Python Programming - PYTHONGURU

by Naren Allam

Chapter 12

Object Orientation

Topics Covering

- Once upon a time when there was no object orientation.
- 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 immagine our software development career,...

Mr.Alex, who owns a bank ABX, is our client now. And good news is that, we were chosen 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 endedup with 100 functions and 40 global varaibles(may contains lists, dictionaries).

We wrote all the code in a single file named 'banking_sytem.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 possibiliy 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 recomended. 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_sytem.py which cintains 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 deveoper 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.

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 but funtions(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 segregated them as,

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

if bank wants to start 100 peronal banking only branches. now, we need 100 units of personal loans functionality

(20 funcs + 10 variables) for 100 branches

Note: Functions(behaviour) are common, but, a set of 10 vars are required for each branch.

We need a dynamic way to create any number of such units.
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 by using a looping statement, creating hundreds of dicts is extremely simple.

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 now. To create a type, we use a construct called "class".

Introducing class

Syntax:

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

    def __init__(self, args, ...):
        self.attrOne = some_val
        self.attrTwo = some_val
        self.attrThree = some_val

    def first_method(self, args, ...):
        # code
    def second_method(self, args, ...):
        # code
```

Class

- The name 'class' has come from - "classification of a type"
- Class is a model of any real-world entity, process or an idea.
- Class contains data (member variables) and actions(member functions or methods)
- class is a type, it's instances(variables) are called as objects.
- object is the physical existance of a class
- A class is an extensible program-code-template for reusablity.
- 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.

How Object Orientation is solving all the above 3 problems?

Let's discuss them in reverse order.

1. Scalability - By making Personal Banking as a type(class), we can create multiple units of same functionality by instantiation.

2. Security - By hiding data inside a class and giving access to only its bounded methods prevent accidental changes.

3. Spaghetti code - As bounded methods only can modify datain a class, we cannot define methods outside the class boundary, thus removing the scope for spaghetti code.

Thinking the problem in object orientation:

1. We identified functions and variables related to Personal Loan functionality and we kept them together. This is called - **data binding**. **Data binding** happens as a resultant of functional segregation.

2. lets bind these 20 funcs and 10 vars and isolate(hide) data inside a container - **data hiding**

3. The container is - **class**

4. We should not restrict everything inside the container, as functions have business with other classes(types). So they should interact with external funcs. Lets expose few funcs to interact with external classes - **abstraction**. The public face of a class, the world sees is **abstraction**.

5. A well defined boundary should exist between data-hiding and abstraction, to ensure hiding unnecessary details and to expose required interfaces, this boundary is **encapulation**

6. How do we resuse existing code? - to extend or to create a variant of existing type we need a mechanism, which is **inheritance**

7. To incorporate new coponents(classes) into the exisiting system, whithout disturbing existing coponent interactionns, a class requires a special property called **polymorphism**. This can be achieved with the help of **overriding** & **overloading** techniques.

Note: Object orientation is all about - in-advance planning of application design by anticipating future changes.

Features of Object orientation

Data binding:

Process of defining functions to work with a specific data. Functional segregation results to data binding, keeping methods n attributes related to functionality together is said to be data binding.

Abstraction:

Providing simple and useful interfaces by hiding complex details. Abstraction isn't about solving a particular problem faster or with fewer resources. Instead, the goal of abstraction is to allow us to arrange information more quickly and reliably in our heads and ignore irrelevant details. The purpose of abstraction is largely for helping humans think, rather than helping computers work.

Eg: A person sees color and brand of a car when a attractive car is going on the road, not the engine and chasis.

Eg: Today a mobile phone can fit into your palm. We hardly press a button today to make a call. Actually 5000+ patents caused the mobile to evolve into present shape. We do not need to go through all the research papers to operate a mobile. The wonderfull experience that we are enjoying today and not knowing the complex details is called abstraction.

A also contains chasis and engine too. The sense of seeing the color and the brand is the abstraction created by the car designer.

Data Hiding:

Hiding all

1. attributes
2. methods which are indirectly serving irrelevent data,
3. methods which are not part of abstraction

is called data hiding.

Encapsulation:

Drawing a boundary between abstraction and data hiding. It decides how much detailing has to be exposed and how much has to be hidden.

Inheritance:

It is a way of reusing existing functionality either for extension or to create a variant(involves overriding).

Polymorphism:

Single interface with multiple functionalities. It is conditional execution of multiple functionalities through same interface.

Summary

- Class is a construct with which we can create abstract datatypes.
- Converting a functionality into a type makes scalability possible.
- Data hiding is solving the problem of security because an external method cannot access * a private member of a class unless it becomes member of a class.
- Data binding keeps data and methods at one place and it does not allow sphagetti code.

In [ ]:
```
d = dict()
```

In [ ]:
```
d.
```

In [ ]:
```
1. Data binding
2. Data hiding
-------------------------
3. Abstration
4. Encapsulation
*************************

5. Inheritance
6. Polymorphism
```

Upgrading Personal Banking sytem with Object Orientation ...

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

class PersonalBanking(object):
        # COMMON
        num_of_customers = 0

        # 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():
            ...
```

