LINGUAGEM DE PROGRAMAÇÃO APLICADA

AULA 4

Prof. Osmar Tenorio Pereira Dias Junior

CONVERSA INICIAL

Ao longo desta etapa você irá:

- aprender a utilizar eventos;
- entender e utilizar decorators;
- utilizar a compreensão de lista (list comprehension);
- continuar nossos estudos com design pattern.

TEMA 1 – EVENTOS

Em programação, um evento é uma ação ou ocorrência que pode ser detectada e manipulada pelo programa. Quando um evento ocorre, como um clique do mouse, pressionamento de tecla, carregamento de página da web etc., o sistema operacional ou o ambiente de execução notifica o programa sobre esse evento. O programa pode então executar um código específico em resposta a esse evento.

Por exemplo, em uma aplicação de interface gráfica de usuário (GUI), um evento pode ser um clique do mouse em um botão. O programa pode ter um código associado a esse evento que seja executado quando o botão é clicado, como abrir uma nova janela ou processar algum dado. A seguir, temos um caso de interrupção de teclado.

```
import keyboard

defon_key_event(event):
    print('Tecla pressionada:', event.name)

print('Pressione qualquer tecla. Pressione "Esc" para sair.')

keyboard.on_press(on_key_event)

# Mantém o programa em execução até pressionar "Esc"
keyboard.wait('esc')
```

1.1 Eventos de tempo

Eventos de tempo em programação, especialmente em Python, referem-se à capacidade de executar determinadas ações em intervalos regulares de tempo ou em momentos específicos no futuro. Isso é útil em uma variedade de cenários, como agendar tarefas periódicas, atualizar dados de forma regular, processar eventos em tempo real e muito mais.

Em Python, você pode lidar com eventos de tempo usando a biblioteca **time** para operações básicas de tempo e a biblioteca **datetime** para manipulação de datas e horas mais complexas. No entanto, uma das maneiras mais comuns de lidar com eventos de tempo é usando a biblioteca **sched**.

```
import sched
import time

# Função que será executada a cada segundo
defprint_message(mensagem):
    print(mensagem)

# Função para agendar o próximo evento
defschedule_next_event(scheduler, intervalo, mensagem):
# Agendando a função print_message para ser executada após 1 segundo
    scheduler.enter(1, 1, schedule_next_event, (scheduler, intervalo,
mensagem))
    print_message(mensagem)

# Criação de um objeto sched
scheduler = sched.scheduler(time.time, time.sleep)

# Agendando o primeiro evento
schedule_next_event(scheduler, 1, "Esta é a mensagem agendada a cada 1
segundo!")

# Executando o loop do scheduler
scheduler.run()
```

TEMA 2 – DECORATORS

Em Python, os decorators são usados para modificar ou estender o comportamento de funções ou métodos sem alterá-los explicitamente. Podemos classificá-los em dois tipos: *decorators* de funções e *decorators* de métodos/classes.

2.1 Decorators em funções

Imagine que você tem uma função em Python que faz algo específico, como imprimir uma mensagem na tela. Agora, digamos que você queira adicionar alguma funcionalidade extra a essa função sem realmente modificá-la. É aí que os decorators entram em cena.

Um decorator em Python é essencialmente uma função que envolve outra função, permitindo adicionar funcionalidades adicionais a ela sem alterá-la diretamente. Então, se você tem uma função existente e deseja adicionar algo extra a ela, como verificar se o usuário está autenticado antes de executá-la, você pode usar um decorator para fazer isso.

Por exemplo, suponha que você tenha uma função *minha_funcao()*, que faz algo simples, como imprimir uma mensagem na tela:

```
defminha_funcao():
    print("Executando minha função")
```

Agora, digamos que você queira adicionar uma verificação de autenticação a essa função, garantindo que apenas usuários autenticados possam executá-la. Você pode definir um decorator para fazer essa verificação:

```
defverificar_autenticacao(funcao):
defwrapper():
if usuario_autenticado():
            funcao()
else:
            print("Usuário não autenticado. Acesso negado.")
return wrapper

@verificar_autenticacao
```

```
defminha_funcao():
    print("Executando minha função")
```

Veja que *verificar_autenticacao* é o decorator. Quando usamos *@verificar_autenticacao* acima da definição da função *minha_funcao()*, estamos aplicando esse decorator a essa função. Isso significa que, sempre que chamamos *minha_funcao()*, na verdade estamos chamando a função *wrapper()*, que verifica primeiro se o usuário está autenticado antes de chamar a função original: *minha_funcao()*.

Os decorators são úteis para adicionar funcionalidades extras a funções sem modificar diretamente seu código, tornando seu código mais modular, legível e fácil de manter.

