Funções e Estrutura de Programas em Python

Sumário

> Requisitos: Python 3.10 ou superior (para uso de type hints modernos com `|`)

1. Definindo Funções

Motivação

Funções são blocos de código reutilizáveis que executam uma tarefa específica. Elas permitem:
- Evitar repetição de código (princípio DRY: Don't Repeat Yourself)
- Organizar o código em unidades lógicas
- Facilitar manutenção e testes
- Melhorar a legibilidade do programa

Definição

Em Python, funções são definidas usando a palavra-chave `def`, seguida do nome da função, parênteses com parâmetros opcionais e dois pontos:

```
def nome_da_funcao(parametros):
    """Docstring opcional"""
    # Corpo da função
    # instrucoes
    return valor  # Opcional
```

Exemplos

```
# Função simples sem parâmetros
def saudacao():
    print("Olá, bem-vindo!")

saudacao()
# Saída: Olá, bem-vindo!

# Função com parâmetros
def saudar_pessoa(nome):
    print(f"Olá, {nome}!")

saudar_pessoa("Maria")
# Saída: Olá, Maria!

# Função com retorno
def soma(a, b):
    resultado = a + b
    return resultado

total = soma(5, 3)
print(total)
# Saída: 8
```

Exercícios

1. Crie uma função `calcular_area_retangulo` que recebe largura e altura e retorna a área.
2. Implemente uma função `eh_par` que verifica se um número é par.
3. Desenvolva uma função `contar_vogais` que conta o número de vogais em uma string.

Desafio

Crie uma função `fibonacci` que gera os primeiros n números da sequência de Fibonacci e os retorna como uma lista.

---

2. Parâmetros e Valores de Retorno

Motivação

Parâmetros permitem que funções sejam flexíveis e trabalhem com diferentes tipos dados. Valores de retorno permitem que funções produzam resultados que podem ser usados em outras partes do programa.

Definição

Parâmetros: Variáveis listadas na definição da função que recebem valores quando a função é chamada.

Argumentos: Valores reais passados para a função durante a chamada.

Retorno: Valor que a função devolve quando é chamada (usando `return`.)

Exemplos

```
# Múltiplos parâmetros
def calcular_media(nota1, nota2, nota3):
    media = (nota1 + nota2 + nota3) / 3
    return media

resultado = calcular_media(7.5, 8.0, 9.0)
print(f"Média: {resultado}")
# Saída: Média: 8.166666666666666

# Múltiplos valores de retorno
def operacoes_basicas(a, b):
    soma = a + b
    subtracao = a - b
    multiplicacao = a * b
    divisao = a / b if b != 0 else None
    return soma, subtracao, multiplicacao, divisao

s, sub, mult, div = operacoes_basicas(10, 5)
print(f"Soma: {s}, Subtração: {sub}")
# Saída: Soma: 15, Subtração: 5

# Função sem retorno explícito (retorna None)
def exibir_mensagem(msg):
    print(msg)
    # return None (implícito)

retorno = exibir_mensagem("Teste")
print(retorno)
# Saída: None
```

Boas Práticas

- Use nomes descritivos para parâmetros
- Documente o tipo esperado de cada parâmetro
- Retorne sempre o mesmo tipo de dado ou None, seja previsível

- Evite modificar objetos mutáveis recebidos como parâmetro (efeitos colaterais)

Exercícios

1. Crie uma função `converter_temperatura` que converte Celsius para Fahrenheit e Kelvin, retornando ambos os valores.
2. Implemente `calcular_desconto` que recebe preço e percentual de desconto, retornando o valor final.
3. Desenvolva `estatisticas_lista` que retorna mínimo, máximo e média de uma lista de números.

---

3. Argumentos Padrão e Argumentos Nomeados

Motivação

Argumentos padrão tornam funções mais flexíveis, permitindo omitir valores para parâmetros menos usados. Argumentos nomeados melhoram a legibilidade ao deixar explícito qual valor corresponde a qual parâmetro.

Definição

Argumentos Padrão: Valores pré-definidos para parâmetros que serão usados se nenhum argumento for fornecido.

Argumentos Nomeados: Passagem de argumentos especificando explicitamente o nome do parâmetro.

