [149]:

```
class A(object):
    def __init__(self):
        self.x = 100

    def foo(self):
        print("I'm A")

class B(A):
    def __init__(self):
        self.x = "apple"

    def foo(self):
        print ("I'm B")

class C(A):
    def __init__(self):
        self.x = 300
    def foo(self):
        print ("I'm C")

class D(B, C):
    def bar(self):
        print ("Exclusive")

d = D()
d.foo()
```

```
I'm B
```

MRO - Method Resolution Order

Changing method resolution order using *_bases_* attribute of the class.

In the below code, in the last line, we can see class C's f() is called.

In [150]:

```
class A(object):
    def foo(self):
        print ("I'm A")

class B(A):
    def foo(self):
        print ("I'm B")

class C(A):
    def foo(self):
        print ("I'm C")

class D(B, C):
    def bar(self):
        print ("I'm D")

def main():
    d = D()
    d.foo()

    D.__bases__ = (C, B)

    d.foo()

    D.__bases__ = (B, C)

    d.foo()
if __name__ == '__main__':
    main()
```

```
I'm B
I'm C
I'm B
```

In [151]:

```
d = {}
print(type(d))
```

```
<class 'dict'>
```

Polymorphism

Single interface, multiple functionalities.
Polymorphism is, conditional and contextual execution of a functionality.

IS - A Relation

A derived class IS-A base class. All the places in the code where we use Base class objects, we can seamlessly use derived class objects, as all the properties of base class are available in derived class.

In [152]:

```
class A(object):
    def play(self):
        print ('Playing a sport')

class B(A):
    def swim(self):
        print ('Swimming in a pool')

class C(B):
    def sing(self):
        print ('Singing a song')

# User
def action(x):
    x.play()


a = A()
b = B()
c = C()

action(c)
```

```
Playing a sport
```

In [153]:

```
isinstance(c, B)
```

Out[153]:

```
True
```

Without polymorphism:

A designer want to display multiple shapes randomly on a canvas. Circle , Rectangle and Triangle classes are available.

In [154]:

```
from random import shuffle
l = [1, 2, 3, 4, 5]
shuffle(l)
print(l)
```

```
[5, 3, 4, 1, 2]
```

In [155]:

```
from random import shuffle

class Circle(object):
    def circle_display(self):
        print ("I'm the Circle")

class Rectangle(object):
    def rect_display(self):
        print ("I'm the Rectangle")

class Triangle(object):
    def tri_display(self):
        print ("I'm the Triangle")


def render_canvas(shapes):
    for x in shapes:
        if isinstance(x, Circle):
            x.circle_display()
        elif isinstance(x, Rectangle):
            x.rect_display()
        elif isinstance(x, Triangle):
            x.tri_display()

c = Circle()
r = Rectangle()
t = Triangle()

l = [c, r, t]
shuffle(l)

render_canvas(l)
```

```
I'm the Triangle
I'm the Rectangle
I'm the Circle
```

With Ploymorphism

When every subclass is overriding and implementing its own definition in display() method, it becomes very easy for other class to iteract with Shape class, as there is only one interface 'display()'.

Use-Case1: Unified Interface

In [156]:

```
from random import shuffle

class Shape(object):
    def display(self):
        raise NotImplementedError()

class Circle(Shape):
    def display(self):
        print ("I'm the Circle")

class Rectangle(Shape):
    def display(self):
        print ("I'm the Rectangle")

class Triangle(Shape):
    def display(self):
        print ("I'm the Triangle")

def render_canvas(shapes):
    for x in shapes:
        x.display()

c = Circle()
r = Rectangle()
t = Triangle()

l = [c, r, t]
shuffle(l)

render_canvas(l)
```

```
I'm the Rectangle
I'm the Triangle
I'm the Circle
```

Use-Case 2: Incorporating changes into system

In [157]:

```
from random import shuffle

class Shape(object):
    def display(self):
        raise NotImplementedError()

class Circle(Shape):
    def display(self):
        print ("I'm the Circle")

class Rectangle(Shape):
    def display(self):
        print ("I'm the Rectangle")

class Triangle(Shape):
    def display(self):
        print ("I'm the Triangle")

def render_canvas(shapes):
    for x in shapes:
        x.display()

# -----------------------

class RoundedRectangle(Rectangle):
    def display(self):
        print ("I'm the Rounded Rectangle")

c = Circle()
r = RoundedRectangle()
t = Triangle()

l = [c, r, t]
shuffle(l)

render_canvas(l)
```

```
I'm the Rounded Rectangle
I'm the Triangle
I'm the Circle
```

Enforcing rules and mandating overriding

There are no strict rules to mandate overriding a single interface. Developers can ignore overriding display() method and still operate.