In [ ]:
```
fro x in range(1000):
    p = PersonalBanking()
```

Let's start with a simple example

Modeling an employee

In [1]:
```
class Employee(object): # POD
    def __init__(self):
        self.num = 0
        self._name = ''
        self.__salary = 0.0

    def get_salary(self):
        return self.__salary

    def _set_salary(self, _sal):
        self.salary = _sal

    def __get_name(self):
        return self.name

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

In [ ]:
```

```

Creating an object for class **Employee**

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

In [22]:
```
class Employee(object): # POD
    num_of_emps = 0
    def __init__(self):
        self.num = 0
        self.name = ''
        self.salary = 0.0

    def get_salary(self):
        return self.salary

    def set_salary(self, _sal):
        self.salary = _sal

    def get_name(self):
        return self.name

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

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

In [14]:
```
e1.__dict__
```

Out[14]:
```
{'num': 0, 'name': '', 'salary': 0.0}
```

In [15]:
```
e1.num = 1234
e1.name = 'Jhon'
e1.salary = 400000.0
```

In [16]:
```
print(e1.num, e1.name, e1.salary)
```
```
1234 Jhon 400000.0
```

In [17]:
```
e1.print_employee()
```
```
num= 1234  name= Jhon  sal= 400000.0
```

In [20]:
```
e2 = Employee()
e2.print_employee()
```
```
num= 0  name=   sal= 0.0
```

In [ ]:
```
e2.get_salary()
```

In [ ]:
```
e1.get_salary()
```

In [23]:
```
Employee.__dict__
```

Out[23]:
```
mappingproxy({'__module__': '__main__',
              'num_of_emps': 0,
              '__init__': <function __main__.Employee.__init__(self)>,
              'get_salary': <function __main__.Employee.get_salary(self)>,
              'set_salary': <function __main__.Employee.set_salary(self, _sal)
>,
              'get_name': <function __main__.Employee.get_name(self)>,
              'print_employee': <function __main__.Employee.print_employee(sel
f)>,
              '__dict__': <attribute '__dict__' of 'Employee' objects>,
              '__weakref__': <attribute '__weakref__' of 'Employee' objects>,
              '__doc__': None})
```

In [ ]:
```
# fig required
```

In [ ]:
```
e2 = Employee()
```

In [ ]:
```
e2.__dict__
```

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 [ ]:
```
e1.num
```

In [ ]:
```
e1.salary
```

In [ ]:
```
e1.get_salary()
```

In [ ]:
```
print(e1.num, e1.name, e1.salary)
```

Accessing data members and member functions explicitely

In [ ]:
```
e1.num = 1234
e1.name = 'John'
e1.salary = 23000

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

In [ ]:
```
e1.print_employee()
```

In [ ]:
```
e2.print_employee()
```

In [ ]:
```
What can be a class?
1. Entity - Exployee, Book, txn, Student
2. Process - Library Management, Personal Banking,
3. Idea - Reconciliation, Expense Sharing

class Student:
    def __init__(self):
        self.rollno = None
        self.name = None
        self.branch = None
        self.year = None
        self.email = None
        self.phno = None

class Book:
    def __init__(self):
        self.book_id = None
        self.isbn = None
        self.authors = []
        self.pages = None
        self.price = None
        self.publisher = None
        self.dimensions = None


class LibraryManagementSystem:
    def __init__(self):
        self.books = []
        self.students = []

    def issue(self, stu, book):
        pass

    def renewal(self, stu, book):
        pass

    def return_book(self, stu, book):
        pass

    def add_book(self):
        pass
    def add_Student(self):
        pass
```

```
Design classes for Parking lot
Design classes for Restaurant Management System
```

In [ ]:

```
class Vehicle:
    def __init__(self):
        self.vehicle_no = None
        self.type = None
        self.in_time = None
        self.out_time = None

class Slot:
    def __init__(self):
        self.slot_id = None
        self.location_id = None


class ParkingLot:

    def __init__(self):
        self.slots = set()
        self.vechicles = set()
        self.registry = {}

    def allocate(vehicle):
        slot =  get_available_slot(self):
        if slot:
            self.registry[slot] = vechicle
        return None

    def deallocate(slot):
            self.registry

    def get_available_slot(self):
        pass
```

In [2]:

```
class Table:
    def __init__(self):
        self.seating = None
        self.table_id = None
        self.reserved = None
        self.in_time = None
        self.out_time = None
        self.table_type = None

class Item:
    def __init__(self):
        self.item_id = None
        self.type = None # desert/starter/main course/breakfast/baverage
        self.price = None
        self.vegan = None # True/False
        self.addon = None # True/False
        self.available_time = None


class RestaurantManagementSystem:
    def __init__(self):
        self.orders = {}

    def make_order(self):
        pass
    def place_order(self):
        pass

    def cancel_order(self):
        pass

    def re_order(self, order_id):
        pass

    def peprare_check(self):
        pass

    def settle_check(self):
        pass
```

Passing paramets to __init__()

In [1]:

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

    def print_data(self):
        print (f'EmpId: {self.num}, EmpName: {self.name}, EmpSalary: {self.sa

    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 = EmployeeTax(1234, 'John', 23600.0) # Employee.__new__().__init__(1234, '
e2 = EmployeeTax(1235, 'Samanta', 45000.0) # Employee.__new__().__init__(1235
e3 = EmployeeTax() # Employee.__new__().__init__()

e1.calculate_tax()
e2.calculate_tax()
e3.calculate_tax()
```

```
EmpId: 1234, EmpName: John, EmpSalary: 23600.0
EmpId: 1235, EmpName: Samanta, EmpSalary: 45000.0
EmpId: 0, EmpName: , EmpSalary: 0.0
```

In [ ]:

```
l1 = [...]
l2 = [...]
l3 = [...]
l4 = [...]

l1.sort()

sorted(l1)

def sort(, reverse=False, key=None)
```

