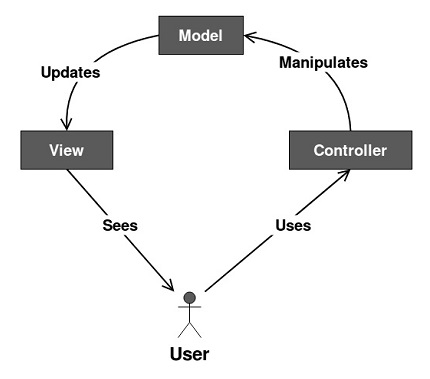
Python support variety of design patterns. The syntax of python is easy to understand and uses English keywords. Python helps in constituting a design pattern using the following parameters –

* Pattern Name
* Intent
* Aliases
* Problem
* Solution
* Structure
* Participants
* Constraints
* Sample Code

Some of the design patterns are:

* **Model View Controller Pattern (MVC)**

It is the most commonly used design pattern and it is easy to design.



**Model**:

It consists of pure application logic which interacts with the database. It includes all the information to represent data to the end user.

**View**:

View represents the HTML files, which interact with the end user.

**Controller**:

It acts as an intermediary between view and model.

**Program:**

Model.py:

import json

class Person(object):

def \_\_init\_\_(self, first\_name = None, last\_name = None):

self.first\_name = first\_name

self.last\_name = last\_name

#returns Person name, ex: John Doe

def name(self):

return ("%s %s" % (self. first\_name,self.last\_name))

@classmethod

#returns all people inside db.txt as list of Person objects

def getAll(self):

database = open ('db.txt', 'r')

result = []

json\_list = json.loads(database.read())

for item in json\_list:

item = json.loads(item)

person = Person(item['first\_name'], item['last\_name'])

result.append(person)

return result

View.py:

from model import Person

def showAllView(list):

print 'In our db we have %i users. Here they are:' % len(list)

for item in list:

print item.name()

def startView ():

print 'Do you want to see everyone in my db? [y/n]'

def endView():

print 'Goodbye!'

from model import Person

import view

def showAll():

#gets list of all Person objects

people\_in\_db = Person.getAll()

#calls view

return view.showAllView(people\_in\_db)

def start():

view.startView()

input = raw\_input()

if input == 'y':

return showAll()

else:

return view.endView()

if \_\_name\_\_ == "\_\_main\_\_":

#running controller function

start()

1. Behavioral Design Pattern:

* Behavioral patterns are all about identifying the common communication patterns between objects and realize these patterns.
* These patterns are concerned with algorithms and the assignment of responsibilities between objects.
* Classification of Behavioral Patterns:

Iterator Method:

* Iterator method is a [**Behavioral Design Pattern**](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/) that allows us to traverse the elements of the collections without taking the exposure of in-depth details of the elements.
* It provides a way to access the elements of complex data structure sequentially without repeating them.
* It’s always handy for Python users to use [**Iterators**](https://www.geeksforgeeks.org/iterators-in-python/) for traversing any kind of data structure doesn’t matter they are linear or no-linear data structures.

**Code:**

def alphabets\_upto(letter):

    """Counts by word numbers, up to a maximum of five"""

    for i in range(65, ord(letter)+1): #ord(): Coverts character into its Unicode code

            yield chr(i)

#( yield() - used to return from a function without destroying the states of its local variable and when the function is called, the execution starts from the last **yield** statement.

"""main method"""

if \_\_name\_\_ == "\_\_main\_\_":

    alphabets\_upto\_K = alphabets\_upto('K')

    alphabets\_upto\_M = alphabets\_upto('M')

    for alpha in alphabets\_upto\_K:

        print(alpha, end=" ")

    print()

    for alpha in alphabets\_upto\_M:

        print(alpha, end=" ")

Mediator Method:

* allows us to reduce the unordered dependencies between the objects.
* objects take the help of mediator objects to communicate with each other.

**Code:**

class Course(object):

    """Mediator class."""

    def displayCourse(self, user, course\_name):

        print("[{}'s course ]: {}".format(user, course\_name))

class User(object):

    '''A class whose instances want to interact with each other.'''

    def \_\_init\_\_(self, name):

        self.name = name

        self.course = Course()

    def sendCourse(self, course\_name):

        self.course.displayCourse(self, course\_name)

    def \_\_str\_\_(self):

        return self.name

"""main method"""

if \_\_name\_\_ == "\_\_main\_\_":

    mayank = User('Mayank')   # user object

    lakshya = User('Lakshya') # user object

    krishna = User('Krishna') # user object

    mayank.sendCourse("Data Structures and Algorithms")

    lakshya.sendCourse("Software Development Engineer")

    krishna.sendCourse("Standard Template Library")

1. Creational Design Pattern

* provides essential information regarding the Class instantiation or the object instantiation.
* Two types: Class Creational pattern and object creational pattern.
* class-creation patterns use inheritance.
* Object-creation patterns use delegation.
* They increase the system's flexibility in terms of the what, who, how, and when of object creation.
* The classification of creational design patterns is:
* Factory Method, Abstract Factory Method, Builder Method, Singleton Method.

