Revisão de Python: Funções

> Funções são blocos de código reutilizáveis que realizam uma tarefa específica.

```
def saudacao(nome):
    print("Olá,", nome)
saudacao("Albert")
```

- Uma função é composta por um cabeçalho e um corpo.
- O cabeçalho contém o nome da função e seus parâmetros.
- O corpo contém as instruções que a função executa.
- A função é chamada com um argumento que é passado para o parâmetro.
- A função pode retornar um valor usando a instrução `return`.

Parâmetros

- Os parâmetros podem ser:
  - Obligatórios
  - Opcionais
  - Variáveis
  - Nomeados Variáveis

- **Obrigatórios:** Devem ser passados na chamada da função.

```
def saudacao(nome):
    print("Olá,", nome)
saudacao("Albert")
```

- **Opcionais:** Têm um valor padrão e podem ser omitidos na chamada da função.

```
def saudacao(nome="Mundo"):
    print("Olá,", nome)
saudacao()
```

- **Variáveis:** Podem receber um número variável de argumentos.

```
def saudacao(*nomes):
    for nome in nomes:
        print("Olá,", nome)
saudacao("Albert", "Maria", "João")
```

> Os parâmetros variáveis são passados como uma tupla.

- **Nomeados Variáveis:** Podem receber um número variável de argumentos nomeados.

```
def saudacao(**nomes):
    for nome, mensagem in nomes.items():
        print(mensagem + ",", nome)
saudacao(Albert="Bom dia", Maria="Boa tarde", João="Boa noite")
```

> Os parâmetros nomeados variáveis são passados como um dicionário.

Exemplo de Função com Parâmetros Variáveis

```
def saudacao(dia, *nomes, **mensagens):
    print("Hoje é", dia)
    for nome in nomes:
        print(mensagens.get(nome, "Olá") + ",", nome)
saudacao("Albert", "Maria", "João", Albert="Bom dia", Maria="Boa tarde", João="Boa noite")
```

Desdobramento de Sequências

- O operador `*` pode ser usado para desdobrar uma sequência em argumentos.

```
def saudacao(nome, sobrenome):
    print("Olá,", nome, sobrenome)
dados = ["Albert", "Muritiba"]
saudacao(*dados)
```

- O operador `**` pode ser usado para desdobrar um dicionário em argumentos nomeados.

```
def saudacao(nome, sobrenome):
    print("Olá,", nome, sobrenome)
dados = {"nome": "Albert", "sobrenome": "Muritiba"}
saudacao(**dados)
```

Regras de Sintaxe para Parâmetros

- Parâmetros obrigatórios devem vir antes dos opcionais.
- Parâmetros variáveis devem vir após os parâmetros obrigatórios e opcionais.
- Parâmetros nomeados variáveis devem vir após os parâmetros obrigatórios, opcionais e variáveis.

```
def saudacao(nome, sobrenome="Mundo", *nomes, **mensagens):
    pass
```

Retorno de Valores

- Uma função pode retornar um valor usando a instrução `return`.
- Uma função pode retornar múltiplos valores usando uma tupla.
- `return` interrompe a execução da função e retorna um valor.
- Se não houver uma instrução `return`, a função retorna `None`.

```
def soma(a, b):
    return a + b
print(soma(2, 3))

def divisao_e_resto(a, b):
    return a // b, a % b
print(divisao_e_resto(10, 3))
```

Retorno de Valores `yield`

- A instrução `yield` é usada para retornar um valor de uma função **geradora**.
- A instrução `yield` **suspende** a execução da função e retorna um valor.
- Quando a função é chamada novamente, a execução é retomada a partir do ponto onde foi suspensa.

```
def contador():
    yield 1
    yield 2
    yield 3
for i in contador():
    print(i)
```

Saída: `1 2 3`

Mais um exemplo de `yield`

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

for i in fibonacci(10):
    print(i)
```

Saída: `0 1 1 2 3 5 8 13 21 34`

- é possível usar um gerador sem usar um loop.

```
f = fibonacci(10)
print(next(f))
print(next(f))
print(next(f))
```

Saída: `0 1 1`

- é possível usar um gerador com um while.

```
f = fibonacci(10)
while True:
    try:
        print(next(f))
    except StopIteration:
        break
```

Saída: `0 1 1 2 3 5 8 13 21 34`

Comparação entre `return` e `yield`

```
def teste_return():
    l = []
    for i in range(1000000):
        l.append(i)
    return l

def teste_yield():
    for i in range(1000000):
        yield i
```

- `teste_return` retorna uma **lista** com um milhão de elementos e retorna a lista. A lista é criada na memória.
- `teste_yield` retorna um **gerador** que poderá gerar um milhão de números. O gerador não cria a lista na memória. Os valores são gerados sob demanda.

Geradores Usando Comprehension

- Compreensão pode ser usada para criar geradores.