In [ ]:

```
e1.calculate_tax() == calculate_tax(e1)
```

In [ ]:
```
e2.calculate_tax()
```

In [ ]:
```
e2.salary = 400000
```

In [ ]:
```
e2.calculate_tax()
```

In [ ]:
```
e1.__dict__
```

In [ ]:
```
Employee.__dict__
```

In [ ]:
```
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.

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

In [ ]:
```
e1.num
```

In [ ]:
```
e1.name
```

In [ ]:
```
e1.__salary
```

In [ ]:
```
e2.calculate_tax()
```

Adding a property at run-time

In [4]:
```
class Example(object):
    def __init__(self):
        self.x = 20
        self.y = 30

    def fun(self):
        self.X = 999

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

In [5]:
```
e1.fun()
```

In [8]:
```
e1.hello = 999
```

In [11]:
```
e1.g = 44
```

In [17]:
```
e1.__dict__
```

Out[17]:
```
{'x': 333, 'y': 30, 'X': 999, 'hello': 999, 'g': 44}
```

In [16]:
```
e2.__dict__
```

Out[16]:
```
{'x': 20, 'y': 30}
```

In [15]:
```
e1.x = 333
e1.x
```

Out[15]:
```
333
```

In [ ]:
```
e1.X = 50
```

In [ ]:
```
e1.X = 100
```

In [ ]:
```
e1.p = 555
```

In [ ]:
```
e1.fun()

e1.z = 20
```

In [ ]:
```
e1.p
```

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

In [ ]:
```
e1.p = 100
```

In [ ]:
```
e1.p
```

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

In [19]:
```
e2.p = 30
```

In [20]:
```
hasattr(e2, 'p')
```

Out[20]:
```
True
```

In [ ]:
```
e3 = Example()
```

In [ ]:
```
hasattr(e3, 'p')
```

In [23]:
```
isinstance(e1, EmployeeTax)
```

Out[23]:
```
False
```

In [24]:
```
isinstance(e1, object)
```

Out[24]:
```
True
```

In [ ]:
```
l = [2, 4, 5]
x = 20
print(type(x), type(l))
```

In [ ]:
```
l[0] --> l.getitem(0)
d['orange'] --> d.getitem('orange')
x = 20
x.get()
```

In [ ]:
```
* data binding  - first stage  X
* encapsulation - sets a boundary
* abstraction   - What others see
--------------------------------------
* data hiding   - What they donot need to see
* inheritance   - Version, Variant
* polymorphism  - consistent interfaces, introduce new changes easily
```

In [10]:
```
from uuid import uuid1 as gen_uuid

class BlueTooth4Comm(object):
    UUID = gen_uuid()
    def __init__(self, _header_format='@@@@BlueTooth4Comm@@@@', _range='2.4GH

        self.__header = _header_format
        self.__range = _range
        self.__mtu = _mtu
        self.__data= None
        self.__packets = []
        self.__connected_devices = []


    def   __register_device(self, device_id):
        self.__connected_devices.append(device_id)
        print(f"Device {device_id} has been successfully registered")

    def   __unregister_device(self, device_id):
        self.connected_devices.remove(device_id)
        print(f"Device {device_id} has been successfully unregistered")

    def   __load_data(self, file_loc):
        print(f"Loading data from {file_loc}")
        with open(file_loc, encoding="utf-8") as f:
            self.data = f.read()

    def   __get_selected_device(self):
        return self.connected_devices[-1]

    def   __prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
            self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    def   __send_data(self, device, file_loc):
        '''
        Sending data to selected device
        '''
        for packet in self.packets:
            print(f"Sending packet '{packet}' to {device}")

if __name__ == '__main__':

    bc4 = BlueTooth4Comm()
    bc4.register_device('234-567-789')
    bc4.load_data('sample_data.txt')
    bc4.prep_data()
    bc4.send_data(bc4.get_selected_device())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-96ce1e9470f7> in <module>
     53
     54     bc4 = BlueTooth4Comm()
---> 55     bc4.register_device('234-567-789')
     56     bc4.load_data('sample_data.txt')
     57     bc4.prep_data()

<ipython-input-10-96ce1e9470f7> in register_device(self, device_id)
     16
     17     def   register_device(self, device_id):
---> 18         self.connected_devices.append(device_id)
     19         print(f"Device {device_id} has been successfully registered")
     20

AttributeError: 'BlueTooth4Comm' object has no attribute 'connected_devices'
```

