Introdução - Funções em Python

Informações do Autor

- **Nome:** ANIMA Lab
- **Data de atualização**: 16/09/2025

---

1) O que é uma função?

Uma **função** é um bloco de código reutilizável que recebe dados de entrada (opcional), realiza uma tarefa e **retorna um resultado** (também opcional).

Na vida real:

- Um liquidificador é como uma função → você coloca frutas (entrada), ele processa (código) e devolve um suco (saída).

Em programação, funções ajudam a:

- **Organizar** o código.
- **Reutilizar** lógica em diferentes partes do programa.
- **Reduzir repetição** de código.

---

2) Criando funções no Python

Em programação, uma função é como uma receita de bolo:

- Você escreve a receita **uma vez** (define a função).
- Depois, pode usá-la **sempre que quiser** (chamar a função).

No Python, usamos a palavra-chave **def** para criar funções.

Estrutura mais simples de uma função

```
def nome_da_funcao():
    # bloco de código
    # aqui ficam as instruções que a função executa
```

- **`def`** → vem de *define* (“definir”).
- **`nome_da_funcao`** → escolha um nome descritivo (ex.: `saudacao`, `mostrar_boas_vindas`).

- `()` → por enquanto está vazio (sem parâmetros).
- **bloco de código** → é o que a função faz (exibir uma mensagem, realizar um cálculo etc.).

A importância da indentação

No Python, a indentação (recuo à direita usando espaços) **define o que faz parte do bloco da função**. Não usamos `{ }` como em outras linguagens.

Exemplo correto:

```
def saudacao():
    print("Olá, bem-vindo à aula de Python!")
    print("Essa mensagem também faz parte da função")
```

Exemplo incorreto (gera erro de indentação):

```
def saudacao():
print("Olá, bem-vindo à aula de Python!")  # ERRO: sem indentação
```

Por padrão, usamos 4 espaços de indentação para cada nível dentro da função.

Exemplo – função de saudação

```
def saudacao():
    print("Olá, bem-vindo à aula de Python!")

saudacao()  # chamada da função
```

Saída:

```
Olá, bem-vindo à aula de Python!
```

- Aqui, a função não recebe nenhum valor e não devolve nada.
- Ela apenas executa a ação de mostrar uma mensagem.
- O `print` faz parte do bloco porque está indentado.

Resumo

- Criamos funções no Python usando `def nome_da_funcao():`.
- O bloco da função deve ser **indentado** (4 espaços).

- Funções podem executar ações mesmo sem receber parâmetros ou devolver valores.
- Chamamos a função escrevendo apenas seu **nome()**.

---

3) Funções com parâmetros

Até agora vimos funções que fazem sempre a mesma coisa, como imprimir uma mensagem fixa. Mas e se quisermos que a função funcione de forma mais flexível, adaptando-se a diferentes situações?

Para isso usamos os **parâmetros**: valores que podemos **passar para dentro da função** no momento em que a chamamos.

O que são parâmetros?

- São como caixinhas temporárias dentro da função.
- Guardam os valores que passamos quando chamamos a função.
- Fazem a função trabalhar com dados diferentes sem precisar reescrever o código.

Exemplo 1 – saudação personalizada

```
def saudacao(nome):
    print("Olá,", nome, "! Bem-vindo à aula de Python!")

saudacao("Maria")
saudacao("João")
```

Saída:

```
Olá, Maria ! Bem-vindo à aula de Python!
Olá, João ! Bem-vindo à aula de Python!
```

O parâmetro `nome` recebe o valor `"Maria"` na primeira chamada e `"João"` na segunda. Assim, a mesma função se adapta para cada pessoa.

Exemplo 2 – cálculo de idade de um idoso

Imagine que queremos calcular a idade a partir do ano de nascimento:

```
def calcula_idade(ano_nascimento, ano_atual):
    idade = ano_atual - ano_nascimento
    print("Idade:", idade, "anos")

calcula_idade(1950, 2025)
calcula_idade(1943, 2025)
```

Saída:

```
Idade: 75 anos
Idade: 82 anos
```

Exemplo 3 – parâmetros em ciência de dados

Em um questionário com idosos, cada um informou quantos minutos de caminhada por semana realiza. Vamos escrever uma função que recebe esse valor e classifica o nível de atividade:

```
def classifica_atividade(minutos):
    if minutos < 150:
        print("Atividade insuficiente")
    else:
        print("Atividade adequada")

classifica_atividade(100)
classifica_atividade(200)
```

Saída:

```
Atividade insuficiente
Atividade adequada
```

Aqui o parâmetro `minutos` permite que a função seja usada para qualquer idoso, sem precisar reescrever o código.

Quando usar parâmetros?

- Quando a função precisa trabalhar com dados variáveis.
- Quando queremos reutilizar a mesma lógica** para diferentes entradas.
- Em ciência de dados, praticamente todas as funções vão receber parâmetros, porque lidamos com **dados que mudam** (listas, números,

respostas de questionários, etc.).