2.2 Decorators em classes

Agora imagine que você tem uma classe em Python que é uma receita para fazer um bolo. Mas, às vezes, você quer adicionar alguns ingredientes extras ou fazer alguma coisa especial no bolo, como decorá-lo de uma maneira especial.

Podemos dizer que um decorator de classe em Python é como uma receita de bolo que pode ser decorada com ingredientes extras ou tratada de uma maneira especial, sem realmente mexer na receita original. Por exemplo, digamos que você tenha uma classe Bolo, que tem um método *assar()* para assar o bolo:

```
classBolo:
defassar(self):
        print("Bolo assado!")
```

Agora, digamos que você queira adicionar um sabor extra ao bolo sem realmente modificar a classe Bolo. Você pode usar um decorator de classe para fazer isso:

```
defadicionar_sabor_extra(classe):
classDecoradaClasse(classe):
defassar(self):
            super().assar()  # Primeiro, assamos o bolo como de
```

```
costume
            print("Adicionando sabor extra ao bolo!")
return DecoradaClasse

@adicionar_sabor_extra
classBolo:
defassar(self):
        print("Bolo assado!")

meu_bolo = Bolo()
meu_bolo.assar()
```

Aqui, *adicionar_sabor_extra* é o decorator de classe. Quando usamos *@adicionar_sabor_extra* acima da definição da classe Bolo, estamos dizendo ao Python para aplicar esse decorator à classe Bolo.

Então, quando chamamos *meu_bolo.assar()*, o Python na verdade está chamando o método *assar()* da classe DecoradaClasse, que primeiro assa o bolo como de costume e depois adiciona um sabor extra a ele.

Assim como os decorators de função, os decorators de classe são úteis para adicionar funcionalidades extras a classes sem modificar diretamente seu código, tornando seu código mais modular e fácil de entender.

TEMA 2 – COMPREENSÃO DE LISTA (LIST COMPREHENSION)

List comprehensions em Python é uma maneira bem legal de criar listas de forma rápida e fácil. É como se você estivesse criando uma lista usando uma espécie de atalho.

Então, imagina que você tem uma lista de números e quer criar outra lista em que cada número par é elevado ao quadrado. Com list comprehensions, você pode fazer isso em uma linha, sem precisar de um monte de código, e ainda ganha em desempenho.

Vejamos o exemplo do código da maneira convencional, sem list comprehensions.

```
lista_pares_quadrado = []
for num in range(10):
```

```
if num % 2 == 0:
        lista_pares_quadrado.append(num * num)
```

Agora, vejamos o mesmo exemplo com list comprehensions:

```
lista_pares_quadrado = [num * num for num in range(10) if num % 2 == 0]
```

Conseguimos escrever tudo em uma só linha!

Agora imaginemos que queremos que, caso número seja ímpar, ele seja substituído pela string *ímpar*. Na maneira tradicional, fazemos:

```
lista_pares_quadrado = []
for num in range(10):
if num % 2 == 0:
        lista_pares_quadrado.append(num * num)
else:
        lista_pares_quadrado.append('ímpar')
```

A versão com list comprehensions seria:

```
lista_pares_quadrado = [num * num if num % 2 == 0else'impar'for
num in range(10) ]
```

TEMA 3 – PADRÃO DE DESIGN DE COMPORTAMENTAL MEDIADOR (MEDIATOR)

O design pattern mediator é utilizado quando você tem um conjunto de objetos que precisam se comunicar entre si, mas quer evitar que eles se comuniquem diretamente, promovendo um acoplamento fraco entre eles. Vamos aplicar isso ao exemplo dos aviões e uma pista de pouso/decolagem.

Imagine que temos vários aviões que precisam pousar e decolar em uma pista de pouso/decolagem. Se cada avião tentasse coordenar diretamente com a pista e com outros aviões, poderia haver confusão e caos, pois cada avião precisaria trocar mensagens com cada um dos outros aviões para solicitar acesso a pista (imagine esse cenário com milhares de aviões?).

Ao invés desse cenário caótico, podemos criar um mediador, que será a torre de controle do aeroporto. A torre de controle será responsável por

coordenar as operações de pouso e decolagem dos aviões. Cada avião terá que se comunicar apenas com a torre de controle, e não diretamente entre si.

O design pattern mediator funciona dessa forma, como uma torre de controle, sendo um mediador em situações como esta.