In [8]:

```
from uuid import uuid1 as gen_uuid

class BlueTooth4Comm(object):
    def __init__(self, _header_format='@@@@BlueTooth4Comm@@@@', _range='2.4GH
        self.__UUID = gen_uuid()
        self.__header = _header_format
        self.__range = _range
        self.__mtu = _mtu

        self.__data= None
        self.__packets = []
        self.__connected_devices = []

    def   __load_data(self, file_loc):
        print(f"Loading data from {file_loc}")
        with open(file_loc, encoding="utf-8") as f:
            self.__data = f.read()

    def   __register_device(self, device_id):
        self.__connected_devices.append(device_id)
        print(f"Device {device_id} has been successfully registered")

    def   __unregister_device(self, device_id):
        self.__connected_devices.remove(device_id)
        print(f"Device {device_id} has been successfully unregistered")

    def   __get_selected_device(self):
        return self.__connected_devices[-1]

    def   __prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.__data)/self.__mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.__data), self.__mtu):
            seq_no = "SEQNO: " + str(x//self.__mtu) + " # "
            self.__packets.append(self.__header + seq_no + self.__data[x:x+se

    def  __send_data(self, device, file_loc):
        '''
        Sending data to selected device
        '''
        self.__load_data(file_loc)
        self.__prep_data()
        for packet in self.__packets:
            print(f"Sending packet '{packet}' to {device}")

if __name__ == '__main__':

    bc4 = BlueTooth4Comm()
    bc4.register_device('234-567-789')
    bc4.send_data(bc4.get_selected_device(), 'sample_data.txt')
```

```
Device 234-567-789 has been successfully registered
Loading data from sample_data.txt
25 packets are sent
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 0 # Sometimes host-based and name
space-based' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 1 #  UUID values are not “differe
nt enough”.' to 234-567-789
```

```
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 2 #  For example, in cases where
you want to' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 3 #  use the UUID as a lookup ke
y, a more ra' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 4 # ndom sequence of values with
more differ' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 5 # entiation is desirable to avo
id collisio' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 6 # ns in a hash table. Having va
lues with f' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 7 # ewer common digits also makes
it easier ' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 8 # to find them in log files. To
add greate' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 9 # r differentiation in your UUI
Ds, use uui' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 10 # d4() to generate them using
random input' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 11 #  values.It is also useful in
some contex' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 12 # ts to create UUID values fro
m names inst' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 13 # ead of random or time-based
values. Vers' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 14 # ions 3 and 5 of the UUID spe
cification u' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 15 # se cryptographic hash values
(MD5 or SHA' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 16 # -1) to combine namespace-spe
cific seed v' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 17 # alues with “names” (DNS host
names, URLs,' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 18 #  object ids, etc.). There ar
e several we' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 19 # ll-known namespaces, identif
ied by pre-d' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 20 # efined UUID values, for work
ing with DNS' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 21 # , URLs, ISO OIDs, and X.500
Distinguishe' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 22 # d Names. You can also define
your own ap' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 23 # plication- specific namespac
es by genera' to 234-567-789
Sending packet '@@@@BlueTooth4Comm@@@@SEQNO: 24 # ting and saving UUID value
s.' to 234-567-789
```

Inheritance

In [ ]:

```python
class Base(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        print (self.x + self.y)

class Derived(Base):
    '''
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        print (self.x + self.y)
    '''
    pass

b = Base(20, 30)
b.add()

d = Derived(40, 50)
d.add()
```

In [ ]:

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

    def add(self):
        print (self.x + self.y)

    def myhelp(self):
        print("This class does the addition")

class Derived(Base):
    """
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        print (self.x + self.y)

    def myhelp(self):
        print("This class does the addition")
    """
    def __init__(self, x, y, z):
        #super().__init__(x, y)
        #super(Derived, self).__init__(x, y)
        self.x = x
        self.y = y
        self.z = z

    def square(self):
        print(self.z ** 2)

b = Base(20, 30)
b.add()

d = Derived(40, 50, 60)
d.add()
d.square()
```

In [ ]:

```
IS-A
```

In [ ]:

```
def add(x, y, z):
    print (x + y + z)


class Parent(object): # 1.0
    def __init__(self, x, y):
        self.x = x
        self.y = y
        # 10000 lines
    def add(self):
        print (self.x + self.y)

    def div(self):
        print (self.x // self.y)

    def sub(self):
        print (self.x - self.y)

    def myhelp(self):
        print("This class does the addition")

class Child(Base): # 2.0
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def add(self):
        print (self.x + self.y + self.z)

    def prod(self):
        print(self.x * self.y * self.z)

class SubChild(Child): # 3.0

    def mod(self):
        print (self.x % self.y)

obj = Parent(20, 30)
#obj = Child(20, 30, 40)
#obj = SubChild(20, 30, 40)

obj.add()
```

In [ ]:

```
def add(x, y, z):
    print(x + y + z)


class Parent(object):  # 1.0
    def __init__(self, x, y):
        self.x = x
        self.y = y
        # 10000 lines

    def add(self):
        print(__class__)
        print(self.x + self.y)

    def div(self):
        print(self.x // self.y)

    def sub(self):
        print(self.x - self.y)



class Child(Parent):  # 2.0
    """"
    def __init__(self, x, y):
        self.x = x
        self.y = y
        # 10000 lines
    def add(self):
        print(self.x + self.y)

    def div(self):
        print(self.x // self.y)

    def sub(self):
        print(self.x - self.y)
    """
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def add(self):
        print(__class__)
        print(self.x + self.y + self.z)

    def prod(self):
        print(self.x * self.y * self.z)


class SubChild(Child):  # 3.0
    """"
     def __init__(self, x, y):
         self.x = x
         self.y = y
         # 10000 lines

     def add(self):
         print(self.x + self.y)

     def div(self):
         print(self.x // self.y)

     def sub(self):
         print(self.x - self.y)


    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def add(self):
        print(self.x + self.y + self.z)

    def prod(self):
        print(self.x * self.y * self.z)

        """
    def mod(self):
        print(self.x % self.y)


obj1 = Parent(20, 30)
obj2 = Child(20, 30, 40)
obj3 = SubChild(20, 30, 40)

'''obj1.add()
obj1.sub()
obj1.div()
print('------------')
obj2.add()
obj2.sub()
obj2.div()
```