Resumo

- Parâmetros tornam funções flexíveis e reutilizáveis.
- Podemos ter um ou vários parâmetros.
- Eles permitem que a função trabalhe com diferentes entradas a cada chamada.

---

4) Funções com retorno

Uma função pode **calcular um valor** e **devolver esse resultado** usando a palavra-chave `return`.

A grande diferença é:

- **`print`** → apenas mostra algo na tela.
- **`return`** → devolve o valor para que possa ser usado em outra parte do programa.

Exemplo 1 – função que retorna o quadrado

```
def quadrado(x):
    return x * x

print(quadrado(4))   # 16
print(quadrado(10))  # 100
```

Explicando:

- Quando chamamos `quadrado(4)`, a função calcula `4 * 4` e devolve 16.
- O `print` na chamada é quem exibe o valor retornado.
- A função por si só não imprime nada.

Exemplo 2 – diferença entre `print` e `return`

```python
def quadrado_print(x):
    print(x * x)   # mostra o resultado, mas não devolve

def quadrado_return(x):
    return x * x  # devolve o valor

# Usando as funções
a = quadrado_print(5)   # mostra 25, mas a variável a recebe None
b = quadrado_return(5)  # devolve 25, a variável b recebe 25

print("a =", a)
print("b =", b)
```

Saída:

```
25
a = None
b = 25
```

- `quadrado_print` não devolveu nada, apenas exibiu na tela.
- `quadrado_return` devolveu 25, que pôde ser armazenado em `b` e usado depois.

Exemplo 3 – média de idades de idosos

Funções com `return` são essenciais em ciência de dados, pois devolvem resultados que serão usados em cálculos posteriores

```python
def media_idades(idades):
    return sum(idades) / len(idades)

grupo = [60, 70, 75, 80, 90]
resultado = media_idades(grupo)

print("Média das idades:", resultado)
```

Saída:

```
Média das idades: 75.0
```

- Aqui a função devolve a média e podemos guardar esse valor em `resultado` para usar depois em outros cálculos.

Exemplo 4 – classificação com retorno

Podemos fazer uma função que devolva um texto de classificação em vez de apenas imprimir.

```
def classifica_idade(idade):
    if idade >= 60:
        return "Idoso"
    else:
        return "Não idoso"

print(classifica_idade(72))  # Idoso
print(classifica_idade(40))  # Não idoso
```

- O `return` permite que o programa decida o que fazer com esse resultado:
  - exibir,
  - salvar em uma variável,
  - ou usar dentro de outro cálculo.

Quando usar `return`?

- Quando precisamos **guardar o resultado** para uso futuro.
- Quando a função faz parte de um cálculo maior.
- Quando queremos que a função seja mais flexível (ela só devolve o valor, quem chamou decide o que fazer com ele).

Resumo

- `print` mostra o valor na tela, mas não devolve nada.
- `return` devolve o valor, permitindo guardar em variáveis ou usar em cálculos.
- Funções com `return` são importantes em ciência de dados, pois todo o tempo precisamos calcular e reutilizar resultados.

---

5) Por que usar funções?

Quando começamos a programar, é comum escrever o mesmo código várias vezes. Mas conforme os programas crescem, isso gera bagunça e erros difíceis de corrigir.

As funções resolvem esse problema, porque permitem organizar, reutilizar e isolar partes do código.

Vantagens principais:

1. **Evitar repetição (reuso de código)**
    - Em vez de escrever 10 vezes a mesma lógica, escrevemos uma vez em uma função e chamamos quando precisar.
    - Isso economiza tempo e reduz chances de erro.

    Exemplo:

    ```
    def calcula_imc(peso, altura):
        return peso / (altura * altura)
    ```

    Agora podemos usar para qualquer idoso:

    ```
    print(calcula_imc(70, 1.70))
    print(calcula_imc(80, 1.65))
    ```

2. **Clareza e organização**
    - Um programa longo pode ser dividido em módulos pequenos (funções), cada um responsável por uma parte.
    - Isso torna o código mais legível.
3. **Testes mais fáceis**
    - Se cada função faz uma tarefa simples, fica fácil testar individualmente para garantir que está correta.
    - Em ciência de dados, isso é essencial para validar cálculos.
4. **Colaboração em equipe**
    - Várias pessoas podem trabalhar no mesmo projeto:
        - uma cuida da função de limpar dados,
        - outra da função de calcular estatísticas,
        - outra da função de gerar gráficos.
    - Isso facilita projetos em grupo e desenvolvimento profissional.

Resumo: funções tornam o código mais limpo, reutilizável, confiável e colaborativo.

---

6) Programação Imperativa x Funcional

Até agora usamos um estilo chamado **imperativo**: damos ordens ao computador passo a passo.

A **programação funcional** é outro jeito de pensar:

- Foca em **funções matemáticas puras**, que sempre devolvem o mesmo resultado para a mesma entrada.
- Evita **efeitos colaterais**, ou seja, não muda variáveis externas de forma inesperada.