Exemplos

```
# Argumentos padrão
def criar_perfil(nome, idade, cidade="Não informada", profissao="Não informada"):
    return f"{nome}, {idade} anos, {cidade}, {profissao}"

print(criar_perfil("João", 30))
# Saída: João, 30 anos, Não informada, Não informada

print(criar_perfil("Maria", 25, "São Paulo"))
# Saída: Maria, 25 anos, São Paulo, Não informada

# Argumentos nomeados
print(criar_perfil("Pedro", 28, profissao="Engenheiro", cidade="Rio de Janeiro"))
# Saída: Pedro, 28 anos, Rio de Janeiro, Engenheiro

# Misturando posicionais e nomeados
print(criar_perfil("Ana", 35, cidade="Curitiba"))
# Saída: Ana, 35 anos, Curitiba, Não informada
```

Armadilhas Comuns

```
# ERRO: Valor mutável como padrão
def adicionar_item(item, lista=[]):  # EVITE ISSO!
    lista.append(item)
    return lista

print(adicionar_item(1))  # [1]
print(adicionar_item(2))  # [1, 2] - Compartilha a mesma lista!

# CORRETO: Use None como padrão
def adicionar_item_correto(item, lista=None):
    if lista is None:
        lista = []
    lista.append(item)
    return lista

print(adicionar_item_correto(1))  # [1]
print(adicionar_item_correto(2))  # [2] - Nova lista
```

Boas Práticas

- Coloque parâmetros sem padrão antes dos com padrão. Caso contrário, ocorre um erro de sintaxe.
- Nunca use listas, dicionários ou outros mutáveis como valores padrão
- Use argumentos nomeados para funções com muitos parâmetros
- Considere usar argumentos nomeados quando a ordem não é óbvia

Exercícios

1. Crie `formatar_data` que aceita dia, mês, ano e um separador padrão "/".
2. Implemente `calcular_preco_final` com preço base, desconto padrão de 0% e taxa de entrega padrão de 0.
3. Desenvolva `enviar_email` com destinatário, assunto, corpo e cc/cco opcionais.

---

4. *args e *kwargs

Motivação

Às vezes não sabemos quantos argumentos uma função receberá. `*args` e `**kwargs` permitem criar funções com número variável de argumentos, tornando-as mais flexíveis e genéricas.

Definição

`*args`: Coleta argumentos posicionais adicionais em uma tupla.

`**kwargs`: Coleta argumentos nomeados adicionais em um dicionário.

Exemplos

```
# *args - argumentos posicionais variáveis
def somar_todos(*numeros):
    total = 0
    for num in numeros:
        total += num
    return total

print(somar_todos(1, 2, 3))        # Saída: 6
print(somar_todos(10, 20, 30, 40)) # Saída: 100
print(somar_todos())               # Saída: 0

# **kwargs - argumentos nomeados variáveis
def exibir_informacoes(**dados):
    for chave, valor in dados.items():
        print(f"{chave}: {valor}")

exibir_informacoes(nome="João", idade=30, cidade="São Paulo")
# Saída:
# nome: João
# idade: 30
# cidade: São Paulo

# Combinando parâmetros normais, *args e **kwargs
def funcao_complexa(obrigatorio, opcional="padrão", *args, **kwargs):
    print(f"Obrigatório: {obrigatorio}")
    print(f"Opcional: {opcional}")
    print(f"Args adicionais: {args}")
    print(f"Kwargs adicionais: {kwargs}")

funcao_complexa(1, 2, 3, 4, 5, nome="Maria", idade=25)
# Saída:
# Obrigatório: 1
# Opcional: 2
# Args adicionais: (3, 4, 5)
# Kwargs adicionais: {'nome': 'Maria', 'idade': 25}
```

Desempacotamento

```
# Desempacotando listas/tuplas com *
def multiplicar(a, b, c):
    return a * b * c

valores = [2, 3, 4]
resultado = multiplicar(*valores)  # Equivale a multiplicar(2, 3, 4)
print(resultado)  # Saída: 24

# Desempacotando dicionários com **
def criar_usuario(nome, idade, cidade):
    return f"{nome}, {idade} anos, vive em {cidade}"

dados = {"nome": "Pedro", "idade": 28, "cidade": "Brasília"}
usuario = criar_usuario(**dados)
print(usuario)  # Saída: Pedro, 28 anos, vive em Brasília
```