```
obj2.prod()
print('------------')
obj3.add()
obj3.sub()
obj3.div()
obj3.prod()
obj3.mod()'''

def action(obj):
    obj.add()
    obj.sub()
    obj.div()

action(obj3)
```

In [ ]:

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

    def add(self):
        print (self.x + self.y)

    def myhelp(self):
        print("This class does the addition")

class Derived(Base):
    def __init__(self, x, y, z):
        super(Derived, self).__init__(x, y)
        self.z = z

    def add(self):
        print (self.x + self.y + self.z)

    def square(self):
        print(self.z ** 2)



b = Base(20, 30)
b.add()

d = Derived(40, 50, 60)
d.add()
```

In [ ]:

```
Entity - Employee, Book, Person - POD
Process - EmployeeTaxation, PersonalLoan, LibraryManagment
Idea - Reconciliation, OnlineBanking
```

Use Case - Bluetooh 4.0 File Transfer

In [ ]:

```
!type sample_data.txt
```

In [ ]:

```
from uuid import uuid1 as gen_uuid

class BlueTooth4Comm(object):
    def __init__(self, _header_format='@@@@BlueTooth4Comm@@@@', _range='2.4GH
        self.__UUID = gen_uuid()
        self.__header = _header_format
        self.__range = _range
        self.__mtu = _mtu

        self.__data= None
        self.__packets = []
        self.__connected_devices = []

    def   __load_data(self, file_loc):
        print(f"Loading data from {file_loc}")
        with open(file_loc, encoding="utf-8") as f:
            self.data = f.read()

    def   register_device(self, device_id):
        self.connected_devices.append(device_id)
        print(f"Device {device_id} has been successfully registered")

    def   unregister_device(self, device_id):
        self.connected_devices.remove(device_id)
        print(f"Device {device_id} has been successfully unregistered")

    def   __get_selected_device(self):
        return self.connected_devices[-1]

    def   __prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
            self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    def  send_data(self, device, file_loc):
        '''
        Sending data to selected device
        '''
        for packet in self.packets:
            print(f"Sending packet '{packet}' to {device}")

if __name__ == '__main__':

    bc4 = BlueTooth4Comm()
    bc4.register_device('234-567-789')
    bc4.load_data('sample_data.txt')
    bc4.prep_data()
    bc4.send_data(bc4.get_selected_device())
```

Inheritance

Syntax:

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

Bluetooth 5.0

Duel Mode has been introduced: In duel mode we can stream same audio to multiple audio devices

In [ ]:
```
from uuid import uuid1 as gen_uuid

class BlueTooth4Comm(object):
    def __init__(self, _header_format='@@@@BlueTooth4Comm@@@@',_range='2.4GHz
        self.UUID = gen_uuid()
        self.header = _header_format
        self.range = _range
        self.data= None
        self.mtu = _mtu
        self.packets = []
        self.connected_devices = []

    def load_data(self, file_loc):
        print(f"Loading data from {file_loc}")
        with open(file_loc, encoding='utf-8') as f:
            self.data = f.read()

    def register_device(self, device_id):
        self.connected_devices.append(device_id)
        print(f"Device {device_id} has been successfully registered")

    def unregister_device(self, device_id):
        self.connected_devices.remove(device_id)
        print(f"Device {device_id} has been successfully unregistered")

    def get_selected_device(self):
        return self.connected_devices[-1]

    def prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
            self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    def send_data(self, device):
        '''
        Sending data to selected device
        '''
        for seq, packet in enumerate(self.packets):
            print(f"Sending packet '{packet}' to {device}")



class BlueTooth5Comm(BlueTooth4Comm):

    def __init__(self, _header_format='#####BT5Header#####', _range='5GHz', _
        super(BlueTooth5Comm, self).__init__(_header_format, _range, _mtu)

        self.concurrent_devices = []

    def set_concurrent_devices(self, devices=[]):
        self.concurrent_devices = devices

    def get_selected_devices(self):
        return self.concurrent_devices
```

```python
    def send_data_many(self, devices=[]):
        for device in devices:
            self.send_data(device)


if __name__ == '__main__':

    bc5 = BlueTooth5Comm();
    bc5.register_device('999-567-789')
    bc5.register_device('888-567-789')
    bc5.register_device('666-567-777')
    bc5.register_device('999-555-111')
    bc5.set_concurrent_devices(['666-567-777', '999-567-789'])
    bc5.load_data('sample_data.txt')
    bc5.prep_data()
    bc5.send_data_many(bc5.get_selected_devices())
```

Types of Inheritance

```
1. Single
     A
     |
     B

2. Hierarchical
     A
    / \
   B   C

3. Multiple
   A   B
    \ /
     C

4. Multi-level
    A
    |
    B
    |
    C

5. Hybrid

    A      A    A   B
   / \     |     \ /
  B   C    B      C
   \ /    / \     |
    D    C   D    D
   (a)    (b)    (c)
```

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

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

class B(A):
    def __init__(self):
        super().__init__()

b = B()
print(b.x)
```

In [5]:
```
class A(object):
    def __init__(self):
        self.x = 100

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

class B(A):
    '''
    def __init__(self):
        self.x = 100

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

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

class C(A):
    def __init__(self):
        self.x = 123.456

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

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

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 [6]:
```
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
```