Factory method:

* Factory Method is a Creational Design Pattern that allows an interface or a class to create an object, but let subclasses decide which class or object to instantiate.
* We can easily add the new types of products without disturbing the existing client code.
* Disadvantages: You end up with huge number of small files

**Code:**

class FrenchLocalizer:

def \_\_init\_\_(self):

self. translations = {"car": "voiture", "bike": "bicyclette",

"cycle":"cyclette"}

def localize (self, message):

"""change the message using translations"""

return self.translations.get(msg, msg)

class SpanishLocalizer:

def \_\_init\_\_(self):

self.translations = {"car": "coche", "bike": "bicicleta",

"cycle":"ciclo"}

def localize(self, msg):

"""change the message using translations"""

return self.translations.get (msg, msg)

class EnglishLocalizer:

"""Simply return the same message"""

def localize(self, msg):

return msg

def Factory(language ="English"):

"""Factory Method"""

localizers = {

"French": FrenchLocalizer,

"English": EnglishLocalizer,

"Spanish": SpanishLocalizer,

}

return localizers[language]()

if \_\_name\_\_ == "\_\_main\_\_":

f = Factory("French")

e = Factory("English")

s = Factory("Spanish")

message = ["car", "bike", "cycle"]

for msg in message:

print(e.localize(msg))

print("In French", f.localize(msg))

print("In Spanish", s.localize(msg))

print("..............")

***Builder Method:***

* It allows you to construct complex objects step by step.
* They instantiate new objects at run time.

Code:

class Director:

\_\_builder = None

def setBuilder(self, builder):

self.\_\_builder = builder

def getCar(self):

car = Car()

# First goes the body

body = self.\_\_builder.getBody()

car.setBody(body)

# Then engine

engine = self.\_\_builder.getEngine()

car.setEngine(engine)

# And four wheels

i = 0

while i < 4:

wheel = self.\_\_builder.getWheel()

i += 1

return car

# The whole product

class Car:

def \_\_init\_\_(self):

self.\_\_wheels = list()

self.\_\_engine = None

self.\_\_body = None

def setBody(self, body):

self.\_\_body = body

def attachWheel(self, wheel):

self.\_\_wheels.append(wheel)

def setEngine(self, engine):

self.\_\_engine = engine

def specification(self):

print ("body: %s" % self.\_\_body.shape)

print ("engine horsepower: %d" % self.\_\_engine.horsepower)

print ("tire size: %d\'" % self.\_\_wheels[0].size)

class Builder:

def getWheel(self): pass

def getEngine(self): pass

def getBody(self): pass

class JeepBuilder(Builder):

def getWheel(self):

wheel = Wheel()

wheel.size = 22

return wheel

def getEngine(self):

engine = Engine()

engine.horsepower = 400

return engine

def getBody(self):

body = Body()

body.shape = "SUV"

return body

# Car parts

class Wheel:

size = None

class Engine:

horsepower = None

class Body:

shape = None

def main():

jeepBuilder = JeepBuilder() # initializing the class

director = Director()

# Build Jeep

print ("Jeep")

director.setBuilder(jeepBuilder)

jeep = director.getCar()

jeep.specification()

print ("")

if \_\_name\_\_ == "\_\_main\_\_":

main()

***Prototype Method:***

* It aims to reduce the number of classes used for an application.
* It is highly recommended to use **Prototype Method** when the object creation is an expensive task in terms of time and usage of resources and already there exists a similar object.
* E.g. Shape class – exact copy
* Without prototype method you should dependent on the code of other class which is certainly not a good practice in Software Development.
* Less number of sub classes. – advantages.
* allows cloning objects, even complex ones, without coupling to their specific classes.

1. Structural Design Patterns:

Structural Design Patterns are used to assemble multiple classes into bigger working structures.

Types:

Adapter Method:

* Adapting one input to a different predetermined output.

**Code**:

class MotorCycle:

    """Class for MotorCycle"""

    def \_\_init\_\_(self):

        self.name = "MotorCycle"

    def TwoWheeler(self):

        return "TwoWheeler"

class Truck:

    """Class for Truck"""

    def \_\_init\_\_(self):

        self.name = "Truck"

    def EightWheeler(self):

        return "EightWheeler"

class Car:

    """Class for Car"""

    def \_\_init\_\_(self):

        self.name = "Car"

    def FourWheeler(self):

        return "FourWheeler"

class Adapter:

    def \_\_init\_\_(self, obj, \*\*adapted\_methods):

        """We set the adapted methods in the object's dict"""

        self.obj = obj

        self. \_\_dict\_\_.update(adapted\_methods)

    def \_\_getattr\_\_(self, attr):

        """All non-adapted calls are passed to the object"""

        return getattr(self.obj, attr)

    def original\_dict(self):

        """Print original object dict"""

        return self.obj.\_\_dict\_\_

""" main method """

if \_\_name\_\_ == "\_\_main\_\_":

    """list to store objects"""

    objects = []

    motorCycle = MotorCycle ()

    objects. append (Adapter (motorCycle, wheels = motorCycle.TwoWheeler))

    truck = Truck ()

    objects.append(Adapter(truck, wheels = truck.EightWheeler))

    car = Car()

    objects. append(Adapter(car, wheels = car.FourWheeler))

    for obj in objects:

       print("A {0} is a {1} vehicle”. Format(obj.name, obj.wheels()))

Decorator Method:

* allows you to dynamically attach new behaviors to objects without changing their implementation.
* It is much easier to implement **Decorator Method** in Python because of its built-in feature.
* It is not equivalent to the **Inheritance** because the new feature is added only to that particular object, not to the entire subclass.