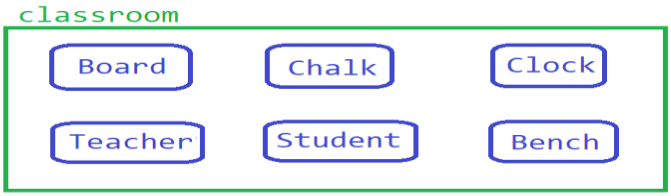
Object Oriented Programming

OOP is a programming paradigm that uses techniques like:
* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

for application development.

Encapsulation

Encapsulation is the OOP technique that divides the application into a collection of entities.



An OOP program organizes its code and data, entitywise.

Encapsulation demands development of a class to represent an entity.
Each entity (class) has
code (methods: to represent operations)
and
data (variables, lists, ...: to represent attributes).

```
Clock
  attributes: hours, minutes, seconds
  operations: set_time(), display_time(), update_time()
```

```
class Clock:
  def __init__(self):
    self.hours = val          #attributes(data)
    self.minutes = val
    self.seconds = val
  def set_time(self):         #operations (methods)
    ...
  def display_time(self):
    ...
  def update_time(self):
    ...
```

In general terms, *Encapsulation binds the code and data of an entity into one unit (class).*

Using a class

A class is used in two ways:
* By instantiation
* By inheritance

Instantiation

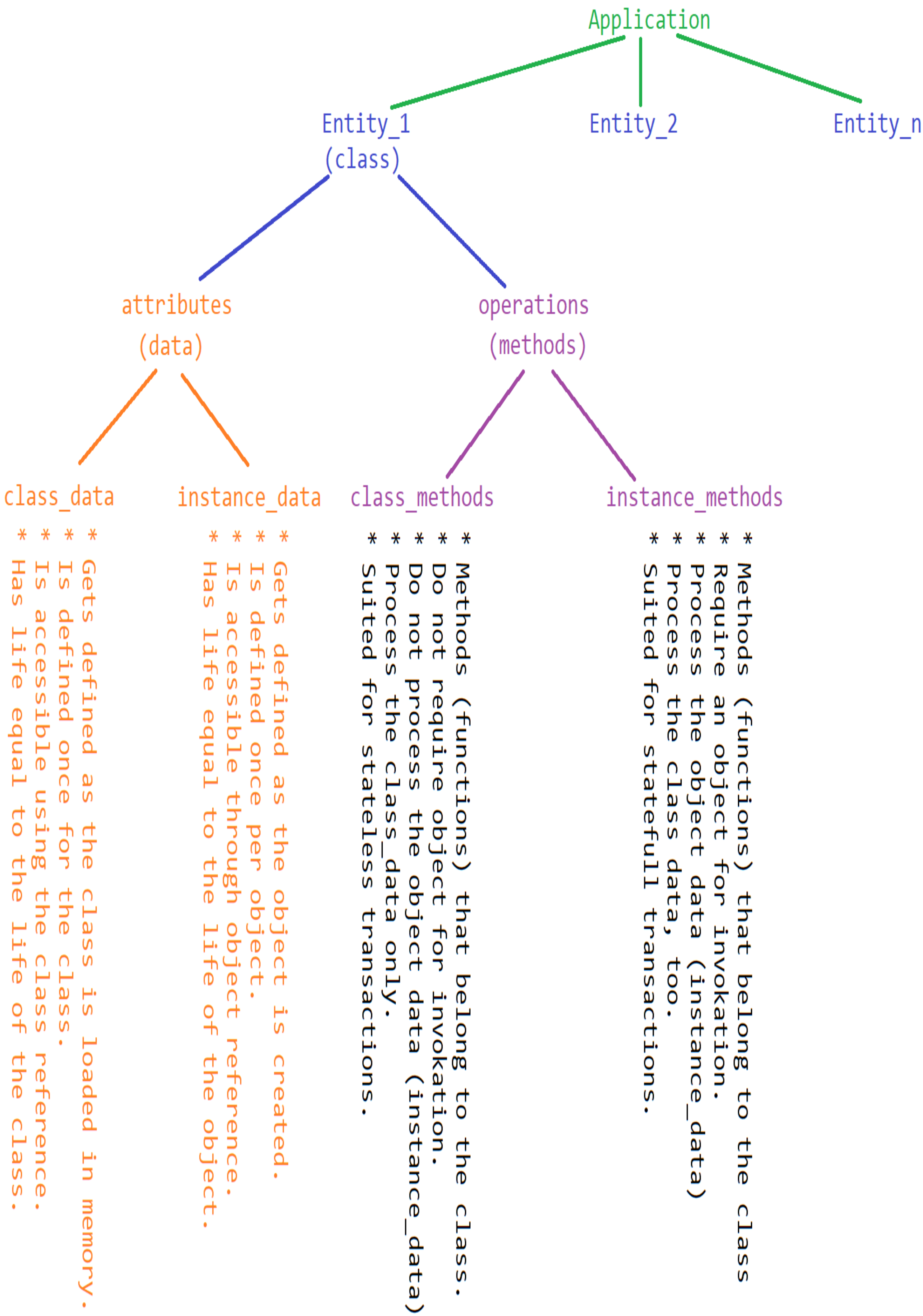
Instantiation means making objects of the class that can store and process data.
One object represents one state of an entity.
Example to represent IndiaTime, LondonTime and NewYorkTime the application requires 3 objects.

```
i_time = Clock()
l_time = Clock()
ny_time = Clock()
```

Inheritance

Inheritance is definition of a derived entity that extends the attributes and operations of a base entity.

```
class Clock:
  attributes: hours, minutes, seconds
  operations: set_time(), display_time(), update_time()

class Alarm_Clock(Clock):
  extended attributes: alarm_hours, alarm_minutes, alarm_tone, alarm_volume
  extends operations: set_alarm(), play_alarm(), snooze()
```


Application
Entity_1
(class)
Entity_2
Entity_n
attributes
(data)
operations
(methods)
class_data
instance_data
class_methods
instance_methods
* Gets defined as the class is loaded in memory.
* Is defined once for the class.
* Is accessible using the class reference.
* Has life equal to the life of the class.
* Gets defined as the object is created.
* Is defined once per object.
* Is accessible through object reference.
* Has life equal to the life of the object.
* Methods (functions) that belong to the class.
* Do not require object for invokation.
* Do not process the object data (instance_data)
* Process the class_data only.
* Suited for stateless transactions.
* Methods (functions) that belong to the class
* Require an object for invokation.
* Process the object data (instance_data)
* Process the class data, too.
* Suited for statefull transactions.

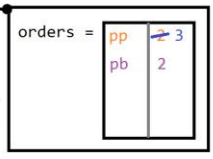
```
#class methods, class_data
#instance methods, instance_data

class Restaurant:
    dishes = {'wada_pav':15, 'cold_drink':25, 'packed_meal':100}

    @classmethod
    def take_away_order(cls, dish):
        if dish in cls.dishes:
            print('Selling:', dish)
            print('Bill Amount:', cls.dishes[dish])
            print('Avoid wastage of food')
        else:
            print(dish, ' not available as a take away order')


    #instance_methods ahead

    #self:
    #self is the first formal parameter for instance methods.
    #It is a reference that is initiailized with the memory location
    #of the object used to invoke an instance_method.
    #It allows access of the objects memory from inside the method.

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

    def order(self, dish, qty=1):
        if dish in self.orders:
            self.orders[dish] += qty
        else:
            self.orders[dish] = qty

    def bill(self):
        sum =0
        i = 1
        for x in self.orders:
            sum += self.orders[x]
            print(i,') ', x, ', Qty: ', self.orders[x], ', Amt: ', self.orders[x])
            i+=1
        print('Total Bill: ', sum)
        print('Thank You, Visit Again')
        print('Avoid wastage of food')

def main():
    print('1. Take Away ')
    print('2. Dine In')
    print('Enter Choice ')
    ch = int(input())

    if ch == 1:
        x = input('Order the dish ')
        Restaurant.take_away_order(x)
    elif ch == 2:
        client1 = Restaurant()
        client1.order('Pani Puri', 2)# order(client1, 'Pani Puri', 2)
        client1.order('Pav Bhaji', 2)# order(client1, 'Pav Bhaji', 2)
        client1.order('Pani Puri', 1)# order(client1, 'Pani Puri', 1)
        client1.bill() #bill(client1)
        client2 = Restaurant()
        client2.order('Poha', 2)  # order(client2, 'Poha', 2)
        client2.bill()  # bill(client2)

main()
```

Case: **Take Away**

Method call: Restaurant.take_away_order(x)
Program control jumps to the method with data as parameter.
Method acts on parameter, class_data
Method returns a value (optional)

Case: **Dine In**

Object created.
ref



object.method is called with params.

Program control jumps to the method with object reference and data as parameters.

In the instance_method the first formal parameter (self) is initialized with the object's memory location.
(So actions on self are actions on the object).

Method acts on parameters, object data (via self) and class data, too.
Method returns a value (optional)

OOP - Inheritance

OOP technique inheritance allows definition of derived entities (classes) from the existing ones.
The derived class (sub class) gets:
* *Code*
* *Data*
* *Compatibility*

from the base classes (super classes).
The derived class is a *host for extended code and data.*

Compatibility by inheritance *allows* the usage of object of a sub class in place where the object of the super class is *expected*.

Python allows multiple types of inheritances:

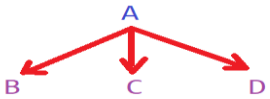
* Inheritance

  One class is derived from another.


```
class A:
    pass
class B(A):
    pass
```

* Hierarchical Inheritance

  Multiple classes are derived from a common super class.




```
class A:
    pass
class B(A):
    pass
class C(A):
    pass
class D(A):
    pass
```

* Multilevel Inheritance

  A sub class is used as a super class for deriving more sub classes.


```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
```

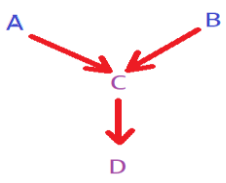
* Multiple Inheritance

  One class is derived from multiple super classes.