Aplicações

- Encapsular chamadas de função (wrappers e decoradores)
- Implementar funções que aceitam configurações variáveis
- Passar argumentos dinamicamente entre funções
- Criar APIs flexíveis

Exercícios

1. Crie uma função que receba uma quantidade variável de números via *args e retorne um dicionário (*kwargs não é permitido na chamada) contendo: quantidade de valores, soma, média e maior número.
2. Crie uma função que receba um nome obrigatório e qualquer quantidade de pares chave=valor via **kwargs representando atributos (ex.: idade, profissão, cidade) e retorne uma frase formatada descrevendo a pessoa.
3. Crie uma função que receba *args com valores mistos (int, float, str) e* *kwargs com regras (ex.: somar=True, concatenar=True) e retorne apenas as operações ativadas nas regras, ignorando os demais dados.

Desafio

Implemente um decorador genérico `log_chamada` que registra nome da função, argumentos posicionais e nomeados de qualquer função decorada.

---

5. Escopo (Local, Global, Nonlocal)

Motivação

Entender escopo é fundamental para evitar bugs envolvendo leitura e modificação de variáveis. O escopo define onde uma variável pode ser acessada e onde pode ser alterada.

Definição

Escopo Local: Variáveis definidas dentro de uma função, acessíveis apenas dentro da função.\
Escopo Global: Variáveis definidas no nível do módulo, acessíveis em todo o arquivo.\
Escopo Nonlocal: Variáveis de funções externas em funções aninhadas.

Regra LEGB

Python busca variáveis na seguinte ordem:
- **Local**: Dentro da função atual
- **Enclosing**: Funções externas envolventes
- **Global**: Nível do módulo
- **Built-in**: Nomes internos do Python

Exemplos

```
# Escopo local vs global
x = 10  # Global

def funcao():
    x = 5  # Local (não afeta a global)
    print(f"Local x: {x}")

funcao()  # Saída: Local x: 5
print(f"Global x: {x}")  # Saída: Global x: 10

# Usando global para modificar variável global
contador = 0

def incrementar():
    global contador
    contador += 1

incrementar()
incrementar()
print(contador)  # Saída: 2

# Escopo nonlocal em funções aninhadas
def externa():
    x = "externa"

    def interna():
        nonlocal x
        x = "modificada pela interna"
        print(f"Interna: {x}")

    print(f"Antes: {x}")
    interna()
    print(f"Depois: {x}")

externa()
# Saída:
# Antes: externa
# Interna: modificada pela interna
# Depois: modificada pela interna
```

Armadilhas Comuns

```
# Erro comum: tentar modificar global sem declarar
total = 0

def adicionar(valor):
    # total += valor  # UnboundLocalError!
    # Python trata 'total' como local porque há atribuição
    pass

# Solução: declarar global
def adicionar_correto(valor):
    global total
    total += valor

adicionar_correto(10)
print(total)  # Saída: 10
```

Boas Práticas

- Evite usar `global` sempre que possível
- Prefira retornar valores em vez de modificar variáveis globais
- Use classes para gerenciar estado compartilhado
- Limite o uso de `nonlocal` a casos específicos (closures, decoradores)
- Mantenha funções puras quando possível (sem efeitos colaterais)

Exercícios

1. Crie um contador usando closure e `nonlocal` em vez de variável global.
2. Implemente uma função que demonstra a diferença entre escopo local e global.
3. Desenvolva um sistema de configuração que usa `global` apropriadamente.

Desafio

Implemente uma função `criar_acumulador` que retorna uma função que acumula valores chamada após chamada, usando closure.

6. Funções Lambda

Motivação

Funções lambda são funções anônimas compostas por uma única expressão, úteis para operações simples que serão usadas uma única vez, especialmente como argumentos para funções de ordem superior.

