LINGUAGEM DE PROGRAMAÇÃO APLICADA

AULA 3

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

Ao longo desta etapa de estudos, você aprenderá sobre:

- módulos, diretórios e pacotes;
- classes abstratas;
- herança múltipla;
- introdução aos *design patterns*.

TEMA 1 – MÓDULOS, DIRETÓRIOS E PACOTES

Em Python, um módulo é um arquivo contendo definições e declarações de Python. O nome do arquivo é o nome do módulo com a extensão “.py” no final. Um módulo pode conter funções, classes e variáveis, bem como instruções executáveis.

Os módulos permitem organizar o código Python em arquivos separados para melhorar a modularidade, a legibilidade e a reutilização do código. Eles também facilitam a manutenção e a colaboração em projetos maiores, pois diferentes partes do código podem ser isoladas em módulos independentes.

Para usar um módulo em um programa Python, você pode importá-lo usando a instrução import. Por exemplo, se você tiver um arquivo chamado meu_modulo.py contendo algumas funções úteis, você pode importá-lo da seguinte forma:

```
import meu_modulo
```

Depois de importar o módulo, você pode acessar as funções, classes e variáveis definidas nele usando a sintaxe nome_do_modulo.nome_da_funcao().

Um diretório em Python é uma pasta que pode conter um ou mais módulos, ou até mesmo outros diretórios dentro dele. Sendo assim, diretórios funcionam da mesma forma que diretórios que você já conhece em sistemas operacionais como Windows e Linux.

Por fim, um pacote Python é uma estrutura que contém um conjunto de módulos Python relacionados e outros pacotes. Ele é usado para organizar e distribuir código Python em hierarquias de diretórios lógicos.

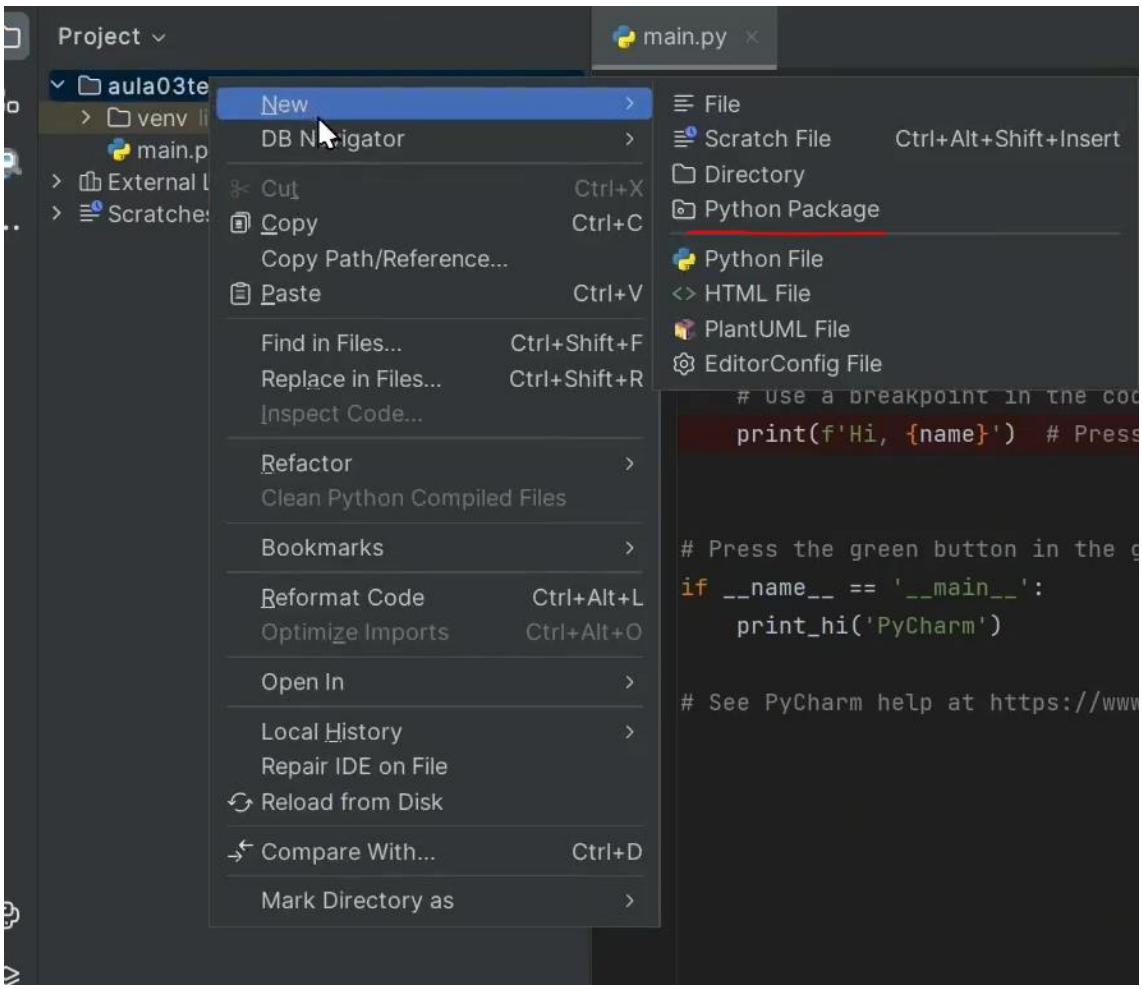
Um pacote Python é basicamente um diretório que contém um arquivo especial chamado __init__.py. Esse arquivo indica ao interpretador Python que