Polymorphism

Single interface, multiple functionalities.
Polymorphism is, conditional and contextual executaion 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 [14]:
```
class A(object): # 1.0 2017
    def play(self):
        print (str(self.__class__) +  ' Can Play a sport')

class B(A): # 2.0 2018

    def swim(self):
        print (str(self.__class__) +  ' Can Swim in a pool')

class C(B): # 3.0 2019
    # overriding
    def play(self):
        print (str(self.__class__) +  ' Can Play a video game')
    def sing(self):
        print (str(self.__class__) +  ' Can Sing a song')
```

```
# User 2018
def action(x):
    x.play()

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

action(c)
```

```
<class '__main__.C'> Can Play a video game
<class '__main__.C'> Can Swim in a pool
<class '__main__.C'> Can Sing a song
```

In [ ]:

```

```

In [ ]:

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

Wi-Fi Technology Introduced

In [ ]:

```
from uuid import uuid1 as gen_uuid
get_ip_dhcp = lambda : '192.168.7.7'

class BlueTooth4Comm(object):
    def __init__(self, _header_format='@@@@BlueTooth4Comm@@@@',_range='2.4GHz
        self.UUID = gen_uuid()
        self.header = _header_format
        self.range = _range
        self.data= None
        self.mtu = _mtu
        self.packets = []
        self.connected_devices = []

    def load_data(self, file_loc):
        print(f"Loading data from {file_loc}")
        with open(file_loc, encoding='utf-8') as f:
            self.data = f.read()

    def register_device(self, device_id):
        self.connected_devices.append(device_id)
        print(f"Device {device_id} has been successfully registered")

    def unregister_device(self, device_id):
        self.connected_devices.remove(device_id)
        print(f"Device {device_id} has been successfully unregistered")

    def get_selected_device(self):
        return self.connected_devices[-1]

    def prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
```

```
                self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    def send_data(self, device):
        '''
        Sending data to selected device
        '''
        for seq, packet in enumerate(self.packets):
            print(f"Sending packet '{packet}' to {device}")



class BlueTooth5Comm(BlueTooth4Comm):

    def __init__(self, _header_format='#####BT5Header#####', _range='5GHz', _
        super().__init__(_header_format, _range, _mtu)
        self.concurrent_devices = []

    def set_concurrent_devices(self, devices=[]):
        self.concurrent_devices = devices

    def get_selected_devices(self):
        return self.concurrent_devices

    def send_data_many(self, devices=[]):
        for device in devices:
            super(BlueTooth5Comm, self).send_data()


class WiFiAcComm(object):
    def __init__(self, _header_format='%%%WiFiAcComm%%%',_range='5GHz', _mt
        self.mac_id = gen_uuid().hex
        self.ip_addr = get_ip_dhcp()
        self.header = _header_format
        self.range = _range
        self.data= None
        self.mtu = _mtu
        self.packets = []
        self.registered_networks = []

    def load_data(self, file_loc):
        print(f"Loading data from {file_loc}")
        with open(file_loc, encoding='utf-8') as f:
            self.data = f.read()

    def connect_network(self, net_id):
        self.registered_networks.append(net_id)
        print(f"Device {net_id} has been successfully registered")

    def forget_network(self, net_id):
        self.registered_networks.remove(net_id)
        print(f"Device {device_id} has been successfully unregistered")

    def get_selected_network(self):
        return self.registered_networks[-1]

    def prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
            self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    def send_data(self, addr):
        '''
        Sending data to selected device
        '''
        for seq, packet in enumerate(self.packets):
            print(f"Sending packet '{packet}' from {self.ip_addr} to {addr}")

    # Multi Casting
    def send_data_many(self, addrs=[]):
        for addr in addrs:
            self.send_data(addr)

if __name__ == '__main__':
    wc = WiFiAcComm();
    wc.connect_network('SSID-1234')
    wc.load_data('sample_data.txt')
    wc.prep_data()
    wc.send_data('192.168.1.109')
    wc.send_data_many(['192.168.1.101', '192.168.1.121', '192.168.1.118', '19
```