Definição

Uma função lambda é definida com a sintaxe:

```
lambda parametros: expressao
```

Equivale a:

```
def funcao_anonima(parametros):
    return expressao
```

Exemplos

```
# Lambda simples
quadrado = lambda x: x ** 2
print(quadrado(5))  # Saída: 25

# Equivalente com def
def quadrado_def(x):
    return x ** 2

# Lambda com múltiplos parâmetros
somar = lambda a, b: a + b
print(somar(3, 7))  # Saída: 10

# Uso comum: com sorted()
pessoas = [
    {"nome": "João", "idade": 30},
    {"nome": "Maria", "idade": 25},
    {"nome": "Pedro", "idade": 35}
]

# Ordenar por idade
ordenadas = sorted(pessoas, key=lambda p: p["idade"])
print([p["nome"] for p in ordenadas])  # Saída: ['Maria', 'João', 'Pedro']

numeros = [1, 2, 3, 4, 5]

# Uso com map()
dobrados = list(map(lambda x: x * 2, numeros))
print(dobrados)  # Saída: [2, 4, 6, 8, 10]

# Uso com filter()
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # Saída: [2, 4]

# Uso com reduce() (necessita importar)
from functools import reduce
produto = reduce(lambda x, y: x * y, numeros)
print(produto)  # Saída: 120
```

> ***Nota**: `reduce()` geralmente é menos legível que loops ou `sum()`.*
> *Use apenas quando a intenção ficar clara para quem lê o código.*

Limitações

- Apenas uma expressão (não pode conter statements como loops completos ou `try/except`; expressões condicionais são permitidas)
- Menos legível para lógica complexa
- Dificulta debugging (sem nome para rastreamento)
- Não pode conter docstrings

Quando Usar

Use lambda quando:
- A função é simples (uma linha)
- Será usada uma única vez
- É argumento para funções como `map`, `filter`, `sorted`

Use `def` quando:
- A função é complexa
- Será reutilizada
- Precisa de documentação
- Contém múltiplas operações

Exemplos

```
# Lambda com condicional (expressão ternária)
absoluto = lambda x: x if x >= 0 else -x
print(absoluto(-5))  # Saída: 5

# Lista de lambdas
operacoes = [
    lambda x: x + 1,
    lambda x: x * 2,
    lambda x: x ** 2
]

numero = 3
for operacao in operacoes:
    print(operacao(numero))  # Saída: 4, 6, 9

# Lambda em compreensão de lista
transformar = lambda f, lista: [f(x) for x in lista]
resultado = transformar(lambda x: x ** 3, [1, 2, 3, 4])
print(resultado)  # Saída: [1, 8, 27, 64]
```

Exercícios

1. Use lambda com `sorted()` para ordenar strings por comprimento.
2. Crie uma lista de funções lambda que calculam potências de 2 (2^1, 2^2, 2^3...).
3. Use `filter()` com lambda para remover valores None de uma lista.

Desafio

Implemente uma calculadora básica usando um dicionário de lambdas para as operações.

---

7. Docstrings e Type Hints (PEP 484)

Motivação

Docstrings e type hints deixam o comportamento explícito e ajudam ferramentas a identificar erros antes da execução.

Definição

Docstrings: Strings de documentação que descrevem o que uma função faz, seus parâmetros e retorno.

Type Hints: Anotações que indicam os tipos esperados de parâmetros e retorno.

Exemplos de Docstrings

```
def calcular_area_triangulo(base, altura):
    """
    Calcula a área de um triângulo.

    Args:
        base (float): A base do triângulo
        altura (float): A altura do triângulo

    Returns:
        float: A área do triângulo

    Raises:
        ValueError: Se base ou altura forem negativos

    Examples:
        >>> calcular_area_triangulo(10, 5)
        25.0
        >>> calcular_area_triangulo(7.5, 4)
        15.0
    """
    if base < 0 or altura < 0:
        raise ValueError("Base e altura devem ser não-negativos")
    return (base * altura) / 2

# Acessando docstring
print(calcular_area_triangulo.__doc__)
help(calcular_area_triangulo)
```

