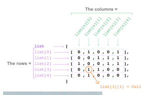
Design Patterns in Python — Factory

medium.com/@hardikpatel_6314/design-patterns-in-python-factory-c728b88603eb

April 19, 2020



Factory is a creational design pattern which helps hiding object creation process.

Pattern Type — Creational

Benefits -

- Object creation can be independent of the class implementation.
- Adding support of new type of object is very easy
- Logic of object creation is hidden

Type of patterns–

- : This design allows to create objects without exposing the object creation logic.
- : This design allows to create objects, but defers the decision to the subclasses to determine the class for object creation.
- : An Abstract Factory is an interface to create related objects without specifying/exposing their classes. The pattern provides objects of another factory, which internally creates other objects.

Simple Factory Pattern -

Basically, it's prerequisite pattern to implement next two patterns, so let's look at the implementation of Simple factory pattern.

```
"""
Learn how to create simple factory which helps to hide
logic of creating objects.
"""


from abc import ABCMeta, abstractmethod


class Person(metaclass=ABCMeta):
    @abstractmethod
    def create(self):
        pass


class HR(Person):
    def create(self, name):
        print(f"HR {name} is created")


class Engineer(Person):
    def create(self, name):
        print(f"Engineer {name} is created")


class PersonFactory(object):
    @classmethod
    def createPerson(cls, designation, name):
        eval(designation)().create(name)

if __name__ == "__main__":    designation = input("Please enter the designation - ")    name =
input("Please enter the person's name - ")    PersonFactory.createPerson(designation, name)
```

Person is abstract concrete class, **HR/Engineer** are concrete class for individual designation and **PersonFactory** is factory class which implements create method to create different objects of different designation using concrete classes.

In this example, You have hidden the logic of creating Person by **PersonFactory(Interface)** which requires the Person designation and name to create them. In future if you want to add new designation you just need to add one class for that by overriding **Person(Concrete)** class and you will be able to create person with that new designation by the same factory.

Factory method pattern -

It provides lot of flexibility and also makes code very generic as not tied to a certain class for object creation. So, we are dependent on the interface not the concrete class as we see in **Simple factory pattern.** As the code that creates the object is separate from the code that uses it, the client doesn’t need to bother about what argument to pass and which class to instantiate. The addition of new classes is easy and involves low maintenance.

Let’s understand factory method pattern by an example -

```
"""
Learn how to create simple factory which helps to hide
logic of creating objects.
"""


from abc import ABCMeta, abstractmethod


class AbstractDegree(metaclass=ABCMeta):
    @abstractmethod
    def info(self):
        pass


class BE(AbstractDegree):
    def info(self):
        print("Bachelor of engineering")


    def __str__(self):
        return "Bachelor of engineering"


class ME(AbstractDegree):
    def info(self):
        print("Master of engineering")


    def __str__(self):
        return "Master of engineering"


class MBA(AbstractDegree):
    def info(self):
        print("Master of business administration")


    def __str__(self):
        return "Master of business administration"
```

```
class ProfileAbstractFactory(object):
    def __init__(self):
        self._degrees = []



    @abstractmethod
    def createProfile(self):
        pass



    def addDegree(self, degree):
        self._degrees.append(degree)



    def getDegrees(self):
        return self._degrees



class ManagerFactory(ProfileAbstractFactory):
    def createProfile(self):
        self.addDegree(BE())
        self.addDegree(MBA())
        return self._degrees



class EngineerFactory(ProfileAbstractFactory):
    def createProfile(self):
        self.addDegree(BE())
        self.addDegree(ME())
        return self._degrees



class ProfileFactory(object):
    def getProfile(self, factory):
        return factory.createProfile()

    if __name__ == "__main__":    pf = ProfileFactory().getProfile(ManagerFactory())    print(pf)
```

Here, In above example, **Degree** is an abstract concrete class and **BE, ME** and **MBA** are concrete classes. **ProfileAbstractFactory** is an abstract factory which has one declared methods **createProfile** which must be implemented by individual profile factory and **addDegrees**, **getDegrees** instance methods to help creating profile of different designations. **EngineerFactory, ManagerFactory** are factories which has implemented **createProfile** method as both can have different degrees. Now **ProfileFactory,** final factory which creates profile which you want by passing instance of **EngineerFactory** or other related.

Now if you want to have support of having HR profile then just need to implement **HRFactory** and you have requirement to add one more **Degree** then you just need to one more concrete class for that.

Abstract Factory Pattern -

This patterns goes beyond factory method pattern and it includes multiple factories. It provides facility to encapsulate group of individual factory of same theme without specifying concrete class.

Let's take an example of Car factory-

```
"""
Provide an interface for creating families of related or dependent
objects without specifying their concrete classes.
"""



from abc import ABCMeta, abstractmethod



class CarFactory(metaclass=ABCMeta):



    @abstractmethod
    def build_parts(self):
        pass



    @abstractmethod
    def build_car(self):
        pass



class SedanCarFactory(CarFactory):



    def build_parts(self):
        return SedanCarParts()



    def build_car(self):
        return SedanCar()



class SUVCarFactory(CarFactory):
    def build_parts(self):
        return SUVCarParts()


    def build_car(self):
        return SUVCar()


class CarParts(metaclass=ABCMeta):
    """
    Declare an interface for a type of car parts.
    """


    @abstractmethod
    def build(self):
        pass


class SedanCarParts(CarParts):


    def build(self):
        print("Sedan car parts are built")


class SUVCarParts(CarParts):


    def build(self):
        print("SUV Car parts are built")


class AbstractCar(metaclass=ABCMeta):
    """
    Declare an interface for a type of cars.
    """


    @abstractmethod
    def assemble(self):
        pass


class SedanCar(AbstractCar):


    def assemble(self, parts):
        print(f"Sedan car is assembled here using {parts}")
        print(f"Sedan car is assembled here using {parts}")


class SUVCar(AbstractCar):

    def assemble(self, parts):        print(f"SUV car is assembled here using {parts}")if __name__
== "__main__":    type = "sedan"    if type == "sedan":        factory = SedanCarFactory()    elif
type == "suv":        factory = SUVCarFactory()    else:        raise Exception(f"Not implemented
{type}")    car_parts = factory.build_parts()    car_builder = factory.build_car()
car_parts.build()    car_builder.assemble(car_parts)
```

In above example, you want two kind of cars manufactured and to do that you will create two different factory which requires two processes, first build the parts, second assemble the car. both the processes will be done by other factories. So We have created one abstract car factory which needs two processes as abstract methods which must be implemented by **Actual Factories**. For example, **SedanCarFactory** and **SUVCarFactory** are factories which have implemented both methods **build_parts** and **build_car** and both use other factories. Now to build parts of **SUV car**, **SUVCarParts** factory is being used to build parts required in **SUV** car and same can happen for **SedanCar.**

Now parts are readyand further to **build_car** using built parts by **SUVCarParts**, **SUVCar** factory is being used which assembe all the parts and then returns the **Car.** Now this whole process user doesn't need to know how it actually build those things.

In the last in example, according to type Car will be created and returned.

I would not recommend this last pattern for normal use until you have problem which has very much common attributes and many different types. That's all I have to say.

Thanks for reading this article. If you have any comments or questions please let me know in comment section.

Originally published at aaravtech.com.