



PYTHON
ACADEMY

PADRÕES DE PROJETO EM PYTHON (DESIGN PATTERNS): ABSTRACT FACTORY

Conheça o padrão de projeto Abstract Factory em Python. Veja exemplos práticos e aprenda a implementar em seus projetos.

[PYTHONACADEMY.COM.BR](https://pythonacademy.com.br)

Este ebook foi gerado por



Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs

Deixe que nossa IA faça o trabalho pesado



Syntax Highlight



Adicione Banners Promocionais



Edite em Markdown em Tempo Real



Infográficos feitos por IA

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

✓ **Atualizado para Python 3.13** (Fevereiro/Março 2025)

Design Patterns modernos: tipo hints, dataclasses, quando usar e não usar.

Salve salve Pythonista 🙌

Os **padrões de projeto** são soluções comprovadas para problemas comuns no desenvolvimento de software.

Em Python, aplicar padrões de projeto pode aumentar a **manutenibilidade**, **flexibilidade** e **reusabilidade** do código.

Neste artigo, abordaremos o padrão de projeto **Abstract Factory**.

Vamos explorar seu intuito, o problema que resolve, a estrutura, aplicabilidade e como implementá-lo na prática com Python.

Com exemplos práticos, você entenderá como aplicar o Abstract Factory para melhorar suas aplicações Python.

O Intuito de Utilizar o Abstract Factory em Python

O **Abstract Factory** é um padrão de criação que fornece uma interface para criar famílias de objetos relacionados sem especificar suas classes concretas.

Em Python, utilizar o Abstract Factory ajuda a **isolar** a criação de objetos, promovendo um código mais **flexível** e **extensível**.

Isso é especialmente útil em sistemas que precisam trabalhar com múltiplas variantes de produtos que pertencem a uma mesma família.

O Problema que o Abstract Factory Resolve

Em aplicações complexas, a criação de objetos pode se tornar **complicada** e **acoplada**.

Sem um padrão de criação adequado, gerenciar diferentes variantes de produtos pode levar a um código difícil de manter e escalar.

Além disso, a troca de implementações concretas pode se tornar um **pesadelo**, já que o código depende fortemente das classes específicas utilizadas.

O Racional por Trás do Abstract Factory como Solução

O **Abstract Factory** resolve esses problemas ao **encapsular** a criação de objetos relacionados.

Ele permite que o sistema seja **independente** das classes concretas necessárias para a criação dos objetos.

Isso promove a **flexibilidade**, permitindo que novas famílias de produtos sejam adicionadas sem modificar o código existente.

Estrutura do Abstract Factory

A estrutura do Abstract Factory envolve os seguintes componentes:

- **AbstractFactory:** Interface que declara métodos para criar cada tipo de produto.
- **ConcreteFactory:** Implementações concretas da AbstractFactory, responsáveis por criar objetos de uma família específica.
- **AbstractProduct:** Interfaces para diferentes tipos de produtos.
- **ConcreteProduct:** Implementações concretas dos AbstractProducts.

Diagrama de Classe

```
AbstractFactory
| -- ConcreteFactory1
| -- ConcreteFactory2
AbstractProductA
| -- ConcreteProductA1
| -- ConcreteProductA2
AbstractProductB
| -- ConcreteProductB1
| -- ConcreteProductB2
```

Aplicabilidade do Abstract Factory

O **Abstract Factory** é aplicável quando:

- **Famílias de objetos** são projetadas para serem usadas juntas.
- O sistema deve ser **independente** de como seus produtos são criados, compostos e representados.

- É necessário que o sistema **seja configurado** com uma das várias famílias de produtos.
- **Mudanças nas classes concretas** não devem afetar o sistema.

Antes de continuar... Está curtindo esse conteúdo? 👍

Pausa para novidade: Criei o **DevBook**, uma plataforma que gera ebooks técnicos usando IA. Imagina compilar todo esse conhecimento sobre Abstract Factory em um ebook profissional, com código formatado e tudo — em minutos! Vale conhecer.



Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs





Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

Implementando o Abstract Factory na Prática com Python

Vamos implementar o padrão **Abstract Factory** em Python com um exemplo prático.

Imagine que estamos desenvolvendo um sistema de GUI que deve funcionar com diferentes estilos de botões e caixas de texto.

Passo 1: Definir Interfaces dos Produtos

Primeiro, definimos as interfaces para os produtos que serão criados pelas fábricas.

```
from abc import ABC, abstractmethod

# Interfaces dos Produtos
class Botao(ABC):
    @abstractmethod
    def desenhar(self):
        pass

class CaixaDeTexto(ABC):
    @abstractmethod
    def desenhar(self):
        pass
```

Explicação do código:

- 1. Importações:** Importamos `ABC` e `abstractmethod` para criar classes abstratas.
- 2. Interfaces dos Produtos:** Definimos as classes abstratas `Botao` e `CaixaDeTexto` com o método abstrato `desenhar`.