Formatos de Docstring

```
# Estilo Google
def funcao_google(param1, param2):
    """
    Descrição breve da função.

    Descrição mais detalhada, se necessário.

    Args:
        param1 (int): Descrição do primeiro parâmetro
        param2 (str): Descrição do segundo parâmetro

    Returns:
        bool: Descrição do retorno
    """
    pass

# Estilo NumPy
def funcao_numpy(param1, param2):
    """
    Descrição breve da função.

    Descrição mais detalhada.

    Parameters
    ----------
    param1 : int
        Descrição do primeiro parâmetro
    param2 : str
        Descrição do segundo parâmetro

    Returns
    -------
    bool
        Descrição do retorno
    """
    pass

# Estilo reStructuredText (Sphinx)
def funcao_rst(param1, param2):
```

```
    """
    Descrição breve da função.

    :param param1: Descrição do primeiro parâmetro
    :type param1: int
    :param param2: Descrição do segundo parâmetro
    :type param2: str
    :return: Descrição do retorno
    :rtype: bool
    """
    pass
```

Type Hints (PEP 484)

```
# Tipos básicos
def saudar(nome: str) -> str:
    return f"Olá, {nome}!"

def somar(a: int, b: int) -> int:
    return a + b

def calcular_media(notas: list[float]) -> float:
    return sum(notas) / len(notas)

# Tipos opcionais
from typing import Optional

def encontrar_usuario(id: int) -> Optional[str]:
    """Retorna o nome do usuário ou None se não encontrado."""
    usuarios = {1: "João", 2: "Maria"}
    return usuarios.get(id)

# Union types (múltiplos tipos possíveis)
from typing import Union

def processar_entrada(valor: Union[int, str]) -> str:
    if isinstance(valor, int):
        return f"Número: {valor}"
    return f"Texto: {valor}"

# Ou usando o operador | (Python 3.10+)
def processar_entrada_moderno(valor: int | str) -> str:
    if isinstance(valor, int):
        return f"Número: {valor}"
    return f"Texto: {valor}"

# Tipos complexos
from typing import Dict, Tuple, Callable, Optional

def processar_dados(
    dados: list[dict[str, int | str]]
) -> tuple[int, float]:
    """
```

```
    Processa uma lista de dicionários.

    Args:
        dados: Lista de dicionários com dados mistos

    Returns:
        Tupla com (total_registros, media_valores)
    """
    total = len(dados)
    valores = [d.get('valor', 0) for d in dados if isinstance(d.get('valor'), int)]
    media = sum(valores) / len(valores) if valores else 0.0
    return total, media

# Type hints para funções (callbacks)
def aplicar_operacao(
    numeros: List[int],
    operacao: Callable[[int], int]
) -> List[int]:
    """Aplica uma operação a cada número da lista."""
    return [operacao(n) for n in numeros]

resultado = aplicar_operacao([1, 2, 3, 4], lambda x: x ** 2)
print(resultado)  # Saída: [1, 4, 9, 16]
```

Type Hints Avançados

```
from typing import TypeVar, Generic, Protocol, List, Optional

# TypeVar para genéricos
T = TypeVar('T')

def primeiro_elemento(lista: List[T]) -> Optional[T]:
    """Retorna o primeiro elemento da lista ou None."""
    return lista[0] if lista else None

# Protocol para duck typing estruturado
class Desenhavel(Protocol):
    def desenhar(self) -> None: ...

def renderizar(obj: Desenhavel) -> None:
    obj.desenhar()

# Classes genéricas
class Pilha(Generic[T]):
    def __init__(self) -> None:
        self._items: List[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

pilha_int: Pilha[int] = Pilha()
pilha_int.push(1)
pilha_int.push(2)
```

Ferramentas para Type Checking

```
# mypy - verificador de tipos estático
pip install mypy
mypy seu_arquivo.py

# pyright - verificador da Microsoft
pip install pyright
pyright seu_arquivo.py
```

Boas Práticas

Docstrings:
- Sempre inclua docstring em funções públicas
- Comece com uma linha resumida
- Documente todos os parâmetros e retorno
- Inclua exemplos quando útil
- Documente exceções levantadas
- Use um estilo consistente em todo o projeto

Type Hints:
- Use type hints em funções públicas de APIs
- Seja específico (use `List[int]` em vez de `list`)
- Prefira `Optional[T]` quando None é possível
- Use `Any` com moderação
- Combine com docstrings para documentação completa
- Execute verificadores de tipo no CI/CD

Exercícios

1. Adicione docstrings completas (estilo Google) a uma função existente.
2. Implemente uma função com type hints incluindo `Optional` e `Union`.
3. Crie uma função genérica usando `TypeVar` que funciona com qualquer tipo.

