**Factory Pattern in Python**

factory pattern examples and use cases in python

The \*\*Factory Pattern\*\* is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It is commonly used when the exact type of the object to be created is determined at runtime.  
  
Below are examples and use cases of the Factory Pattern in Python.  
  
---  
  
## \*\*Example 1: Simple Factory Pattern\*\*  
  
In this example, a factory method is used to create objects of different types based on input.  
  
```python  
class Dog:  
 def speak(self):  
 return "Woof!"  
  
class Cat:  
 def speak(self):  
 return "Meow!"  
  
class AnimalFactory:  
 @staticmethod  
 def get\_animal(animal\_type):  
 if animal\_type == "dog":  
 return Dog()  
 elif animal\_type == "cat":  
 return Cat()  
 else:  
 raise ValueError("Unknown animal type")  
  
# Usage  
animal = AnimalFactory.get\_animal("dog")  
print(animal.speak()) # Output: Woof!  
  
animal = AnimalFactory.get\_animal("cat")  
print(animal.speak()) # Output: Meow!  
```  
  
### \*\*Use Case\*\*  
- When you need to create objects without specifying their exact class.  
- Example: A GUI framework where buttons, text boxes, and other widgets are created dynamically based on the platform (Windows, macOS, etc.).  
  
---  
  
## \*\*Example 2: Factory Method Pattern\*\*  
  
The Factory Method Pattern allows subclasses to decide which class to instantiate. This is achieved by defining a method in the base class and letting subclasses override it.  
  
```python  
from abc import ABC, abstractmethod  
  
class Animal(ABC):  
 @abstractmethod  
 def speak(self):  
 pass  
  
class Dog(Animal):  
 def speak(self):  
 return "Woof!"  
  
class Cat(Animal):  
 def speak(self):  
 return "Meow!"  
  
class AnimalFactory(ABC):  
 @abstractmethod  
 def create\_animal(self):  
 pass  
  
class DogFactory(AnimalFactory):  
 def create\_animal(self):  
 return Dog()  
  
class CatFactory(AnimalFactory):  
 def create\_animal(self):  
 return Cat()  
  
# Usage  
dog\_factory = DogFactory()  
dog = dog\_factory.create\_animal()  
print(dog.speak()) # Output: Woof!  
  
cat\_factory = CatFactory()  
cat = cat\_factory.create\_animal()  
print(cat.speak()) # Output: Meow!  
```  
  
### \*\*Use Case\*\*  
- When a class delegates the responsibility of instantiating objects to its subclasses.  
- Example: A game where different types of enemies (e.g., zombies, aliens) are created dynamically based on the level.  
  
---  
  
## \*\*Example 3: Abstract Factory Pattern\*\*  
  
The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.  
  
```python  
from abc import ABC, abstractmethod  
  
# Abstract Products  
class Chair(ABC):  
 @abstractmethod  
 def sit\_on(self):  
 pass  
  
class Sofa(ABC):  
 @abstractmethod  
 def lie\_on(self):  
 pass  
  
# Concrete Products  
class ModernChair(Chair):  
 def sit\_on(self):  
 return "Sitting on a modern chair."  
  
class VictorianChair(Chair):  
 def sit\_on(self):  
 return "Sitting on a Victorian chair."  
  
class ModernSofa(Sofa):  
 def lie\_on(self):  
 return "Lying on a modern sofa."  
  
class VictorianSofa(Sofa):  
 def lie\_on(self):  
 return "Lying on a Victorian sofa."  
  
# Abstract Factory  
class FurnitureFactory(ABC):  
 @abstractmethod  
 def create\_chair(self):  
 pass  
  
 @abstractmethod  
 def create\_sofa(self):  
 pass  
  
# Concrete Factories  
class ModernFurnitureFactory(FurnitureFactory):  
 def create\_chair(self):  
 return ModernChair()  
  
 def create\_sofa(self):  
 return ModernSofa()  
  
class VictorianFurnitureFactory(FurnitureFactory):  
 def create\_chair(self):  
 return VictorianChair()  
  
 def create\_sofa(self):  
 return VictorianSofa()  
  
# Usage  
def get\_furniture(factory: FurnitureFactory):  
 chair = factory.create\_chair()  
 sofa = factory.create\_sofa()  
 print(chair.sit\_on())  
 print(sofa.lie\_on())  
  
modern\_factory = ModernFurnitureFactory()  
victorian\_factory = VictorianFurnitureFactory()  
  
print("Modern Furniture:")  
get\_furniture(modern\_factory)  
  
print("\nVictorian Furniture:")  
get\_furniture(victorian\_factory)  
```  
  
### \*\*Use Case\*\*  
- When you need to create families of related objects (e.g., chairs and sofas) that must be used together.  
- Example: A theme-based UI toolkit where widgets (buttons, scrollbars, etc.) are created based on the selected theme (dark mode, light mode).  
  
---  
  
## \*\*Advantages of Factory Pattern\*\*  
1. \*\*Encapsulation\*\*: The creation logic is encapsulated in a single place, making the code easier to maintain.  
2. \*\*Flexibility\*\*: New types of objects can be added without modifying existing code.  
3. \*\*Abstraction\*\*: The client code depends on abstractions rather than concrete implementations.  
  
---  
  
## \*\*Common Use Cases\*\*  
1. \*\*Database Connections\*\*: Creating database connections dynamically based on the database type (e.g., MySQL, PostgreSQL).  
2. \*\*Logging Frameworks\*\*: Creating different types of loggers (e.g., file logger, console logger).  
3. \*\*Game Development\*\*: Creating different types of characters, weapons, or levels dynamically.  
4. \*\*UI Frameworks\*\*: Creating platform-specific UI components (e.g., Windows, macOS, Linux).  
5. \*\*Parsing Libraries\*\*: Creating parsers for different file formats (e.g., JSON, XML, CSV).  
  
Let me know if you'd like further clarification or additional examples!