o diretório é um pacote e pode conter módulos Python ou outros pacotes. Este arquivo pode ser vazio ou pode conter código de inicialização para o pacote.

Os pacotes Python são usados para dividir e organizar o código em componentes lógicos e separar funcionalidades relacionadas em diferentes arquivos e diretórios. Isso facilita a modularidade, a reutilização de código e a manutenção do projeto, tornando-o mais organizado e fácil de gerenciar.

Podemos criar nosso próprio pacote Python no PyCharm. Para isso, vamos ao projeto em que queremos criar o pacote e vamos em NEW > PYTHON PACKAGE.

Figura 1 – Criando um pacote Python



TEMA 2 – CLASSE ABSTRATA BASE (*ABSTRACT BASE CLASS* – ABC) E HERANÇA MÚLTIPLA

Vejamos as definições a seguir.

2.1 Classe Abstrata Base (*Abstract Base Class* – ABC)

Uma Classe Abstrata Base (*Abstract Base Class* – ABC) é uma classe que serve como modelo para outras classes e não pode ser instanciada diretamente. Geralmente, ela contém métodos abstratos, que são métodos sem implementação definida, deixando para as subclasses fornecerem sua própria implementação.

As ABCs em Python são definidas no módulo abc (*Abstract Base Classes*) e são usadas para definir interfaces comuns que devem ser implementadas por diferentes classes. Elas fornecem um meio de definir um contrato entre a classe base e suas subclasses, garantindo que todas as subclasses forneçam implementações consistentes para os métodos especificados pela classe abstrata.

Para criar uma classe abstrata em Python, você pode herdar da classe ABC do módulo abc e usar o decorador @abstractmethod para marcar os métodos que devem ser implementados pelas subclasses. Por exemplo:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def fazer_som(self):
        pass

class Cachorro(Animal):
    def fazer_som(self):
        return "Au Au"

class Gato(Animal):
    def fazer_som(self):
        return "Miau"
```

2.2 Herança múltipla

A herança múltipla é um conceito da Programação Orientada a Objetos (POO) em que uma classe pode herdar atributos e métodos de mais de uma classe pai. Isso significa que uma classe pode ser derivada de várias classes base, permitindo que ela compartilhe características e comportamentos de todas essas classes.

O uso de herança múltipla é bastante necessário quando falamos de classe ABC, e é por isso que precisamos trazer este conceito aqui neste momento. Abaixo, vemos um exemplo de herança múltipla:

```
class Pai1:
    def metodo_pai1(self):
        print("Método do Pai 1")

class Pai2:
    def metodo_pai2(self):
        print("Método do Pai 2")