Desafio

Desenvolva um módulo completo com múltiplas funções, todas com docstrings detalhadas e type hints corretos. Execute mypy para verificar a correção dos tipos.

Aplicações Práticas

1. Sistema de Validação

```
from typing import Callable, Any, List, Tuple, Optional

def validar_entrada(
    valor: Any,
    validadores: List[Callable[[Any], bool]],
    mensagens_erro: List[str]
) -> Tuple[bool, Optional[str]]:
    if len(validadores) != len(mensagens_erro):
        raise ValueError("validadores e mensagens_erro devem ter o mesmo tamanho")

    for validador, mensagem in zip(validadores, mensagens_erro):
        if not validador(valor):
            return False, mensagem

    return True, None


# Uso
idade = 15
valido, erro = validar_entrada(
    idade,
    [
        lambda x: isinstance(x, int),
        lambda x: x >= 18,
        lambda x: x <= 120
    ],
    [
        "Idade deve ser um número inteiro",
        "Idade deve ser pelo menos 18",
        "Idade deve ser no máximo 120"
    ]
)

if not valido:
    print(f"Erro: {erro}")
```

2. Decorador de Cache

```
from functools import wraps
from typing import Callable, Any, Dict, Tuple


def cache(func: Callable) -> Callable:
    """
    Decorador que cacheia resultados de função.
    Funciona apenas com argumentos hashable.
    Não é thread-safe.
    """
    cache_dict: Dict[Tuple[tuple, tuple], Any] = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        chave = (args, tuple(sorted(kwargs.items())))
        if chave not in cache_dict:
            cache_dict[chave] = func(*args, **kwargs)
        return cache_dict[chave]

    return wrapper


@cache
def fibonacci(n: int) -> int:
    """Calcula o n-ésimo número de Fibonacci."""
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


if __name__ == "__main__":
    print(fibonacci(100))  # Rápido graças ao cache
```

3. Pipeline de Processamento

```
from typing import Callable, List, TypeVar

T = TypeVar('T')

def pipeline(*funcoes: Callable[[T], T]) -> Callable[[T], T]:
    """
    Cria um pipeline de funções.

    Args:
        *funcoes: Funções a serem aplicadas em sequência

    Returns:
        Função que aplica todas as transformações
    """
    def processar(valor: T) -> T:
        resultado = valor
        for funcao in funcoes:
            resultado = funcao(resultado)
        return resultado
    return processar

# Uso
processar_texto = pipeline(
    str.lower,
    str.strip,
    lambda s: s.replace(" ", "_")
)

print(processar_texto("  Olá Mundo  "))  # Saída: olá_mundo
```

Resumo de Boas Práticas

1. **Nomeação**: Use nomes descritivos e verbos para funções.

2. **Tamanho**: Mantenha funções pequenas e focadas (princípio da responsabilidade única).
3. **Parâmetros**: Limite a 3-5 parâmetros; use objetos ou `**kwargs` para mais.
4. **Retorno**: Seja consistente com tipos de retorno.
5. **Efeitos colaterais**: Evite alterar estado global; retorne valores sempre que possível.
6. **Documentação**: Sempre documente APIs públicas.
7. **Type hints**: Use em código compartilhado e APIs públicas.
8. **DRY**: Não repita código; use abstrações (funções, classes, módulos).
9. **Testabilidade**: Escreva funções que sejam fáceis de testar.
10. **Composição**: Componha funções pequenas para criar comportamentos complexos.

Recursos Adicionais

- PEP 8: Guia de estilo Python
- PEP 257: Convenções de Docstring
- PEP 484: Type Hints
- PEP 526: Syntax for Variable Annotations
- Documentação oficial Python: https://docs.python.org/3/
- Real Python: https://realpython.com/
- Python Type Checking Guide: https://mypy.readthedocs.io/