```
g = (i for i in range(100) if i % 2 == 0)
print(next(g))
print(next(g))
print(next(g))
```

- Se for usado colchetes `[]` em vez de parênteses `()`, será criada uma lista em vez de um gerador.

Escopo de Variáveis

> O escopo de uma variável é a parte do programa onde a variável é acessível.

- Variáveis definidas dentro de uma função têm **escopo local**.

```
def funcao():
    x = 5
    print(x)
funcao()
print(x) # Erro
```

- Variáveis definidas fora de uma função têm **escopo global**.

```
x = 5
def funcao():
    print(x)
funcao()
print(x) # Saída: 5
```

- Variáveis locais têm precedência sobre variáveis globais.

```
x = 5
def funcao():
    x = 10
    print(x) # Saída: 10
funcao()
print(x) #Saída: 5
```

- Estruturas condicionais e de repetição **não** criam escopo local ao contrário do que ocorrem em outras linguagens de programação como C, Java e JavaScript. Em Python, o escopo local é criado apenas por funções e classes

```
y = 0
if True:
    y = 5
print(y) # Saída: 5
```

```
for i in range(5):
    z = 5
print(z) # Saída: 5
```

Acesso a Variáveis Globais

- Variáveis globais podem ser acessadas e modificadas dentro de uma função usando a instrução `global`.

```
x = 5
def funcao():
    print(x) # Erro

funcao()
```

```
x = 5
def funcao():
    global x
    print(x) # Saída: 5
    x = 10
funcao()
print(x) # Saída: 10
```

Por que evitar variáveis globais?

- **Poluição do Espaço de Nomes:** Variáveis globais podem ser acessadas e modificadas em qualquer lugar do programa, o que pode levar a erros difíceis de depurar.
- **Legibilidade:** Variáveis globais podem dificultar a compreensão do código, pois o comportamento de uma função pode depender do estado de variáveis globais.
- **Manutenção:** Variáveis globais podem dificultar a manutenção do código, pois o comportamento de uma função pode depender do estado de variáveis globais.

Escopo estático

- Python não tem um equivalente direto ao escopo estático, mas é possível simular criando uma variável de função que mantém seu valor entre chamadas.

```
def contador():
    contador.count += 1
    return contador.count
contador.count = 0
print(contador())
print(contador())
print(contador())
```

- No exemplo acima, a variável `contador.count` mantém seu valor entre chamadas da função `contador`.
- Embora `contador.count` seja uma variável global, seu nome associa-se à função `contador` evitando as desvantagens das variáveis globais.
- `yield` também pode ser usado para simular o mesmo comportamento.

Exercícios

1. Escreva uma função que receba um número e retorne se é primo ou não.
2. Usando a função do exercício anterior, escreva uma função que receba um número e retorne uma lista com todos os números primos até o número passado como argumento.
3. Altere a função do exercício anterior para que ela retorne um **gerador** em vez de uma lista.

Funções Lambda

> `lambdas` são funções anônimas e curtas. Elas podem receber qualquer número de parâmetros e são uma única expressão. Elas acessam apenas as variáveis em sua lista de parâmetros e no escopo global. Geralmente são expressões que transformam um valor.

```
soma = lambda a, b: a + b
print(soma(2, 3)) # Saída: 5
```

- As funções anônimas costumam ser usadas como argumentos para funções de ordem superior, como `map`, `filter` e `sorted`.

Funções como Objetos

> Em Python, as funções são objetos de primeira classe, o que significa que elas podem ser atribuídas a variáveis, passadas como argumentos e retornadas por outras funções.

```
def saudacao(nome):
    return "Olá, " + nome
f = saudacao
print(f("Mundo")) # Saída: Olá, Albert
```

- Funções podem ser atribuídas a variáveis, passadas como argumentos e retornadas por outras funções.

> Atenção ao nomear variáveis com o mesmo nome de funções embutidas, pois isso pode causar confusão. Exemplo: `list = [1, 2, 3]` e `list()`.

Funções de Ordem Superior

> Funções de ordem superior são funções que recebem outras funções como argumentos ou retornam funções.

- **Map:** Retorna um gerador que aplica uma função a cada item de um iterável.

```
def dobro(x):
    return 2 * x
print(list(map(dobro, [1, 2, 3, 4, 5]))) # Saída: [2, 4, 6, 8, 10]
```

- **Filter:** Retorna um gerador que filtra os itens de um iterável usando uma função.