In [19]:

```
from uuid import uuid1 as gen_uuid
from abc import ABC, abstractmethod

class CommDevice(ABC):

    @abstractmethod
    def load_data(self, file):
        ...

    @abstractmethod
    def prep_data(self):
        pass

    @abstractmethod
    def send_data(self, target=[]):
        pass


class BlueTooth4Comm(CommDevice):

    def __init__(self, _header_format='@@@@BlueTooth4Comm@@@@',_range='2.4GHz

        self.UUID = gen_uuid()
        self.header = _header_format
        self.range = _range
        self.data= None
        self.mtu = _mtu
        self.packets = []
        self.connected_devices = []

    def load_data(self, file):
        print(f"Loading data from {file}")
        with open(file, encoding='utf-8') as f:
            self.data = f.read()

    def register_device(self, device_id):
        self.connected_devices.append(device_id)
        print(f"Device {device_id} has been successfully registered")

    def unregister_device(self, device_id):
        self.connected_devices.remove(device_id)
        print(f"Device {device_id} has been successfully unregistered")

    def get_selected_device(self):
        return self.connected_devices[-1]

    def prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
            self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    def send_data(self, target):
        '''
        Sending data to selected device
        '''
        for seq, packet in enumerate(self.packets):
            print(f"Sending packet '{packet}' to {target}")



class BlueTooth5Comm(BlueTooth4Comm):

    def __init__(self, _header_format='#####BT5Header#####', _range='5GHz', _
        super(BlueTooth5Comm, self).__init__(_header_format, _range, _mtu)
        self.concurrent_devices = []

    def set_concurrent_devices(self, devices=[]):
        self.concurrent_devices = devices

    def get_selected_devices(self):
        return self.concurrent_devices

    def send_data(self, target=[]):
        for device in target:
            super(BlueTooth5Comm, self).send_data(device)


class WiFiAcComm(CommDevice):
    mac_id = None
    ip_addr = None

    @staticmethod
    def set_ip_dhcp():
         WiFiAcComm.ip_addr = '192.168.7.7'

    @classmethod
    def set_mac_id(cls):
        WiFiAcComm.mac_id = gen_uuid().hex

    def __init__(self, _header_format='%%%%WiFiAcComm%%%%',_range='5GHz', _mt

        self.header = _header_format
        self.range = _range
        self.data= None
        self.mtu = _mtu
        self.packets = []
        self.registered_networks = []
        self.connect_network(_network)

    def load_data(self, file):
        print(f"Loading data from {file}")
        with open(file, encoding='utf-8') as f:
            self.data = f.read()

    def connect_network(self, net_id):
        self.registered_networks.append(net_id)
        print(f"Device {net_id} has been successfully registered")

    def forget_network(self, net_id):
        self.registered_networks.remove(net_id)
        print(f"Device {device_id} has been successfully unregistered")

    def get_selected_network(self):
        return self.registered_networks[-1]

    def prep_data(self):
        '''
        Splitting data into packets based on MTU
        '''
        from math import ceil
        packet_count = int(ceil(len(self.data)/self.mtu))
        print(f"{packet_count} packets are sent")

        for x in range(0, len(self.data), self.mtu):
            seq_no = "SEQNO: " + str(x//self.mtu) + " # "
            self.packets.append(self.header + seq_no + self.data[x:x+self.mtu

    # Multi Casting
    def send_data(self, target=[]):
        for addr in target:
            for seq, packet in enumerate(self.packets):
                print(f"Sending packet '{packet}' from {self.ip_addr} to {add


WiFiAcComm.set_mac_id()
WiFiAcComm.set_ip_dhcp()

def get_wireless_tech(tech='bt4'):

    devices = {
        'bt4' : BlueTooth4Comm(),
        'bt5' : BlueTooth5Comm(),
        'wifi': WiFiAcComm()
    }

    return devices[tech]

if __name__ == '__main__':

    #comm = get_wireless_tech(tech='bt4')
    comm = get_wireless_tech(tech='wifi')

    comm.load_data(file='sample_data.txt')
    comm.prep_data()
    comm.send_data(target='mydevice')
```

```
Device SSID-1234 has been successfully registered
Loading data from sample_data.txt
1 packets are sent
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not “different enough”. For example, in cases where yo
```

```
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with "names" (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to m
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not "different enough". For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with "names" (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to y
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not "different enough". For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with "names" (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to d
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not "different enough". For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with "names" (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to e
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not "different enough". For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with "names" (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to v
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
```

```
e-based UUID values are not “different enough”. For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with “names” (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to i
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not “different enough”. For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with “names” (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to c
Sending packet '%%%%WiFiAcComm%%%%SEQNO: 0 # Sometimes host-based and namespac
e-based UUID values are not “different enough”. For example, in cases where yo
u want to use the UUID as a lookup key, a more random sequence of values with
more differentiation is desirable to avoid collisions in a hash table. Having
values with fewer common digits also makes it easier to find them in log file
s. To add greater differentiation in your UUIDs, use uuid4() to generate them
using random input values.It is also useful in some contexts to create UUID va
lues from names instead of random or time-based values. Versions 3 and 5 of th
e UUID specification use cryptographic hash values (MD5 or SHA-1) to combine n
amespace-specific seed values with “names” (DNS hostnames, URLs, object ids, e
tc.). There are several well-known namespaces, identified by pre-defined UUID
values, for working with DNS, URLs, ISO OIDs, and X.500 Distinguished Names. Y
ou can also define your own application- specific namespaces by generating and
saving UUID values.' from 192.168.7.7 to e
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-6cb780a02130> in <module>
    161     comm.prep_data()
    162     comm.send_data(target='mydevice')
--> 163 UI

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

In [ ]:

```
UIApp
||
WirlessTech - Abstraction Interfaces
/     \    \     \
BT4  WiFI  WiDi  IR
|
BT5
```

In [ ]:

```
create_keyboard(keys):
    for k in keys:
        k.render_key()

class Key():
    def render_key(self):
        self.set_shape('rounded rectangle')
```

```
MyKey IS-A Key
class MyKey(Key):
    def render_key(self):
        self.set_shape('rectangle')

keys = []
for _ in range(12):
    # keys.append(Key())
    keys.append(MyKey())

create_keyboard(keys)
```

Without polymorphism:

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

In [ ]:

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

In [ ]:

```
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() for _ in  range(5)]
r = [Rectangle() for _ in range(7) ]
t = [Triangle() for _ in range(5)]

l = c + r + t

shuffle(l)

render_canvas(l)
```

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: Common Interface, allows the objects to change their behaviour independently without changing interfaces.

In [ ]:
```
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)
```

Use-Case 2: Incorporating changes into system

In [ ]:
```
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")

#client
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 = Rectangle()
r = RoundedRectangle()
t = Triangle()

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

render_canvas(l)
```

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 [ ]:

```
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 ('I"m an Idiot')

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)
```

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 2.7

In [ ]:

```
from abc import ABCMeta, abstractmethod

class Base(object):
    __metaclass__ = ABCMeta
    @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.bar()
```

In Python 3.6

In [24]:

```
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()
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-24-bbaabc31a6ca> in <module>
     18 
     19 
---> 20 d = Derived()
     21 d.foo()

