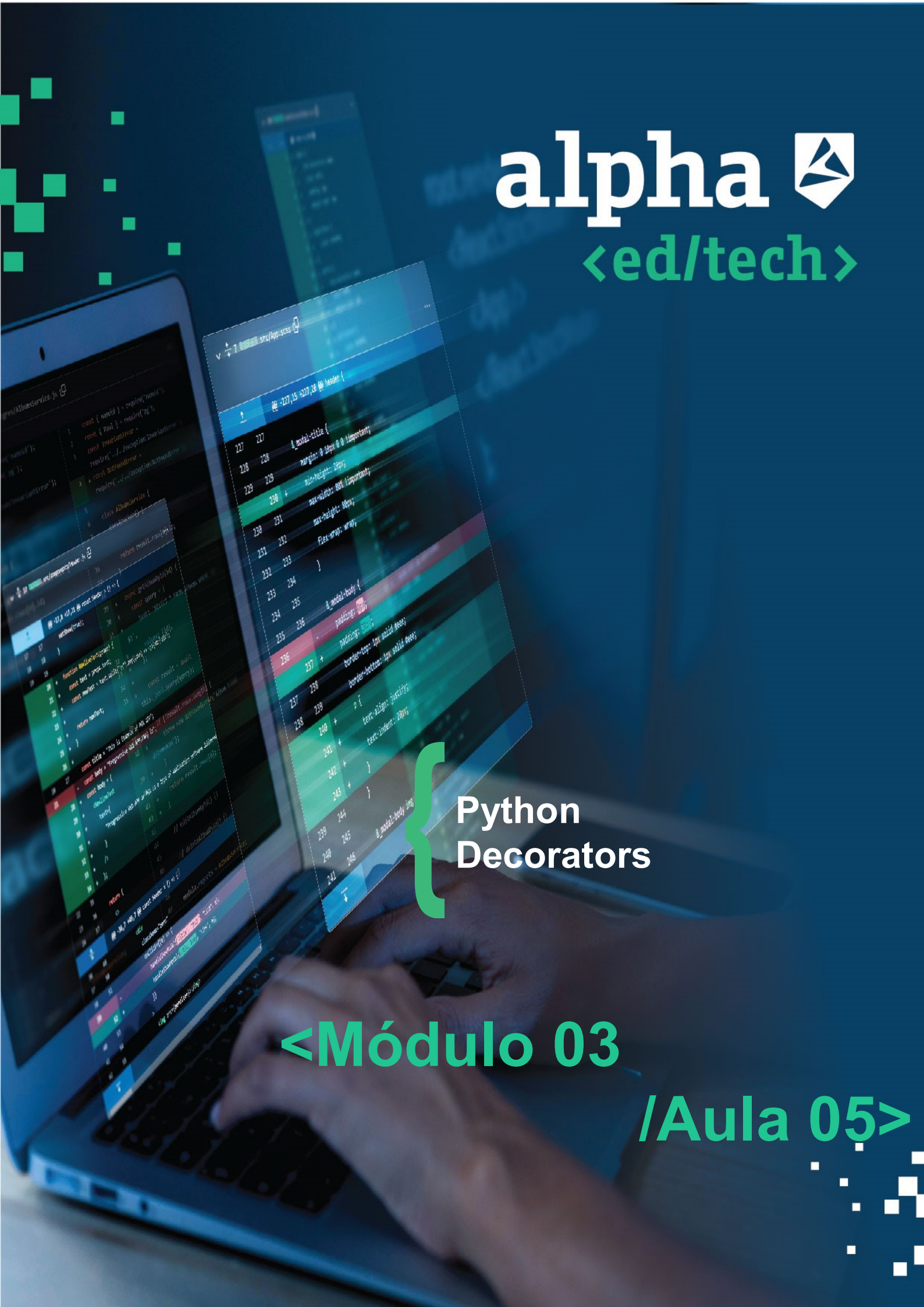
alpha
<ed/tech>
Python
Decorators
<Módulo 03
/Aula 05>

Decorators

Em Python, um **decorador**, ou **decorator**, é uma maneira elegante e poderosa de modificar ou **estender o comportamento de funções ou métodos sem alterar seu código interno**.
Em termos simples, um **decorator** é uma função que "envolve" outra função para adicionar alguma funcionalidade a ela.

Utilizar type hints quando estiver usando **decorator** pode ser complexo (**não recomendável**).

Vamos iniciar com uma função simples, para melhor entendimento:

```
def funcao_simples():
    print("Executando a função simples.")
```

Agora, vamos criar um **decorator** básico:

```
def meu_decorator(funcao):
    def wrapper():
        print("Antes da execução da função.")
        funcao()
        print("Depois da execução da função.")
    return wrapper
```

Neste exemplo, `meu_decorator` é um **decorator**.
Ele **recebe uma função como argumento** (`funcao`) e **retorna uma nova função** chamada `wrapper`, neste exemplo.
Esta nova função envolve a função original (`funcao`) e adiciona algum código antes e depois da sua execução.

Vamos utilizar o **decorator** na nossa função simples:

```
@meu_decorator
def funcao_simples():
    print("Executando a função simples.")
```

A linha `@meu_decorator` é uma maneira conveniente de aplicar o **decorator** à função `funcao_simples`.

Agora, quando chamamos `funcao_simples()`, o **decorator** entra em ação:

```
funcao_simples():

# Antes da execução da função.
# Executando a função simples.
# Depois da execução da função.
```

Isso mostra como o **decorator** alterou o comportamento da função original sem modificar seu código interno.

Decorators

Outro exemplo:

```
import time
def print_start(f):
    def new_f(x):
        print("Started...")
        return f(x)
    return new_f

def print_end(f):
    def new_f(x):
        y = f(x)
        print("Ended.")
        return y
    return new_f

@print_start
@print_end
def wait_and_multiply_by_2(x: float) -> float:
    time.sleep(5)
    return 2 * xx = wait_and_multiply_by_2(3)
```

Decorators são frequentemente utilizados para:

Logging: Adicionar logs antes e depois da execução de uma função.

Autenticação e Autorização: Verificar se o usuário tem permissão para executar uma função.

Medição de Desempenho: Medir quanto tempo uma função leva para ser executada.

Transformação de Resultados: Modificar ou processar o resultado de uma função.

Entender **decorators** é essencial para programadores Python avançados, pois eles proporcionam uma maneira flexível de estender e personalizar o comportamento de funções de forma limpa e eficaz.