Aqui está um exemplo:

```
classTorreDeControle:
def__int__(self):
        self.avioes = []

defadicionar_aviao(self, aviao):
        self.avioes.append(aviao)
        aviao.registrar_torre(self)

defenviar_mensagem(self, aviao, mensagem):
        print(f"Torre de controle para {aviao.nome}:
{mensagem}")
for outro_aviao in self.avioes:
if outro_aviao != aviao:
                outro_aviao.receber_mensagem(aviao, mensagem)

classAviao:
def__init__(self, nome):
        self.nome = nome
        self.torre = None

defregistrar_torre(self, torre):
        self.torre = torre

defenviar_mensagem(self, mensagem):
        self.torre.enviar_mensagem(self, mensagem)

defreceber_mensagem(self, aviao, mensagem):
        print(f"{self.nome} recebeu uma mensagem de
{aviao.nome}: {mensagem}")

#__main__
```

```
#Criando a torre de controle
torre_de_controle = TorreDeControle()

aviao1 = Aviao("Avião 1")
aviao2 = Aviao("Avião 2")
aviao3 = Aviao("Avião 3")

#Adicionando os aviões na torre de controle
torre_de_controle.adicionar_aviao(aviao1)
torre_de_controle.adicionar_aviao(aviao2)
torre_de_controle.adicionar_aviao(aviao3)

# Enviando mensagens entre aviões

aviao1.enviar_mensagem("Vamos decolar!")
aviao2.enviar_mensagem("Confirmando, prontos para decolar!")
aviao3.enviar_mensagem("Aguardando autorização para decolar!")
```

TEMA 4 – PADRÃO DE DESIGN DE CRIACIONAL OBSERVADOR

O design pattern observer é usado quando você tem um objeto (o "sujeito" ou "observável") que precisa notificar outros objetos (os "observadores") sobre mudanças em seu estado. É como se você estivesse sintonizando (se inscrevendo) em uma estação de rádio para ouvi-la.

Imagine que você tem uma estação de rádio que transmite diferentes estações, como rock, pop e jazz. Você tem ouvintes que querem saber quando a estação muda para uma nova frequência. Aqui está como podemos implementar isso usando o design pattern observer:

```
classMensageiro:
def__init__(self):
        self.observadores = []

defadicionar_observador(self, observador):
        self.observadores.append(observador)

defremover_observador(self, observador):
```

```
        self.observadores.remove(observador)

defnotificar_observadores(self, remetente, mensagem):
for observador in self.observadores:
            observador.receber_notificacao(remetente, mensagem)

classUsuario:
def__init__(self, nome):
        self.nome = nome

defreceber_notificacao(self, remetente, mensagem):
        print(f"[{self.nome} Nova mensagem de {remetente}:
{mensagem}]")

classGrupo:
def__init__(self, nome):
        self.nome = nome
        self.membros = []

defadicionar_membro(self, membro):
        self.membros.append(membro)

defreceber_notificacao(self, remetente, mensagem):
        print(f"Grupo {self.nome}: Nova mensagem de {remetente}:
{mensagem}")


mensageiro = Mensageiro()

# Criando usuários
usuario1 = Usuario("Alice")
usuario2 = Usuario("Bob")
usuario3 = Usuario("Charlie")

#Criando grupos
grupo1 = Grupo("Famili")
grupo2 = Grupo("Trabalho")
```

```
#Adicionando observadores ao mensageiro
mensageiro.adicionar_observador(usuario1)
mensageiro.adicionar_observador(usuario2)
mensageiro.adicionar_observador(usuario3)

# Adicionando membros ao grupo
grupo1.adicionar_membro(usuario1)
grupo1.adicionar_membro(usuario2)

grupo2.adicionar_membro(usuario2)
grupo2.adicionar_membro(usuario3)

#Enviando mensagens
mensageiro.notificar_observadores("Admin", "Bem-vindos ao nosso aplicativo de mensagens!")
grupo1.receber_notificacao("Admin", "Nova reunião marcada para amanhã.")
usuario3.receber_notificacao("Alice", "Oi, tudo bem?")
```

```
usuario3.receber_notificacao("Alice", "Oi, tudo bem?")
```

FINALIZANDO

Ao longo desta etapa, você aprendeu:

- como utilizar eventos;
- entender e utilizar *decorators*;
- utilizar compreensão de lista (*list comprehension*);
- como usar o design pattern mediator;
- como usar o design pattern observer;

REFERÊNCIAS

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.

Dmitry Z. **Design Patter.** Refactoring Guru. Acesso em: <https://refactoring.guru/design-patterns>. Acesso em: 19 mar. 2024.

ROSEWOOD, E. **Programação orientada a objetos em Python**: Dos fundamentos às técnicas avançadas. [S.l]: [s.n], [S.d.]. Produção independente.