In [158]:

```
from random import shuffle

class Shape(object):
    def display(self):
        raise NotImplementedError('Abstract method')

class Circle(Shape):
    def display(self):
        print ("I'm the Circle")

class Rectangle(Shape):
    def display(self):
        print ("I'm the Rectangle")

class Triangle(Shape):
    def display(self):
        print ("I'm the Triangle")

class Hexagon(Shape):
    def draw(self):
        print ('Im unique')

def render_canvas(shapes):
    for x in shapes:
        x.display()

c = Circle()
r = Rectangle()
t = Triangle()
h = Hexagon()

l = [c, r, t, h]
shuffle(l)

render_canvas(l)
```

```
I'm the Rectangle
I'm the Triangle

---------------------------------------------------------------------------
-----
NotImplementedError                       Traceback (most recent call
 last)
<ipython-input-158-c571fabf489f> in <module>()
     33 shuffle(l)
     34
---> 35 render_canvas(l)

<ipython-input-158-c571fabf489f> in render_canvas(shapes)
     23 def render_canvas(shapes):
     24     for x in shapes:
---> 25         x.display()
     26
     27 c = Circle()

<ipython-input-158-c571fabf489f> in display(self)
      3 class Shape(object):
      4     def display(self):
----> 5         raise NotImplementedError('Abstract method')
```

```
      6
      7 class Circle(Shape):

NotImplementedError: Abstract method
```

At least we can stop execution in run-time by raising an exception. But it will be late and not certain.

There is one way to achive this in python. 'abc' module. Using which we can make the base class an abstract class, this ensures uniform interface, by forcing all subclassses to provide implementation.

What is Abstract class, when to use abstract class?

- Abstract classes are classes that contain one or more abstract methods.
- An abstract method is a method that is declared, but contains no implementation.
- Abstract classes can not be instantiated, and require subclasses to provide implementations for the abstract methods.

Using abc module

In Python 3.6

In [ ]:

```
from abc import ABC, abstractmethod

class Base(ABC):
    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass

    def fun():
        print ("have fun!")

class Derived(Base):
    def foo(self):
        print ('Derived foo() called')


d = Derived()
d.foo()
```

We must override all abstract methods, cannot leave them unimplemented.

In [161]:

```
from abc import ABC, abstractmethod

class Base(ABC):

    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass

    def fun():
        print ("have fun!")

class Derived(Base):
    def foo(self):
        print ('Derived foo() called')
    def bar(self):
        print ('Derived bar foo() called')


d = Derived()
d.bar()
```

```
Derived bar foo() called
```

Implementing Shape classes using abc module

In [164]:

```
from random import shuffle
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def display(self):
        pass

class Circle(Shape):
    def display(self):
        print ("I'm the Circle")

class Rectangle(Shape):
    def display(self):
        print ("I'm the Rectangle")

class Triangle(Shape):
    def display(self):
        print ("I'm the Triangle")

class Hexagon(Shape):
    def draw(self):
        print ('Im unique')

def render_canvas(shapes):
    for x in shapes:
        x.display()

c = Circle()
r = Rectangle()
t = Triangle()
h = Hexagon()

l = [c, r, t, h]
shuffle(l)

render_canvas(l)
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-164-99814862d242> in <module>()
     30 r = Rectangle()
     31 t = Triangle()
---> 32 h = Hexagon()
     33
     34 l = [c, r, t, h]

TypeError: Can't instantiate abstract class Hexagon with abstract meth
ods display
```

Abstract classes prevent object instantiation, which gives better understanding and leads to good design.

Hexagon class must override display() method

In [165]:

```
from random import shuffle
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def display(self):
        raise NotImplementedError()

class Circle(Shape):
    def display(self):
        print ("I'm the Circle")

class Rectangle(Shape):
    def display(self):
        print ("I'm the Rectangle")

class Triangle(Shape):
    def display(self):
        print ("I'm the Triangle")

class Hexagon(Shape):
    def display(self):
        print ("I'm the Hexagon and I'm a shape")

def render_canvas(shapes):
    for x in shapes:
        x.display()

c = Circle()
r = Rectangle()
t = Triangle()
h = Hexagon()

l = [c, r, t, h]
shuffle(l)

render_canvas(l)
```

```
I'm the Triangle
I'm the Hexagon and I'm a shape
I'm the Rectangle
I'm the Circle
```

Private Memebrs

- prefixing with __(double undescore) hides property from accessing
- prefixing _ doen't do anything. But by convention, it means, **"not for public use"**. So do not use other's code which has mehtods or attributes prefixed with _(underscore)

In [166]:

```
class A(object):
    def __init__(self):
        self.x = 222
        self._y = 333
        self.__z = 555

    def f1(self):
        print('__z:', self.__z)
        print ("I'm fun")

    def _f2(self):
        print('__z:', self.__z)
        print ("I'm _fun, dont use me, you will be at risk")

    def __f3(self):
        print('__z:', self.__z)
        print ("I'm __fun, you cannot use me")


a = A()
```

Accessing private data members

In [167]:

```
a.x
```

Out[167]:

```
222
```

In [168]:

```
a._y
```

Out[168]:

```
333
```

In [169]:

```
a.__z
```

```
---------------------------------------------------------------------
-----
AttributeError                            Traceback (most recent call
 last)
<ipython-input-169-965fa129e2df> in <module>()
----> 1 a.__z

AttributeError: 'A' object has no attribute '__z'
```

Accessing private members(Hack): Looking at objects dictionary.

In [170]:

```
1 a.__dict__
```

Out[170]:

```
{'_A__z': 555, '_y': 333, 'x': 222}
```

In side object, a dictionary is maintained, __z is actually mangled by interpreter as _A__z

In [171]:

```
1 a._A__z
```

Out[171]:

```
555
```

Accessing private member functions

In [172]:

```
1 a.f1()
```

```
__z: 555
I'm fun
```

In [173]:

```
1 a._f2()
```

```
__z: 555
I'm _fun, dont use me, you will be at risk
```

In [174]:

```
1 a.__f3()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
 last)
<ipython-input-174-251ad2bdaabe> in <module>()
----> 1 a.__f3()

AttributeError: 'A' object has no attribute '__f3'
```

Accessing private Member Functions(Hack): Looking at Class's dictionary.

In [175]:

```
A.__dict__
```

Out[175]:

```
mappingproxy({'_A__f3': <function __main__.A.__f3>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__doc__': None,
              '__init__': <function __main__.A.__init__>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '_f2': <function __main__.A._f2>,
              'f1': <function __main__.A.f1>})
```

In [176]:

```
a._A__f3()
```

```
__z: 555
I'm __fun, you cannot use me
```

Creating inline objects, classes, types

Syntax:

```
    className = type('className', (bases,), {'propertyName' : 'propertyValue'})
```

In [177]:

```
def f(self, eid, name):
    self.empId = eid
    self.name = name

Employee = type('Employee', (object,), {'empId' : 1234, 'name': 'John', '__init_
e = Employee(1234, 'John')
print (e.empId, e.name)
```

```
1234 John
```

Static variables, Static Methods and Class Methods

When we want to execute code before creating first instance of a class, we create static variables and static functions.

In [178]:

```
class A(object):
    # static variable
    db_conn = None
    obj_count = 0

    @staticmethod
    def getDBConnection():
        A.db_conn = "MYSQL"
        print ("db initiated")

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        A.obj_count += 1

    def fun(self):
        if A.db_conn == 'MYSQL':
            print (self.x + self.y + self.z)
        else:
            print ('Error: DB not initialized')

A.getDBConnection()

a1 = A(20, 30, 40)
a2 = A(50, 60, 70)
a3 = A(20, 30, 40)
a4 = A(50, 60, 70)

print ('Object count: ', A.obj_count)
```

```
db initiated
Object count:  4
```

In [179]:

```
a1.fun()
a2.fun()
```

```
90
180
```

In [180]:

```
a1.getDBConnection() # not recommended, Pls donot do this
```

```
db initiated
```

In [181]:

```
a1.obj_count
```

Out[181]:

```
4
```

In [182]:

```
a2.obj_count
```

Out[182]:

```
4
```

In [183]:

```
A.obj_count
```

Out[183]:

```
4
```

In [184]:

```
a1.obj_count = 10
```

In [185]:

```
print (a1.obj_count, a2.obj_count, A.obj_count)
```

```
10 4 4
```

In [186]:

```
a1.__dict__
```

Out[186]:

```
{'obj_count': 10, 'x': 20, 'y': 30, 'z': 40}
```

In [187]:

```
A.obj_count
```

Out[187]:

```
4
```

In [188]:

```
A.__dict__
```

Out[188]:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'A' objects>,
              '__doc__': None,
              '__init__': <function __main__.A.__init__>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              'db_conn': 'MYSQL',
              'fun': <function __main__.A.fun>,
              'getDBConnection': <staticmethod at 0x7fa5e031c9e8>,
              'obj_count': 4})
```

class method : if we need to use class attributes

In [189]:

```
## class method

class A(object):
    # static variables
    logger = None
    dbConn = None
    phi = 3.14
    objectCount = 0

    def __init__(self, x, y , z):
        self.x = x
        self.y = y
        self.z = z
        A.objectCount += 1

    @staticmethod
    def getDBConnection():
        A.dbConn = "Conection to MySQL"
        print("db initiated")


    @classmethod
    def getLogger(cls):
        cls.logger = "logger created"
        print ("logger Initilized")


    def fun(self):
        print ("I'm fun")
        print (A.logger)

```

In [190]:

```
A.__dict__
```

Out[190]:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'A' objects>,
              '__doc__': None,
              '__init__': <function __main__.A.__init__>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              'dbConn': None,
              'fun': <function __main__.A.fun>,
              'getDBConnection': <staticmethod at 0x7fa5e031cf28>,
              'getLogger': <classmethod at 0x7fa5e031cf98>,
              'logger': None,
              'objectCount': 0,
              'phi': 3.14})
```

In [191]:

```
A.getDBConnection() # class method
A.getLogger() # static method
```

```
db initiated
logger Initilized
```

In [192]:

```
A.dbConn # static variable
```

Out[192]:

```
'Conection to MySQL'
```

In [193]:

```
a = A(2, 3, 4)
print (a.__dict__)
```

```
{'x': 2, 'y': 3, 'z': 4}
```

In [194]:

```
l = []
for x in range(5):
    l.append(A(2, 3, 4))

print(A.objectCount)
```

```
6
```

In [195]:

```
class A(object):
    @classmethod
    def get_instance(cls):
        return cls()

    def fun(self):
        print("I'm A")

class B(A):
    def fun(self):
        print("I'm B")

A.get_instance().fun()
B.get_instance().fun()
```

```
I'm A
I'm B
```

Funcion Objects (Functor), Callable objects

Purpose: To maintain common interface across multiple family of classes.

In [196]:

```
class Sqr(object):
    def __init__(self, _x):
        self.x = _x

    def sqr(self):
        return self.x * self.x
```

In [197]:

```
a = Sqr(20)
```

In [198]:

```
print(a.sqr())
```

```
400
```

In [199]:

```
a()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-199-8d7b4527e81d> in <module>()
----> 1 a()

TypeError: 'Sqr' object is not callable
```

In [200]:

```
class Sqr(object):
    def __init__(self, _x):
        self.x = _x

    def __call__(self):
        return self.x * self.x
```

In [201]:

```
s = Sqr(20)
s() # s.__call__()
```

Out[201]:

```
400
```

In [202]:

```
s.__call__()
```

Out[202]:

```
400
```

Multiple family of classes:

In [203]:

```
class Animal(object):
    def run(self):
        raise NotImplementedError()

class Tiger(Animal):
    def run(self):
        print ('Ofcourse! I run')

class Cheetah(Animal):
    def run(self):
        print ('Im the speed')

# -------------------------
class Bird(object):
    def fly(self):
        raise NotImplementedError()

class Eagle(Bird):
    def fly(self):
        print ('I fly the highest')

class Swift(Bird):
    def fly(self):
        print ('Im the fastest')

# -------------------------
class SeaAnimal(object):
    def swim(self):
        raise NotImplementedError()

class Dolphin(SeaAnimal):
    def swim(self):
        print ('I jump aswell')

class Whale(SeaAnimal):
    def swim(self):
        print ('I dont need to')

def observe_speed(obj):
    if isinstance(obj, Animal):
        obj.run()
    elif isinstance(obj, Bird):
        obj.fly()
    elif isinstance(obj, SeaAnimal):
        obj.swim()


obj1 = Cheetah()
obj2 = Swift()
obj3 = Whale()

observe_speed(obj1)
observe_speed(obj2)
observe_speed(obj3)
```

```
Im the speed
Im the fastest
I dont need to
```

In [204]:

```
class Animal(object):
    def __call__(self):
        raise NotImplementedError()

class Tiger(Animal):
    def __call__(self):
        print ('Ofcourse! I run')

class Cheetah(Animal):
    def __call__(self):
        print ('Im the speed')

# -------------------------
class Bird(object):
    def __call__(self):
        raise NotImplementedError()

class Eagle(Bird):
    def __call__(self):
        print ('I fly the hihest')

class Swift(Bird):
    def __call__(self):
        print ('Im the fastest')

# -------------------------
class SeaAnimal(object):
    def __call__(self):
        raise NotImplementedError()

class Dolphin(SeaAnimal):
    def __call__(self):
        print ('I jump aswell')

class Whale(SeaAnimal):
    def __call__(self):
        print ('I dont need to')

def observe_speed(obj):
    obj()


obj1 = Cheetah()
obj2 = Swift()
obj3 = Whale()

observe_speed(obj1)
observe_speed(obj2)
observe_speed(obj3)
```

```
Im the speed
Im the fastest
I dont need to
```

Decorator and Context manager

In [205]:

```
import time
def fun(n):
    x = 0
    for i in range(n):
        x += i*i
    return x
```

In [206]:

```
%%timeit
fun(1000000)
```

```
10 loops, best of 3: 118 ms per loop
```

In [207]:

```
import time

class TimeItDec(object):

    def __init__(self, f):
        self.fun = f

    def __call__(self, *args, **kwargs):
        start = time.clock()
        ret = self.fun(*args, **kwargs)
        end = time.clock()
        print ('Decorator - time taken:',  end - start)
        return ret

class TimeItContext(object):
    def __enter__(self):
        self.start = time.clock()

    def __exit__(self, *args, **kwargs):
        self.end = time.clock()
        print ('Context Manager - time taken:',  self.end - self.start)

@TimeItDec
def compute(n):
    z = 0
    for i in range(n):
        z += i
    return z

if __name__ == '__main__':

    res = compute(1000000)

    with TimeItContext() as tc:
        for i in range(1000000):
            i += i * i

    print ('Sum of 1000000 numbers = ', res)
```

```
Decorator - time taken: 0.10194100000000006
Context Manager - time taken: 0.2954809999999988
Sum of 1000000 numbers =  499999500000
```

In [208]:

```
import time
class TimeIt(object):

    def __init__(self, f=None):
        self.fun = f

    def __call__(self, *args, **kwargs):
        start = time.clock()
        ret = self.fun(*args, **kwargs)
        end = time.clock()
        print ('time taken:',  end - start)
        return ret

    def __enter__(self):
        self.start = time.clock()

    def __exit__(self, *args, **kwargs):
        self.end = time.clock()
        print ('time taken:',  self.end - self.start)

# As decorator
@TimeIt
def compute(x, y):
    z = x + y
    for i in range(1000000):
        z += i

    return z

if __name__ == '__main__':

    z = compute(2, 3)
    # As Context manager
    with TimeIt() as tm:
        for i in range(1000000):
            i += i * i
    print ('Sum of 1000000 numbers = ', z)
```

```
time taken: 0.08089600000000097
time taken: 0.2772710000000007
Sum of 1000000 numbers =  499999500005
```

In [209]:

```
timeit(fun)
```

```
10000000 loops, best of 3: 34.1 ns per loop
```

Function Overloading

In [210]:

```
class Sample(object):
    def fun(self):
        print ('Apple')

    def fun(self, n):
        print ('Apple'*n)


s = Sample()
s.fun()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-210-3633d8ec56b5> in <module>()
      8
      9 s = Sample()
---> 10 s.fun()

TypeError: fun() missing 1 required positional argument: 'n'
```

In [211]:

```
class Sample(object):

    def fun(self):
        print ('Apple')

    def fun(self, n):
        print ('Apple'*n)

s = Sample()
s.fun(4)
```

```
AppleAppleAppleApple
```

- Overloading is static polymorphism
- Method overloading is not having any significance in python.
- Operator methods can be overloaded for a class.
- Objects can be keys in a set or dict. Bydefault id() of the object is considered for hashing.
- To change the hashing criteria,we should override *_hash_*() and *_eq_*()
- Operator overloading can be achieved by overriding corresponding magic methods.