class Filho(Pai1, Pai2):
    pass
```

TEMA 3 – INTRODUÇÃO AOS PADRÕES DE DESIGN DE SOFTWARE

Imagine que você gosta muito de construir casas com Lego. Às vezes, quando está construindo uma casa, você se depara com problemas como fazer um telhado resistente ou adicionar uma janela bonita.

Bem, os padrões de design de software são como truques ou instruções especiais para resolver esses tipos de problemas quando estamos construindo software. Assim como você pode usar um padrão específico de peças de Lego para criar um telhado forte, nós usamos padrões de design para construir programas de computador de forma inteligente e organizada.

Então, quando estamos programando, encontramos problemas comuns, como criar objetos, organizar as partes de um programa ou fazer objetos trabalharem juntos. Os padrões de design nos dão soluções testadas e aprovadas para esses problemas.

Eles são como receitas que nos mostram como resolver um problema específico de maneira inteligente. Usar esses padrões torna nosso código mais fácil de entender, mais flexível e mais fácil de modificar, assim como usar os padrões de Lego torna suas construções melhores e mais legais.

Podemos classificar os *design patterns* em alguns tipos:

- Padrões estruturais: lidam com a organização de classes e objetos, fornecendo uma maneira de compor estruturas maiores a partir de partes menores. Citamos dois tipos que trabalharemos em nossos estudos: Adaptador e Proxy.
- Padrões comportamentais: lidam com a interação entre objetos e a definição de responsabilidades, gerenciando a comunicação e o comportamento entre diferentes objetos. Citamos dois tipos que trabalharemos em nossos estudos: Mediador e Observador.
- Padrões de criação lidam com a criação de objetos, oferecendo mecanismos flexíveis para instanciar classes. Citamos dois tipos que trabalharemos em nossos estudos: Builder e Factory.

TEMA 4 – PADRÃO DE DESIGN CRIACIONAL: FACTORY (FÁBRICA)

Imagine que você tem uma pizzaria e, às vezes, as pessoas pedem diferentes tipos de pizza: margherita, pepperoni, quatro queijos etc. Agora, você pode criar cada pizza manualmente, passo a passo, toda vez que alguém pedir, mas isso leva muito tempo e esforço.

Em vez disso, você pode usar um processo mais inteligente, chamado padrão de design Factory. Aqui está como funciona:

1. Você tem uma linha de produção onde você faz diferentes tipos de pizzas.
2. Quando alguém faz um pedido, em vez de fazer a pizza manualmente, você simplesmente diz à linha de produção o tipo de pizza que você precisa.
3. A linha de produção então faz a pizza certa para você, seguindo as receitas e instruções pré-definidas para cada tipo de pizza.

Então, em vez de fazer cada pizza do zero toda vez, você usa a “fábrica de pizzas” para criar a pizza certa quando alguém pede. Isso economiza tempo e esforço e garante que cada pizza seja consistente e de alta qualidade.

No mundo da programação, o padrão de design Factory funciona de maneira semelhante. Ele é usado para criar objetos sem expor a lógica de criação diretamente ao cliente. Em vez disso, o cliente simplesmente solicita o objeto necessário à fábrica, que cuida da criação e devolução do objeto apropriado. Isso torna o código mais flexível, escalável e fácil de manter.

Vejamos este exemplo em Python. Abaixo, criamos uma classe que contém todas as etapas de fabricação da pizza. Criamos também classes para cada tipo de pizza, como Margherita e Pepperoni, por exemplo. Por fim, criamos uma classe de Factory, que irá fabricar a pizza de acordo com o pedido dos clientes.

```
class Pizza:
    def prepare(self):
        pass

    def bake(self):
        pass

    def cut(self):
        pass

    def box(self):
        pass

class MargheritaPizza(Pizza):
    def prepare(self):
        print("Preparando Margherita Pizza...")

    def bake(self):
        print("Assando Margherita Pizza...")

    def cut(self):
        print("Cortando Margherita Pizza...")

    def box(self):
        print("Embalando Margherita Pizza...")

class PepperoniPizza(Pizza):
```

```
    def prepare(self):
        print("Preparando Pepperoni Pizza...")

    def bake(self):
        print("Assando Pepperoni Pizza...")

    def cut(self):
        print("Cortando Pepperoni Pizza...")

    def box(self):
        print("Embalando Pepperoni Pizza...")

class PizzaFactory:
    def create_pizza(self, pizza_type):
        if pizza_type == "margherita":
            return MargheritaPizza()
        elif pizza_type == "pepperoni":
            return PepperoniPizza()
        else:
            raise ValueError("Tipo de pizza desconhecido.")

# Exemplo de uso da fábrica de pizzas
factory = PizzaFactory()

pizza1 = factory.create_pizza("margherita")
pizza1.prepare()
pizza1.bake()
pizza1.cut()
pizza1.box()