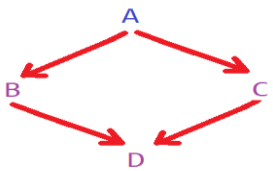

```
class A:
    pass
class B:
    pass
class C(A,B):
    pass
```

* Hybrid Inheritance

  Is the combinition of multiple and multilevel inheritance.




```
class A:
    pass
class B:
    pass
class C(A,B):
    pass
class D(C):
    pass
```




```
class A:
    pass
class B(A):
    pass
class C(A):
    pass
class D(B,C):
    pass
```

OOP-Polymorphism

Polymorpshism allows an application to have multiple implementations of an operation.

Say the operation is move(), across following hierarchy of classes:

```
Animal
  move()
    abstract

Fish(Animal)
  move()
    swimming

Bird(Animal)
  move()
    flying
```

Here move() is the method that is primarily inherited by the sub class but is redefined (overridden) to suit the sub classes behaviours.

For an algorithm (say a race) in which a super (Animal) type is allowed, type compatiblity would support usage of objects of sub types (Fish and Bird) too.

```
def race(animalRef):
 if isinstance(animalRef, Animal):
   on your marks
   get set
   while not finished:
      animalRef.move()
   decide the winner
```

```
f = Fish()

b = Bird()
```

When the animalRef is initialized using Fish object then the call to the polymorphic method (move()) makes Python execute (bind to) the defintion of move() provided by Fish (type of initializing object).

Similarly when the animalRef is initialized using Bird object then the call to the polymorphic method (move()) makes Python execute (bind to) the defintion of move() provided by Bird (type of initializing object).

Finally said, to implement polymorphism in Python.

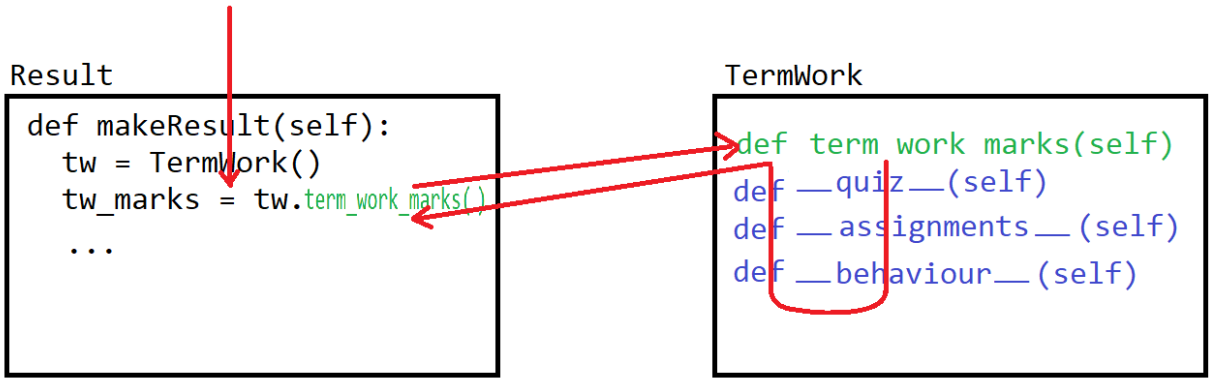
* Hierarchy of classes is defined using inheritance.
* Polymorpshic method is overridden by each class to suit respective behaviours.
* An algorithm that acts on super class type is provided the objects of sub class types. On call to the polymorphic method, Python identifies the type of sub class object and dynamically binds the call with definition provided by it (identified sub class type).
  Thus achieving multiple definitions for an operation.

Abstraction

Abstraction is the program designing element of OOP, in which an object exposes limited public attributes and behaviours while hiding the rest of the implementations.

Example:

```
class TermWork:
  def __quiz__(self):  #private (for interal use)
    pass
  def __assignments__(self): #private (for interal use)
    pass
  def __behaviour__(self): #private (for interal use)
    pass

  def term_work_marks(self): #public (for calls outside the class)
     q_score = self.__quiz__()
     a_score = self.__assignments__()
     b_score = self.__behaviour__()
     #...
     tw = q_score + a_score + b_score
```



For data abstraction, the data members are made:
- private (for use inside the class)
- protected (for use inside the class and sub classes)
- public (for use across the application)

For code abstraction, the task is divided into sub tasks. The sub tasks methods are made private or protected, while the method that represents the task is made public.

Python doesnt have private, protected or public keywords for limiting the access of a member of the class.

A naming convention in which:

> private members are prefixed and suffixed with two underscores,
> protected members are prefixed and suffixed with one underscore and
> public members are not applied any prefix and suffix

is suggestive of access.