> To implement '<' between objects, we should override *_lt_*(),
> To implement '+' between objects, we should override *_add_*()

- *_lt_*() method is considered for list's sort() method internally
- *_str_*() method is used to represent object as string()
- *_str_*() method is used by 'print' statement when print an object
- *_str_*()method is used when using str() conversion function on objects.

- *_repr_*() is used to syntactically represent object construction using constructor.
  so that, we can reconstruct the object using eval()

Printing objects

In [212]:

```
class Employee(object):
    def __init__(self, _num, _name, _salary):
        self.empNum = _num
        self.empName = _name
        self.empSalary = _salary

    def printData(self):
        print ('EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.empNum,
                                                              self.empName,
                                                              self.empSalary))
    def calculateTax(self):
        slab = (self.empSalary * 12) - 300000
        tax = 0
        if slab > 0:
            tax = slab * 0.1
        print ("tax for empid: {} is {}".format(self.empNum, tax))

e1 = Employee(1234, 'John', 23500.0)
```

In [213]:

```
print(e1)
```

```
<__main__.Employee object at 0x7fa5e03928d0>
```

Above statement is equal to

In [214]:

```
print (str(e1)) # str(e1) is equal to e1.__str__()
```

```
<__main__.Employee object at 0x7fa5e03928d0>
```

Let's implement __str__ method for **Employee** class

In [215]:

```
class Employee(object):
    def __init__(self, _num, _name, _salary):
        self.empNum = _num
        self.empName = _name
        self.empSalary = _salary

    def printData(self):
        print ('EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.empNum,
                                                              self.empName,
                                                              self.empSalary))
    def calculateTax(self):
        slab = (self.empSalary * 12) - 300000
        tax = 0
        if slab > 0:
            tax = slab * 0.1
        print ("tax for empid: {} is {}".format(self.empNum, tax))

    def __str__(self):
        return 'EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.empNum,
                                                              self.empName,
                                                            self.empSalary)


e1 = Employee(1234, 'John', 23500.0)
print(e1)# str(e1) ==> e1.__str__()

```

```
EmpId: 1234, EmpName: John, EmpSalary: 23500.0
```

Perfect, __str__() is called. Lets try another printing technique, simply print 'e1' through shell.

In [216]:

```
e1
```

Out[216]:

```
<__main__.Employee at 0x7fa5e032c1d0>
```

Strange, again same output. Python shell calls a different method other than str(), which is repr(). This method is mainly used for printing a string representation of an object, through which we can reconstruct same object. Generally this string format is different than str() and exactly looks like construction statement.

In the below example we are going to provide both str() and repr()

In [217]:

```
class Employee(object):

    def __init__(self, _num, _name, _salary):
        self.empNum = _num
        self.empName = _name
        self.empSalary = _salary

    def printData(self):
        print ('EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.empNum,
                                                              self.empName,
                                                              self.empSalary))
    def calculateTax(self):
        slab = (self.empSalary * 12) - 300000
        tax = 0
        if slab > 0:
            tax = slab * 0.1
        print ("tax for empid: {} is {}".format(self.empNum, tax))

    def __str__(self):
        return 'EmpId: {}, EmpName: {}, EmpSalary: {}'.format(self.empNum,
                                                              self.empName,
                                                           self.empSalary)

    def __repr__(self):
        return "Employee({}, '{}', {})".format(self.empNum,
                                               self.empName,
                                            self.empSalary)

e1 = Employee(1234, 'John', 23500.0)
```

In [218]:

```
print (e1) # invokes e1.__str__() or str(e1)
```

```
EmpId: 1234, EmpName: John, EmpSalary: 23500.0
```

In [219]:

```
e1 # invokes e1.__repr__() or repr(e1)
```

Out[219]:

```
Employee(1234, 'John', 23500.0)
```

Difference between above two printing statements is

In [220]:

```
e1 # repr(e1) ==> e1.__repr__()
```

Out[220]:

```
Employee(1234, 'John', 23500.0)
```

In [221]:

```
repr(e1)
```

Out[221]:

```
"Employee(1234, 'John', 23500.0)"
```

In [222]:

```
e1.__repr__()
```

Out[222]:

```
"Employee(1234, 'John', 23500.0)"
```

eval() fiunction

Executes string as code

In [223]:

```
eval('20 + 30')
```

Out[223]:

```
50
```

In [224]:

```
x = 20
y = 40
eval('x*y', globals(), locals())
```

Out[224]:

```
800
```

In [225]:

```
obj = eval(repr(e1))
```

In [226]:

```
id(e1), id(obj)
```

Out[226]:

```
(140350408172992, 140350407882624)
```

repr() : evaluatable string representation of an object (can "eval()" it, meaning it is a string representation that evaluates to a Python object

With the return value of repr() it should be possible to recreate our object using eval().

Operator overloading

In [227]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)
    def __repr__(self):
        return "Employee({}, '{}', {})".format(self.eid, self.ename,
                                               self.esal)

e1 = Employee(1234, 'John corner', 5000.0)
e2 = Employee(1235, 'Stuart', 26000.0)
e3 = Employee(1236, 'snadra', 19000.0)
```

In [228]:

```
e2 < e3
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-228-460665f828f4> in <module>()
----> 1 e2 < e3

TypeError: '<' not supported between instances of 'Employee' and 'Empl
oyee'
```

In [229]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)
    def __repr__(self):
        return 'Employee({}, {}, {})'.format(self.eid, self.ename,
                                               self.esal)
    def __lt__(self, other):
        print ('lt called!')
        return self.esal < other.esal


e1 = Employee(1234, 'John', 5000.0)
e2 = Employee(1235, 'Stuart', 25000.0)
e3 = Employee(1236, 'snadra', 19000.0)

```

In [230]:

```
e2 < e3 # internally works like this, e2.__lt__(e3)
```

```
lt called!
```

Out[230]:

```
False
```

In [231]:

```
e2 + e3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-231-70db920a5a56> in <module>()
----> 1 e2 + e3

TypeError: unsupported operand type(s) for +: 'Employee' and 'Employe
e'
```

In [232]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)
    def __repr__(self):
        return 'Employee({}, {}, {})'.format(self.eid, self.ename,
                                             self.esal)
    def __lt__(self, other):
        return self.esal < other.esal

    def __add__(self, other):
        return self.esal + other.esal

e1 = Employee(1234, 'John', 5000.0)
e2 = Employee(1235, 'Stuart', 25000.0)
e3 = Employee(1236, 'snadra', 19000.0)

```

In [233]:

```
e1 + e2 # internally works like this, e1.__add__(e2)
```

Out[233]:

```
30000.0
```

In [234]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)

    def __repr__(self):
        return 'Employee({}, {}, {})'.format(self.eid, self.ename,
                                             self.esal)

e1 = Employee(1234, 'John', 5000.0)
e2 = Employee(1235, 'Stuart', 25000.0)
e3 = Employee(1236, 'sandra', 19000.0)
e4 = Employee(1236, 'sandra', 19000.0)
```

In [235]:

```
set([e1, e2, e3, e4])
```

Out[235]:

```
{Employee(1234, John, 5000.0),
 Employee(1235, Stuart, 25000.0),
 Employee(1236, sandra, 19000.0),
 Employee(1236, sandra, 19000.0)}
```

In [236]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)
    def __repr__(self):
        return 'Employee({}, {}, {})'.format(self.eid, self.ename,
                                             self.esal)
    def __hash__(self):
        print ('Hash called')
        return hash(self.eid)

e1 = Employee(1234, 'John', 5000.0)
e2 = Employee(1235, 'Stuart', 25000.0)
e3 = Employee(1236, 'sandra', 19000.0)
e4 = Employee(1236, 'sandra', 19000.0)
```

In [237]:

```
set([e1, e2, e3, e4])
```

```
Hash called
Hash called
Hash called
Hash called
```

Out[237]:

```
{Employee(1234, John, 5000.0),
 Employee(1235, Stuart, 25000.0),
 Employee(1236, sandra, 19000.0),
 Employee(1236, sandra, 19000.0)}
```

In [238]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)
    def __repr__(self):
        return 'Employee({}, {}, {})'.format(self.eid, self.ename,
                                             self.esal)
    def __hash__(self):
        print ('Hash called')
        return hash(self.eid)

    def __eq__(self, other):
        print ('eq called')
        return self.eid == other.eid

e1 = Employee(1234, 'John', 5000.0)
e2 = Employee(1235, 'Stuart', 25000.0)
e3 = Employee(1236, 'sandra', 19000.0)
e4 = Employee(1236, 'sandra', 19000.0)
```

In [239]:

```
set([e1, e2, e3, e4])
```

```
Hash called
Hash called
Hash called
Hash called
eq called
```

Out[239]:

```
{Employee(1234, John, 5000.0),
 Employee(1235, Stuart, 25000.0),
 Employee(1236, sandra, 19000.0)}
```

Note:

If we want to store objects as set elements or keys in a dictionary, _hash_() and _eq_() both must be overriden.

Because, for different values, if hash codes are same,it should compare their values to check both are different are not.
If different, it stores values in the same hash bucket, else ignores. If we do not implement *_eq_*(), set doesn't consider
user defined *_hash_*() method.

In [240]:

```
class Employee(object):
    def __init__(self, _id, _name, _sal):
        self.eid = _id
        self.ename = _name
        self.esal = _sal

    def __str__(self):
        return str(self.eid) + ', ' + self.ename + ', ' + str(self.esal)
    def __repr__(self):
        return 'Employee({}, {}, {})'.format(self.eid, self.ename,
                                             self.esal)

    def __lt__(self, other):
        print('lt is called')
        return self.esal < other.esal

    def __hash__(self):
        return hash(self.eid)

    def __eq__(self, other):
        print ('Eq Called')
        return self.eid == other.eid


e1 = Employee(1234, 'John', 5000.0)
e2 = Employee(1235, 'Stuart', 25000.0)
e3 = Employee(1236, 'sandra', 19000.0)
e4 = Employee(1236, 'sandra', 19000.0)
```

Sorting Objects

In [241]:

```
# sort method internally using  __lt__() method of Employee class
# esal is the criteria.

l = [Employee(1237, 'Stuart', 1000),
     Employee(1234, 'John', 25000),
     Employee(1235, 'Stuart', 15000),
     Employee(1236, 'snadra', 19000)]

l.sort()
l
```

```
lt is called
lt is called
lt is called
lt is called
lt is called
lt is called
```

Out[241]:

```
[Employee(1237, Stuart, 1000),
 Employee(1235, Stuart, 15000),
 Employee(1236, snadra, 19000),
 Employee(1234, John, 25000)]
```

Explicitly providing creteria

In [242]:

```
l.sort(key=lambda x:x.eid, reverse=True)
l
```

Out[242]:

```
[Employee(1237, Stuart, 1000),
 Employee(1236, snadra, 19000),
 Employee(1235, Stuart, 15000),
 Employee(1234, John, 25000)]
```

In [243]:

```
sorted(l, key=lambda x:x.esal)
```

Out[243]:

```
[Employee(1237, Stuart, 1000),
 Employee(1235, Stuart, 15000),
 Employee(1236, snadra, 19000),
 Employee(1234, John, 25000)]
```

In [244]:

```
max(l, key=lambda x:x.eid)
```

Out[244]:

```
Employee(1237, Stuart, 1000)
```

In [245]:

```
min(l, key=lambda x:x.esal)
```

Out[245]:

```
Employee(1237, Stuart, 1000)
```

Function Overloading

In [246]:

```
class A(object):
    def fun(self):
        print("Hello...")

    def fun(self, x):
        print(x * x)
```

In [247]:

```
a = A()
```

In [248]:

```
a.fun()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-248-7301c6f31cb5> in <module>()
----> 1 a.fun()

TypeError: fun() missing 1 required positional argument: 'x'
```

In [249]:

```
a.fun(5)
```

```
25
```

In [250]:

```
class A(object):
    def fun(self, x):
        print(x * x)

    def fun(self):
        print("Hello...")

a = A()
```

In [251]:

```
a.fun()
```

```
Hello...
```