Passo 2: Implementar Produtos Concretos

Agora, implementamos as classes concretas que herdam das interfaces dos produtos.

```
# Produtos Concretos para Tema Claro
class BotaoClaro(Botao):
    def desenhar(self):
        print("Desenhando Botão Claro")

class CaixaDeTextoClaro(CaixaDeTexto):
    def desenhar(self):
        print("Desenhando Caixa de Texto Clara")

# Produtos Concretos para Tema Escuro
class BotaoEscuro(Botao):
    def desenhar(self):
        print("Desenhando Botão Escuro")

class CaixaDeTextoEscuro(CaixaDeTexto):
    def desenhar(self):
        print("Desenhando Caixa de Texto Escura")
```

Explicação do código:

1. **Produtos Concretos:** Criamos `BotaoClaro`, `CaixaDeTextoClaro`, `BotaoEscuro` e `CaixaDeTextoEscuro` que implementam os métodos `desenhar` de suas respectivas interfaces.

Passo 3: Definir a Interface da Fábrica Abstrata

Definimos a interface que declara métodos para criar cada tipo de produto.


```
# Interface da Fábrica Abstrata
class FabricaGUI(ABC):
    @abstractmethod
    def criar_botao(self) -> Botao:
        pass

    @abstractmethod
    def criar_caixa_de_texto(self) -> CaixaDeTexto:
        pass
```

Explicação do código:

- Fábrica Abstrata:** A classe `FabricaGUI` declara os métodos `criar_botao` e `criar_caixa_de_texto` que devem ser implementados pelas fábricas concretas.

Passo 4: Implementar Fábricas Concretas

Implementamos as fábricas concretas que criam os produtos específicos.

```
# Fábricas Concretas
class FabricaClaro(FabricaGUI):
    def criar_botao(self) -> Botao:
        return BotaoClaro()

    def criar_caixa_de_texto(self) -> CaixaDeTexto:
        return CaixaDeTextoClaro()

class FabricaEscuro(FabricaGUI):
    def criar_botao(self) -> Botao:
        return BotaoEscuro()

    def criar_caixa_de_texto(self) -> CaixaDeTexto:
        return CaixaDeTextoEscuro()
```

Explicação do código:

1. **FabricaClaro:** Implementa `criar_botao` e `criar_caixa_de_texto` retornando objetos claros.
2. **FabricaEscuro:** Implementa os mesmos métodos retornando objetos escuros.

Passo 5: Utilizar a Fábrica Abstrata

Finalmente, utilizamos a fábrica abstrata para criar os produtos sem depender das classes concretas.

```
def aplicar_tema(fabrica: FabricaGUI):  
    botao = fabrica.criar_botao()  
    caixa_texto = fabrica.criar_caixa_de_texto()  
    botao.desenhar()  
    caixa_texto.desenhar()  
  
# Exemplo com Tema Claro  
print("Aplicando Tema Claro:")  
fabrica_claro = FabricaClaro()  
aplicar_tema(fabrica_claro)  
  
# Exemplo com Tema Escuro  
print("\nAplicando Tema Escuro:")  
fabrica_escuro = FabricaEscuro()  
aplicar_tema(fabrica_escuro)
```

Explicação do código:

1. **Função `aplicar_tema`:** Recebe uma fábrica abstrata, cria os produtos e chama seus métodos `desenhar`.

2. **Aplicação do Tema Claro e Escuro:** Criamos instâncias de `FabricaClaro` e `FabricaEscuro` e aplicamos os temas, observando como os objetos são criados de forma abstrata.

Saída Esperada:

E a saída será:

```
Aplicando Tema Claro:
Desenhando Botão Claro
Desenhando Caixa de Texto Clara

Aplicando Tema Escuro:
Desenhando Botão Escuro
Desenhando Caixa de Texto Escura
```

Exemplos Práticos de Abstract Factory em Python

Vamos explorar mais dois exemplos práticos onde o **Abstract Factory** pode ser aplicado em Python.

Exemplo 1: Fabricação de Carros

Imagine que estamos desenvolvendo um sistema que deve funcionar com diferentes tipos de carros (SUV, Sedã). Cada tipo de carro possui partes diferentes como Motor e Roda.

```

from abc import ABC, abstractmethod

# Interfaces dos Produtos
class Motor(ABC):
    @abstractmethod
    def especificacoes(self):
        pass

class Roda(ABC):
    @abstractmethod
    def tipo(self):
        pass

# Produtos Concretos
class MotorV8(Motor):
    def especificacoes(self):
        print("Motor V8: Potência de 450cv")

class RodaEsportiva(Roda):
    def tipo(self):
        print("Roda Esportiva: 18 polegadas")

class MotorEconomico(Motor):
    def especificacoes(self):
        print("Motor Econômico: Consumo reduzido")

class RodaCompacta(Roda):
    def tipo(self):
        print("Roda Compacta: 16 polegadas")

# Fábrica Abstrata
class FabricaCarro(ABC):
    @abstractmethod
    def criar_motor(self) -> Motor:
        pass

    @abstractmethod
    def criar_roda(self) -> Roda:
        pass

# Fábricas Concretas