pizza2 = factory.create_pizza("pepperoni")
pizza2.prepare()
pizza2.bake()
pizza2.cut()
pizza2.box()
```

TEMA 5 – PADRÃO DE DESIGN CRIACIONAL: BUILDER (CONSTRUTOR)

Imagine que você está construindo uma casa do zero. Existem muitos detalhes e etapas envolvidas: decidir o layout, escolher os materiais, construir as paredes, instalar o telhado etc. Uma casa pode ter um ou mais quartos, um ou mais banheiros, ter ou não ter garagem etc. É muito trabalho e cada casa pode ter suas próprias características únicas.

Agora, o padrão de design Builder entra em cena para nos ajudar a simplificar esse processo. Aqui está como funciona:

1. Primeiro, temos o diretor da construção, que é responsável por coordenar todo o processo de construção.
2. Em seguida, temos o construtor, que é responsável por construir cada parte da casa, uma de cada vez.
3. O diretor diz ao construtor o que precisa ser feito, e o construtor cuida de todas as etapas necessárias para construir a casa, seguindo um plano predefinido.
4. No final, temos uma casa completa, construída de acordo com as especificações fornecidas pelo diretor.

Então, em vez de lidar com todos os detalhes de construção sozinho, o diretor delega as tarefas para o construtor, que sabe exatamente o que fazer em cada etapa.

No mundo da programação, o padrão de design Builder é usado para construir objetos complexos passo a passo. Em vez de criar o objeto diretamente, você usa um construtor para definir suas propriedades e configurá-lo de acordo com suas necessidades. Isso torna o código mais modular, flexível e fácil de entender, especialmente quando você está lidando com objetos complexos com muitas partes e opções.

A seguir, vemos esse exemplo de construção de uma casa, primeiramente, sem o uso de design pattern:

```
class Casa:
    def __init__(self, num_quartos=None, num_banheiros=None,
tem_garagem=False, tem_jardim=False):
        self.num_quartos = num_quartos
        self.num_banheiros = num_banheiros
        self.tem_garagem = tem_garagem
```

```
        self.tem_jardim = tem_jardim

    def __str__(self):
        caracteristicas = []
        if self.num_quartos:
            caracteristicas.append(f"Número de quartos:
{self.num_quartos}")
        if self.num_banheiros:
            caracteristicas.append(f"Número de banheiros:
{self.num_banheiros}")
        if self.tem_garagem:
            caracteristicas.append("Possui garagem")
        if self.tem_jardim:
            caracteristicas.append("Possui jardim")

        return ', '.join(caracteristicas)


# Construindo uma casa com garagem e jardim
casa = Casa(num_quartos=3, num_banheiros=2, tem_garagem=True,
tem_jardim=True)
print("Características da casa:", casa)

# Construindo outra casa sem garagem e com 4 quartos
outra_casa = Casa(num_quartos=4, num_banheiros=3)
print("Características da outra casa:", outra_casa)
```

A seguir, vemos a mesma implementação usando o design pattern Builder:

```
# Classe Casa representa o produto final que queremos construir
class Casa:
    def __init__(self):
        self.num_quartos = None
        self.num_banheiros = None
        self.tem_garagem = False
        self.tem_jardim = False

    def __str__(self):
        caracteristicas = []
        if self.num_quartos:
            caracteristicas.append(f"Número de quartos:
{self.num_quartos}")
```

```
        if self.num_banheiros:
            caracteristicas.append(f"Número de banheiros:
{self.num_banheiros}")
        if self.tem_garagem:
            caracteristicas.append("Possui garagem")
        if self.tem_jardim:
            caracteristicas.append("Possui jardim")

        return ', '.join(caracteristicas)


# Classe Builder para construir a Casa passo a passo
class CasaBuilder:
    def __init__(self):
        self.casa = Casa()

    def set_num_quartos(self, num_quartos):
        self.casa.num_quartos = num_quartos
        return self

    def set_num_banheiros(self, num_banheiros):
        self.casa.num_banheiros = num_banheiros
        return self

    def adicionar_garagem(self):
        self.casa.tem_garagem = True
        return self

    def adicionar_jardim(self):
        self.casa.tem_jardim = True
        return self

    def obter_casa(self):
        return self.casa


# Construindo uma casa com garagem e jardim
builder_casa = CasaBuilder()
casa = builder_casa.set_num_quartos(3).\
                    set_num_banheiros(2).\
                    adicionar_garagem().\
                    adicionar_jardim().\
                    obter_casa()
```

```
print("Características da casa:", casa)

# Construindo outra casa sem garagem e com 4 quartos
outra_casa =
CasaBuilder().set_num_quartos(4).set_num_banheiros(3).obter_casa()
print("Características da outra casa:", outra_casa)
```

FINALIZANDO

Durante esta etapa de estudos, você aprendeu sobre:

- módulos, diretórios e pacotes;
- classes abstratas;
- herança múltipla;
- introdução aos padrões de design;
- padrão de design Factory (fábrica);
- padrão de design Builder (construtor).

REFERÊNCIAS

PUGA, S.; RISSETI, G. **Lógica de Programação e Estrutura de Dados**. 3. ed. São Paulo: Pearson, 2016.

ZHART, D. ***Design patterns***: Refactoring Guru. Disponível em: <https://refactoring.guru/design-patterns>. Acesso em: 17 mar. 2024.

ROSEWOOD, E. **Programação orientada a objetos em Python**: dos fundamentos às técnicas avançadas. Edição Independente.