Exemplo funcional (bom)

```
def incrementa(x):
    return x + 1

print(incrementa(5))  # 6
print(incrementa(5))  # 6 (sempre o mesmo resultado)
```

- `incrementa(5)` sempre devolve 6.
- Não depende de nada além do valor de entrada `x`.
- Isso torna o comportamento **previsível e confiável**.

Exemplo com efeito colateral (ruim)

```
valor = 1

def incrementa(x):
    return x + valor

print(incrementa(5))  # 6
valor = 2
print(incrementa(5))  # 7 (resultado mudou!)
```

- O resultado muda porque depende da variável externa `valor`.
- Isso dificulta testes e pode causar erros inesperados.

Esse tipo de função não é referencialmente transparente, pois o resultado não depende só da entrada.

Relação com Ciência de Dados

Em ciência de dados, queremos funções que sejam **confiáveis e repetíveis**:

- Uma função que calcula a média de idades deve sempre dar o mesmo resultado para os mesmos dados.
- Se ela depender de variáveis externas (como um “valor global”), podemos ter resultados inconsistentes em análises.

Resumo

- Programação **imperativa** → foco em ordens passo a passo.
- Programação **funcional** → foco em funções puras, sem efeitos colaterais.
- Em ciência de dados, preferimos funções funcionais, porque garantem **reprodutibilidade e confiança nos resultados**.

---

7) Vantagens das funções (e do estilo funcional)

As funções trazem várias vantagens, principalmente quando pensamos em programas maiores ou em análise de dados.

Menos código repetido

- Ao criar uma função, podemos reutilizar a mesma lógica em diferentes partes do programa.
- Isso evita copiar e colar o mesmo trecho várias vezes.

Exemplo:

```
def calcula_media(valores):
    return sum(valores) / len(valores)

idades = [65, 72, 80, 90]
pesos = [70, 65, 80, 90]

print("Média de idades:", calcula_media(idades))
print("Média de pesos:", calcula_media(pesos))
```

Uma única função serve para qualquer lista numérica.

Facilidade de entender, testar e manter

- Funções pequenas têm uma responsabilidade clara.
- Se algo der errado, sabemos onde procurar o problema.
- Podemos testar cada função separadamente.

Evitam efeitos colaterais inesperados

- Funções funcionais puras não dependem de variáveis externas.
- Isso garante que, dadas as mesmas entradas, o resultado sempre será o mesmo.

Exemplo de função pura (segura):

```
def soma(a, b):
    return a + b
```

Exemplo de função com efeito colateral (perigosa):

```
fator = 2
def multiplica(x):
    return x * fator
```

Se `fator` mudar, o resultado muda sem aviso → difícil de rastrear.

Reuso em diferentes contextos

- A mesma função pode ser usada em projetos diferentes.
- Em ciência de dados, criamos funções de limpeza, transformação e análise que podem ser aplicadas a diferentes bases.

Exemplo:

```
def classifica_idoso(idade):
    return "Idoso" if idade >= 60 else "Não idoso"

print(classifica_idoso(72))  # Idoso
print(classifica_idoso(45))  # Não idoso
```

A função pode ser usada em várias pesquisas diferentes.

---

8) Limitações da abordagem funcional pura

Apesar das vantagens, a programação funcional pura (sem variáveis externas, sem efeitos colaterais) também tem desafios:

Dificuldade com entrada e saída (I/O)

- Em ciência de dados, precisamos ler arquivos, mostrar gráficos, salvar resultados.
- Isso é um efeito colateral inevitável: interagir com o “mundo externo”.
- Em uma abordagem puramente funcional, isso é mais complexo de lidar.

Programas interativos ficam mais complicados

- Em jogos, sistemas de login, coleta de dados de usuários, precisamos de interação constante.
- Fazer isso só com funções puras exige estruturas complexas e menos intuitivas.

Exige uma forma diferente de pensar

- Iniciantes estão mais acostumados ao estilo imperativo (passo a passo).
- O estilo funcional puro pode ser difícil de entender no início.

Por isso, no Python...

O Python **não é 100% funcional**. Ele mistura:

- **Estilo funcional** → usamos funções puras sempre que possível (clareza, confiabilidade).
- **Estilo imperativo** → usamos variáveis e efeitos colaterais quando necessário (entrada de dados, gráficos, salvar arquivos).

Isso nos dá **flexibilidade**: organizamos o código em funções, mas sem abrir mão de praticidade.

Resumo

- **Funções e estilo funcional**: evitam repetição, facilitam manutenção, aumentam clareza e confiabilidade.
- **Limitações do estilo funcional puro**: lidar com I/O, interatividade e iniciantes pode ser complicado.
- **No Python**: usamos o **melhor dos dois mundos** → funções para organização e clareza, imperatividade para praticidade.

Conclusões

- Uma função é um **bloco de código reutilizável** que pode receber entradas e devolver saídas.
- Em Python usamos `def nome(parâmetros):`.
- O `return` devolve um valor para quem chamou a função.
- Funções tornam o código mais **organizado, reutilizável e confiável**.
- A **programação funcional** foca em funções sem efeitos colaterais, que sempre dão o mesmo resultado para os mesmos dados.