```

```

class FabricaEsportiva(FabricaCarro):
    def criar_motor(self) -> Motor:
        return MotorV8()

    def criar_roda(self) -> Roda:
        return RodaEsportiva()

class FabricaEconomica(FabricaCarro):
    def criar_motor(self) -> Motor:
        return MotorEconomico()

    def criar_roda(self) -> Roda:
        return RodaCompacta()

# Utilização
def montar_carro(fabrica: FabricaCarro):
    motor = fabrica.criar_motor()
    roda = fabrica.criar_roda()
    motor.especificacoes()
    roda.tipo()

print("Montando Carro Esportivo:")
fabrica_esportiva = FabricaEsportiva()
montar_carro(fabrica_esportiva)

print("\nMontando Carro Econômico:")
fabrica_economica = FabricaEconomica()
montar_carro(fabrica_economica)

```

Saída Esperada:

E a saída será:

Montando Carro Esportivo:
Motor V8: Potência de 450cv
Roda Esportiva: 18 polegadas

Montando Carro Econômico:
Motor Econômico: Consumo reduzido
Roda Compacta: 16 polegadas

Exemplo 2: Criação de Interfaces de Usuário

Suponha que estamos desenvolvendo uma aplicação que deve suportar diferentes sistemas operacionais (Windows, MacOS). Cada sistema operacional possui estilos diferentes para botões e menus.


```

from abc import ABC, abstractmethod

# Interfaces dos Produtos
class Botao(ABC):
    @abstractmethod
    def renderizar(self):
        pass

class Menu(ABC):
    @abstractmethod
    def exibir(self):
        pass

# Produtos Concretos
class BotaoWindows(Botao):
    def renderizar(self):
        print("Renderizando Botão no estilo Windows")

class MenuWindows(Menu):
    def exibir(self):
        print("Exibindo Menu no estilo Windows")

class BotaoMac(Botao):
    def renderizar(self):
        print("Renderizando Botão no estilo MacOS")

class MenuMac(Menu):
    def exibir(self):
        print("Exibindo Menu no estilo MacOS")

# Fábrica Abstrata
class FabricaInterface(ABC):
    @abstractmethod
    def criar_botao(self) -> Botao:
        pass

    @abstractmethod
    def criar_menu(self) -> Menu:
        pass

# Fábricas Concretas

```

```

class FabricaWindows(FabricaInterface):
    def criar_botao(self) -> Botao:
        return BotaoWindows()

    def criar_menu(self) -> Menu:
        return MenuWindows()

class FabricaMac(FabricaInterface):
    def criar_botao(self) -> Botao:
        return BotaoMac()

    def criar_menu(self) -> Menu:
        return MenuMac()

# Utilização
def configurar_interface(fabrica: FabricaInterface):
    botao = fabrica.criar_botao()
    menu = fabrica.criar_menu()
    botao.renderizar()
    menu.exibir()

print("Configurando Interface para Windows:")
fabrica_windows = FabricaWindows()
configurar_interface(fabrica_windows)

print("\nConfigurando Interface para MacOS:")
fabrica_mac = FabricaMac()
configurar_interface(fabrica_mac)

```

Saída Esperada:

E a saída será:

Configurando Interface para Windows:
Renderizando Botão no estilo Windows
Exibindo Menu no estilo Windows

Configurando Interface para MacOS:
Renderizando Botão no estilo MacOS
Exibindo Menu no estilo MacOS

Conclusão

O padrão de projeto **Abstract Factory** é uma ferramenta poderosa para criar famílias de objetos relacionados de forma **abstrata** e **flexível**.

Em Python, sua implementação promove um código mais **organizado**, **extensível** e **manutenível**, facilitando a gestão de diferentes variantes de produtos sem acoplamento direto às suas implementações concretas.

Através dos exemplos apresentados, vimos como aplicar o Abstract Factory em diferentes cenários práticos, desde interfaces gráficas até sistemas automotivos.

Incorporar padrões de projeto como o Abstract Factory em suas aplicações Python pode significativamente **melhorar a estrutura** e **a qualidade** do seu código, tornando-o mais robusto e adaptável às mudanças futuras.

Não se esqueça de conferir!



DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



 Syntax Highlight

 Infográficos feitos por IA

 Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

 Edite em Markdown em Tempo Real

TESTE AGORA 

 PRIMEIRO CAPÍTULO 100% GRÁTIS