```
def par(x):
    return x % 2 == 0
print(list(filter(par, [1, 2, 3, 4, 5]))) # Saída: [2, 4]
```

- **Sorted:** Ordena os itens de uma sequência.

```
print(sorted([1, 2, 3, 4, 5], key=lambda x: -x)) # Saída: [5, 4, 3, 2, 1]
```

- **Reduce:** Aplica uma função a pares de itens de um iterável até que reste apenas um item.

```
from functools import reduce
def soma(x, y):
    return x + y
print(reduce(soma, [1, 2, 3, 4, 5])) # Saída: 15
```

Dica de performance

- Mesmo para funções simples, é muito mais eficiente usar funções embutidas do que escrever funções personalizadas.
- Daí a importância de conhecer as funções embutidas do Python.
- As funções embutidas do Python são escritas em C e são muito mais rápidas que funções personalizadas escritas em Python.

Medindo o Tempo de Execução no Colab

- Colab fornece uma maneira prática de medir o tempo de execução de um comando ou de um bloco de código.
  - `%time` mede o tempo de execução de um comando em uma única linha.
  - `%%time` mede o tempo de execução de um bloco de código.
  - `%timeit` executa um comando várias vezes e mede o tempo médio de execução.
  - `%%timeit` executa um bloco de código várias vezes e mede o tempo médio de execução.

> Use `timeit` para medir o tempo de execução de funções muito rápidas, pois `time` pode não ser preciso o suficiente.

Comparação de Desempenho

```
%%timeit
soma = 0
for i in range(100000):
    soma += i*2
```

```
%%timeit
soma = sum(i*2 for i in range(100000))
```

```
%%timeit
soma = sum([i*2 for i in range(100000)])
```

```
%%timeit
soma = sum(map(lambda x: x*2, range(100000)))
```

Decoradores

> Decoradores são funções que envolvem outras funções para estender seu comportamento.

```
def decorador(funcao):
    def wrapper():
        print("Antes da função")
        funcao()
        print("Depois da função")
    return wrapper

def saudacao():
    print("Olá")

saudacao = decorador(saudacao)
saudacao()
```

- **Sintaxe de Açúcar:** Com o uso do `@`, é possível aplicar um decorador a uma função de forma mais simples.

```
def decorador(funcao):
    def wrapper(*args, **kwargs):
        print("Antes da função")
        funcao(*args, **kwargs)
        print("Depois da função")
    return wrapper

@decorador
def saudacao(nome):
    print("Olá,", nome)

saudacao()
```

> **Sintaxe de açúcar** é uma sintaxe mais curta e fácil de ler para uma operação que pode ser feita de outra forma.

- Exemplo de emprego de decoradores
  - **Logging:** Registrar informações sobre a execução de uma função.
  - **Tempo de Execução:** Medir o tempo de execução de uma função.
  - **Cache:** Armazenar o resultado de uma função para evitar cálculos repetidos.
  - **Autenticação:** Verificar se o usuário tem permissão para executar uma função.
  - **Validação:** Verificar se os argumentos de uma função são válidos.
  - **Tratamento de Erros:** Tratar exceções lançadas por uma função.
  - **Retentativas:** Tentar executar uma função novamente em caso de falha.
  - **Agendamento:** Agendar a execução de uma função para um horário específico.
  - Outros.

Tipo None

- `None` é um tipo de dado que representa a ausência de valor, usado para indicar que uma variável **não** tem um valor válido.
- `None` é retornado por funções que não têm uma instrução `return`.

```
def funcao():
    pass
print(funcao()) # Saída: None
```

- Verificar se uma variável é `None` é feito usando o operador de igualdade `==` ou `is`.

```
x = None
if x is None:
    print("x é None")
```

> O operador `is` é usado para verificar se duas variáveis se referem ao mesmo objeto na memória. Funciona com `None`, pois None é um objeto único (singleton).

- O tipo de `None` é `NoneType`.

```
print(type(None)) # Saída: <class 'NoneType'>
```

Exercícios

1. Usando lambda, escreva uma função que retorne o quadrado de um número.
2. Escreva um decorador que imprima quantas vezes uma função foi chamada.
3. Esceva um função que retorna o maior valor de uma lista. Se a lista for vazia, a função deve levantar uma exceção `ValueError`.
4. Crie um decorador para a função do exercício anterior que, ao ocorrer uma exceção, imprima uma mensagem de erro e retorne `None`.
5. Escreva um decorador que imprima a lista de argumentos e o resultado de uma função.