TypeError: Can't instantiate abstract class Derived with abstract methods bar
```

We must override all abstract m,ethods, cannot leave them unimplemented.

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


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

Impleneting Shape classes using abc module

In [20]:

```
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-20-99814862d242> 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 methods disp
lay
```

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

Hexagon class must override display() method

In [ ]:

```
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)
```

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 whihc has mehtods or attributes prefixed with _(underscore)

In [13]:

```
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")

    # not for public use, will be disabled anytime soon
    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 [14]:

```
a.x
```

Out[14]:

```
222
```

In [15]:

```
a._y
```

Out[15]:

```
333
```

In [16]:

```
a.__z
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-965fa129e2df> in <module>
----> 1 a.__z

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

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

In [17]:

```
a.__dict__
```

Out[17]:

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

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

In [18]:
```
a._A__z
```

Out[18]:
```
555
```

Accessing private member functions

In [19]:
```
a.f1()
```

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

In [20]:
```
a._f2()
```

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

In [21]:
```
a.__f3()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-251ad2bdaabe> in <module>
----> 1 a.__f3()

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

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

In [22]:
```
A.__dict__
```

Out[22]:
```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.A.__init__(self)>,
              'f1': <function __main__.A.f1(self)>,
              '_f2': <function __main__.A._f2(self)>,
              '_A__f3': <function __main__.A.__f3(self)>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '__doc__': None})
```

In [23]:
```
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 [ ]:
```
def f(self, eid, name):
    self.empId = eid
    self.name = name

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

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 [ ]:
```
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)
print ('Object count: ', A.obj_count)
a4 = A(50, 60, 70)

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

In [ ]:
```
a1.fun()
a2.fun()
```

In [ ]:
```
a1.getDBConnection() # not recomeded, Pls donot do this
```

In [ ]:
```
a1.obj_count
```

In [ ]:
```
a2.obj_count
```

In [ ]:
```
A.obj_count
```

In [ ]:
```
a1.obj_count = 10
```

In [ ]:
```
print (a1.obj_count, a2.obj_count, A.obj_count)
```

In [ ]:
```
a1.__dict__
```

In [ ]:
```
A.obj_count
```

In [ ]:
```
A.__dict__
```

class method : if we need to use class attributes

In [ ]:
```
## 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 [ ]:
```
A.__dict__
```

In [ ]:
```
A.getDBConnection() # static method
A.getLogger() # class method
```

In [ ]:
```
A.dbConn # static variable
```

In [ ]:
```
a = A(2, 3, 4)
print (a.__dict__)
```

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

print(A.objectCount)
```

In [ ]:
```
class A(object):
    instance = None

    @classmethod
    def get_instance(cls):
        if not cls.instance:
            print("Inside", cls)
            cls.instance = cls()
        return cls.instance

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

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

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

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

Funcion Objects (Functor), Callable objects

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

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

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

In [ ]:
```
a = Sqr(20)
```

In [ ]:
```
print(a.sqr())
```

In [ ]:
```
a()
```

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

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

In [ ]:
```
s = Sqr(40)
s()
```

In [ ]:
```
s.__call__()
```

Multiple family of classes:

In [ ]:
```
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)
```

In [ ]:
```
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)
```

Decorator and Context manager

In [ ]:

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

In [ ]:

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

In [ ]:

```
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)
```

In [ ]:

```
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)
```

In [ ]:

```
timeit(fun)
```

Function Overloading

In [ ]:

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

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


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

In [ ]:

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



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

- 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 [ ]:
```
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)
e1.printData()
```

In [ ]:
```
print(e1)
```

Above statement is equal to

In [ ]:
```
print (str(e1)) # str(e1) is equal to e1.__str__()
```

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

In [ ]:
```
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 f'EmpId: {self.empNum}, EmpName: {self.empName}, EmpSalary: {s


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

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

In [ ]:
```
e1
```

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 [ ]:
```
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 f'EmpId: {self.empNum}, EmpName: {self.empName}, EmpSalary: {s

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

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

In [ ]:
```
print (e1) # invokes e1.__str__() or str(e1)
```

In [ ]:
```
e1 # invokes e1.__repr__() or repr(e1)
```

In [ ]:
```
l = [2, 3, 4, 5]
d = {2: "Two", 1:"One"}
l
```

Difference between above two printing statements is

In [ ]:
```
e1 # repr(e1) ==> e1.__repr__()
```

In [ ]:
```
repr(e1)
```

In [ ]:
```
e1.__repr__()
```

eval() fiunction

Executes string as code

In [ ]:
```
eval('20 + 30')
```

In [ ]:
```
x = 20
y = 40
eval('x*y', globals(), locals())
```

In [ ]:
```
obj = eval(repr(e1))
```

In [ ]:
```
id(e1), id(obj)
```

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 [ ]:
```
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({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 [ ]:
```
e2 < e3
```

In [ ]:
```
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 [ ]:

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

In [ ]:

```
e2 + e3
```

In [ ]:

```
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 [ ]:

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

In [ ]:

```
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 [ ]:

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

In [ ]:

```
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 [ ]:

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

In [ ]:

```
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 [ ]:

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

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 [ ]:
```
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 [ ]:
```
# 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
```

Explicitly providing creteria

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

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

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

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

Function Overloading

In [ ]:

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

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

In [ ]:

```
a = A()
```

In [ ]:

```
a.fun()
```

In [ ]:

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

In [ ]:

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

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

a = A()
```

In [ ]:

```
a.fun()
```

Note: Function overloading is not possible in python

In [ ]: