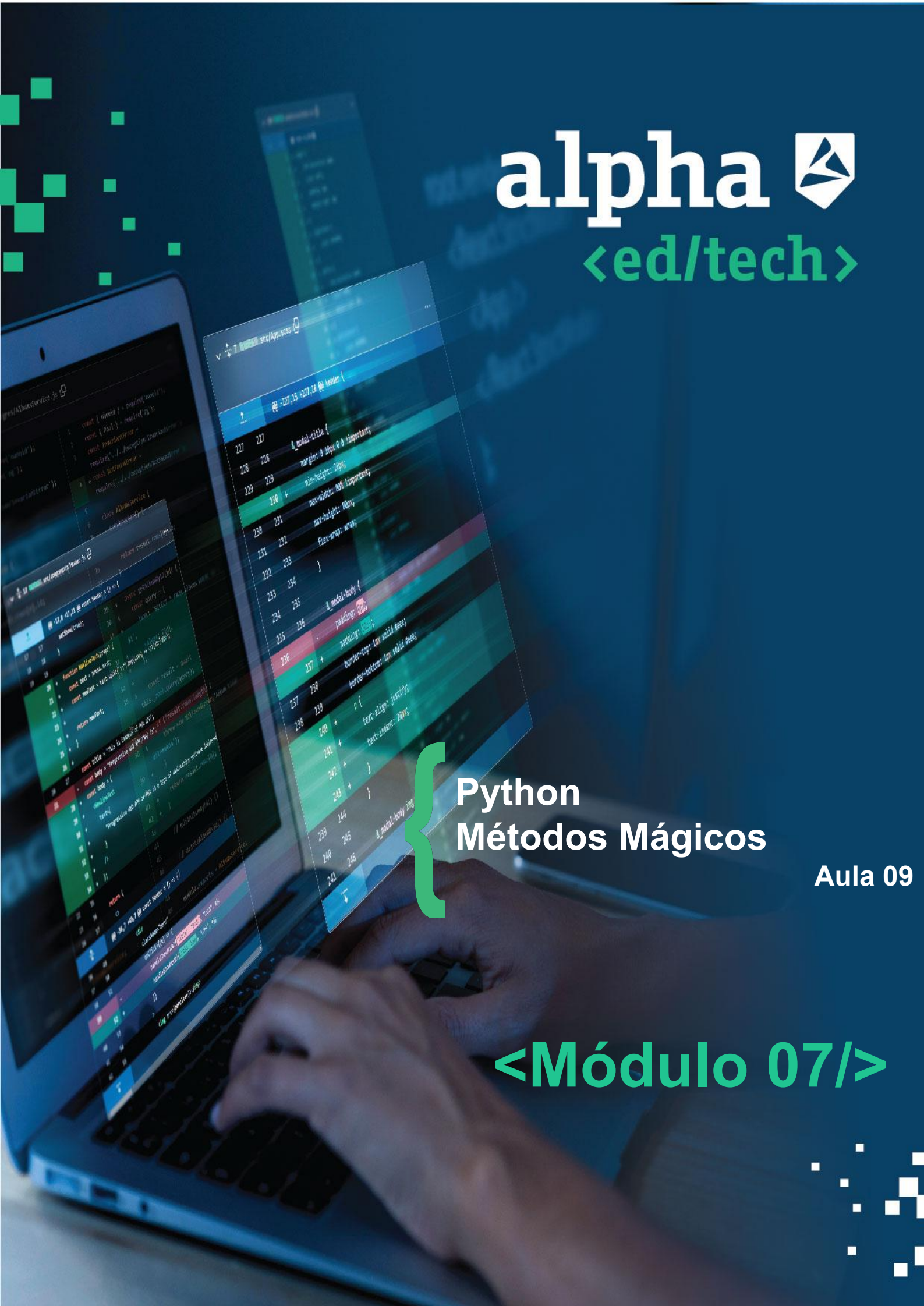
alpha
<ed/tech>
Python
Métodos Mágicos
Aula 09
<Módulo 07/>

Métodos mágicos
também conhecidos como métodos especiais

Métodos Mágicos em Python!

Podemos utilizar estes métodos especiais para tornar nosso código mais elegante, poderoso e pythônico.

São funções com nomes especiais que iniciam e terminam com dois underscores (__) em Python.
Eles permitem que você customize o comportamento de suas classes, definindo como seus objetos interagem com operadores, funções built-in e outras partes da linguagem.

Por que são chamados de "mágicos"? Porque são invocados implicitamente pelo interpretador Python em determinadas situações, como quando você realiza uma operação aritmética com objetos de uma classe personalizada ou quando tenta imprimir um objeto.

Alguns dos métodos mágicos mais comuns e suas funcionalidades:

- `__init__`**:** Chamado automaticamente quando um objeto é criado. É usado para inicializar os atributos do objeto.
- `__str__`**:** Retorna uma representação em string do objeto, utilizada quando você chama `print()` em um objeto.
- `__repr__`**:** Retorna uma representação do objeto que pode ser usada para recriá-lo, útil para depuração.
- `__len__`**:** Retorna o comprimento do objeto, permitindo o uso da função `len()`.
- `__add__`**:** Define o comportamento do operador de adição (`+`) para objetos da classe.
- `__sub__`**:** Define o comportamento do operador de subtração (`-`).
- `__mul__`**:** Define o comportamento do operador de multiplicação (`*`).
- `__truediv__`**:** Define o comportamento da divisão verdadeira (`/`).
- `__floordiv__`**:** Define o comportamento da divisão inteira (`//`).
- `__mod__`**:** Define o comportamento do operador módulo (`%`).
- `__lt__`**:** Define o comportamento do operador menor que (`<`).
- `__gt__`**:** Define o comportamento do operador maior que (`>`).
- `__eq__`**:** Define o comportamento do operador de igualdade (`==`).
- `__ne__`**:** Define o comportamento do operador de diferença (`!=`).
- `__getitem__`**:** Permite que os objetos sejam indexados como listas ou dicionários.
- `__setitem__`**:** Permite que os valores de um objeto sejam atribuídos usando índices.
- `__delitem__`**:** Permite que itens sejam deletados de um objeto usando índices.
- `__iter__`**:** Torna um objeto iterável, permitindo o uso em loops `for`.
- `__next__`**:** Retorna o próximo elemento em uma iteração.
- `__call__`**:** Permite que instâncias de uma classe sejam chamadas como funções.

E muitos outros! A lista completa de métodos mágicos pode ser encontrada na documentação oficial do Python.

1. Representação de Objetos

Problema: Imagine que você está desenvolvendo um sistema de gerenciamento de biblioteca e criou uma classe `Livro`. Ao tentar imprimir informações sobre um livro, você recebe algo como:

```
class Livro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor


livro = Livro("1984", "George Orwell")
print(livro)
# output
# <__main__.Livro object at 0x7f3f6b6c9d90>
```

Solução: Podemos utilizar os métodos `__str__` e `__repr__` para fornecer representações mais úteis do nosso objeto.

```
class Livro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def __str__(self):
        return f"{self.titulo} por {self.autor}"

    def __repr__(self):
        return f'Livro(titulo="{self.titulo}", autor="{self.autor}")'

# Uso:
livro = Livro("1984", "George Orwell")
print(livro)  # Saída: 1984 por George Orwell
print(repr(livro))  # Saída: Livro(titulo="1984", autor="George Orwell")
```

Metáfora: Pense no `__str__` como seu "nome social" - como você quer ser chamado no dia a dia, e no `__repr__` como seu "nome completo no RG" - uma identificação formal e precisa.

2. Operadores Aritméticos

Problema: Você está criando uma classe `Dinheiro` para representar valores monetários e quer realizar operações matemáticas com ela.

```python
class Dinheiro:
    def __init__(self, valor):
        self.valor = valor

    def __add__(self, outro):
        return Dinheiro(self.valor + outro.valor)

    def __sub__(self, outro):
        return Dinheiro(self.valor - outro.valor)

    def __str__(self):
        return f"R$ {self.valor:.2f}"

# Uso:
salario = Dinheiro(1000)
bonus = Dinheiro(200)
total = salario + bonus
print(total)  # Saída: R$ 1200.00
```

Metáfora: Os operadores aritméticos são como regras de um jogo - você define como as peças (objetos) podem interagir entre si.

Problema Avançado: Você precisa fazer herança de uma classe melhorada (refatorada) dessa `class Dinheiro`, que sirva tanto para Reais, como para Dólares ou para Euros. Precisa então ter mais um atributo na classe para indicar qual a moeda, precisa do símbolo de Dólar e o símbolo do Euro, para gerar a saída. Será que você vai usar polimorfismo? Vamos deixar esse problema avançado para os exercícios.

3. Comparações

Problema: Você precisa comparar objetos personalizados, como ao ordenar uma lista de produtos por preço.

```python
# Exemplo de Comparação entre Produtos usando Métodos Mágicos

class Produto:
    def __init__(self, nome, preco):
        self.nome = nome
        self.preco = preco

    # Maior que (>)
    def __gt__(self, outro):
        if not isinstance(outro, Produto):
            raise TypeError("Não é possível comparar um Produto com outro tipo")
        return self.preco > outro.preco

    # Menor que (<)
    def __lt__(self, outro):
        if not isinstance(outro, Produto):
            raise TypeError("Não é possível comparar um Produto com outro tipo")
        return self.preco < outro.preco
```

```
    # Representação em string do objeto
    def __str__(self):
        return f"{self.nome}: R$ {self.preco:.2f}"

    # Representação formal do objeto
    def __repr__(self):
        return f'Produto(nome="{self.nome}", preco={self.preco})'

# Exemplos de uso:
if __name__ == "__main__":
    # Criando alguns produtos
    cafe = Produto("Café", 15.90)
    pao = Produto("Pão", 8.50)
    laptop = Produto("Notebook", 3500.00)
    mouse = Produto("Mouse", 50.00)

    # Comparações individuais
    print(f"\n=== Comparações Individuais ===")
    print(f"{cafe} > {pao}? {cafe > pao}")  # True (15.90 > 8.50)
    print(f"{mouse} < {laptop}? {mouse < laptop}")  # True (50.00 < 3500.00)

    # Criando uma lista e ordenando
    produtos = [cafe, pao, laptop, mouse]
    print(f"\n=== Lista Original ===")
    for p in produtos:
        print(p)

    produtos_ordenados = sorted(produtos)  # Ordena do menor para o maior preço
    print(f"\n=== Lista Ordenada (menor para maior preço) ===")
    for p in produtos_ordenados:
        print(p)

    # Encontrando produto mais caro e mais barato
    mais_caro = max(produtos)
    mais_barato = min(produtos)
    print(f"\n=== Produto mais caro e mais barato ===")
    print(f"Mais caro: {mais_caro}")
    print(f"Mais barato: {mais_barato}")

    # Filtrando produtos
    produtos_caros = [p for p in produtos if p > mouse]
    print(f"\n=== Produtos mais caros que o mouse (R$ 50.00) ===")
    for p in produtos_caros:
        print(p)

    # Testando igualdade
    cafe2 = Produto("Café Premium", 15.90)
    print(f"\n=== Teste de Igualdade ===")
    print(f"{cafe} tem mesmo preço que {cafe2}? {cafe == cafe2}")

    # Tratamento de erro
    try:
        print(cafe > "teste")
    except TypeError as e:
        print(f"\nErro ao comparar tipos diferentes: {e}")
```

Este exemplo demonstra:

1. Implementação dos métodos mágicos de comparação (`__gt__`, `__lt__`, `__eq__`)
2. Validação de tipos nas comparações
3. Diferentes formas de usar as comparações:
   - Comparação direta entre dois produtos
   - Ordenação de lista de produtos
   - Uso de `max()` e `min()`
   - Filtragem baseada em comparação
   - Teste de igualdade entre objetos
4. Tratamento de exceções para comparações inválidas
5. Uso de métodos de representação (`__str__` e `__repr__`)
6. Aplicações práticas em um contexto de negócio
7. Integração com funções built-in do Python (`sorted()`, `max()`, `min()`)
8. Uso de list comprehension com comparações personalizadas
9. Demonstração de polimorfismo através das operações de comparação
10. Boas práticas de programação:
    - Validação de tipos
    - Mensagens de erro claras
    - Código legível e bem documentado
    - Comportamento consistente nas operações

Este exemplo serve como um modelo completo de como implementar e utilizar métodos mágicos de comparação em classes Python, seguindo as convenções da linguagem e proporcionando uma interface intuitiva para os usuários da classe.

4. Comportamento de Containers

Problema: Você está criando uma classe `Estoque` que precisa se comportar como uma lista ou dicionário, permitindo acesso aos itens por índice ou chave.

```
class Estoque:
    def __init__(self):
        self._items = {}

    def __getitem__(self, codigo):
        return self._items[codigo]

    def __setitem__(self, codigo, produto):
        self._items[codigo] = produto

    def __len__(self):
        return len(self._items)

# Uso:
estoque = Estoque()
estoque["A001"] = Produto("Notebook", 3500)
estoque["A002"] = Produto("Mouse", 50)

print(estoque["A001"])  # Saída: Notebook: R$ 3500.00
print(len(estoque))     # Saída: 2
```

Metáfora: Pense nos métodos de container como as regras de um armário com gavetas. O método `__getitem__` é como abrir uma gaveta específica, `__setitem__` é como colocar algo em uma gaveta, e `__len__` é como contar quantas gavetas estão ocupadas.

5. Iteração

Problema: Você quer criar uma classe que gere uma sequência de números Fibonacci e possa ser usada em um loop for.

```
class Fibonacci:
    def __init__(self, limite):
        self.limite = limite
        self.a, self.b = 0, 1
        self.contador = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.contador >= self.limite:
            raise StopIteration

        resultado = self.a
        self.a, self.b = self.b, self.a + self.b
        self.contador += 1
        return resultado

# Uso:
fib = Fibonacci(5)
for numero in fib:
    print(numero)  # Saída: 0, 1, 1, 2, 3
```

Metáfora: Imagine `__iter__` e `__next__` como um livro: `__iter__` é como abrir o livro na primeira página, e `__next__` é como virar para a próxima página até chegar ao fim.

6. Callable Objects

Problema: Você quer criar um objeto que possa ser chamado como uma função, por exemplo, um calculador de descontos que mantém um histórico.

```
class Desconto:
    def __init__(self):
        self.historico = []

    def __call__(self, valor, percentual):
        desconto = valor * (percentual / 100)
        self.historico.append((valor, percentual, desconto))
        return valor - desconto

    def mostrar_historico(self):
        for valor, perc, desc in self.historico:
            print(f"Valor: R${valor:.2f}, Desconto: {perc}%, Final:
R${valor-desc:.2f}")

# Uso:
calculador = Desconto()
preco_final = calculador(100, 20)  # Calcula 20% de desconto em 100
print(f"Preço com desconto: R${preco_final:.2f}")
calculador.mostrar_historico()
```

Metáfora: O método `__call__` transforma seu objeto em algo como um controle remoto universal - você pode programá-lo para executar uma série de ações com um único "clique" (chamada).

Dicas e Boas Práticas

1. **Consistência**: Mantenha a consistência nas operações. Se implementar `__add__`, considere implementar `__radd__` para suportar a operação reversa.
2. **Documentação**: Use docstrings para documentar o comportamento esperado dos métodos mágicos:

```
def __add__(self, outro):
    """
    Soma dois objetos.

    Args:
        outro: Objeto a ser somado com este

    Returns:
        Novo objeto com a soma dos valores
    """
    pass
```

3. **Validação**: Sempre valide os tipos dos argumentos em operações:

```
def __add__(self, outro):
    if not isinstance(outro, type(self)):
        raise TypeError(f"Não é possível somar {type(self)} com {type(outro)}")
    # ... resto do código
```

4. **Imutabilidade**: Considere fazer objetos imutáveis quando apropriado, retornando novos objetos em vez de modificar o existente.

Conclusão

Os métodos mágicos são uma ferramenta poderosa que permite que suas classes se comportem de maneira mais natural e pythônica. Eles permitem que você: - Defina representações claras de seus objetos - Implemente operações matemáticas intuitivas - Crie objetos que se comportam como containers - Torne seus objetos iteráveis - Customize praticamente qualquer aspecto do comportamento de seus objetos

Lembre-se: com grandes poderes vêm grandes responsabilidades. Use os métodos mágicos com sabedoria, mantendo seu código claro e intuitivo.

Recursos Adicionais

- https://dbader.org/blog/python-dunder-methods
- Real Python: Operator and Function Overloading in Custom Python Classes
- https://realpython.com/python-magic-methods/
- Documentação oficial do Python sobre métodos especiais
- Python Data Model