Introdução - Parte-2 - Funções em Python

Informações do Autor

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

Introdução

Na primeira parte vimos como criar funções básicas, passando ou não parâmetros e retornando valores. Essas funções já permitem organizar programas simples.

Nesta segunda parte vamos avançar um passo: entender como o Python trata situações mais variadas no uso de funções. Em programas de análise de dados, é comum:

- Trabalhar com mais de uma informação ao mesmo tempo (ex.: nome, idade, cidade);
- Ter valores opcionais (ex.: um parâmetro com valor padrão, como cidade = "Goiânia");
- Chamar funções passando os dados em ordens diferentes ou usando nomes;
- Lidar com quantidades variáveis de argumentos, já que muitas vezes não sabemos quantos dados virão;
- Criar funções pequenas para cálculos rápidos, conhecidas como funções anônimas (`lambda`);
- Entender a diferença entre variáveis que vivem dentro e fora das funções (escopo).

1) Parâmetros e argumentos

Quando criamos uma função, podemos planejar que ela receba informações de fora. Essas informações são chamadas de parâmetros dentro da função. Na hora de chamar a função, passamos valores concretos para esses parâmetros, e esses valores são chamados de argumentos.

É como preencher um formulário:

- o campo do formulário corresponde ao parâmetro;

- a resposta que você escreve nesse campo é o argumento.

Estrutura

```
def nome_da_funcao(parâmetro):
    # usa o parâmetro dentro do bloco
    ...
```

Exemplo

```
def saudacao(nome):   # 'nome' é o parâmetro
    print("Olá,", nome)

saudacao("Maria")     # "Maria" é o argumento
saudacao("João")      # "João" é o argumento
```

Passo a passo

1. Criamos a função `saudacao` com um parâmetro chamado `nome`.
2. Dentro da função, usamos `print("Olá,", nome)`.
3. Ao chamar `saudacao("Maria")`, o argumento `"Maria"` é colocado no parâmetro `nome`.
4. O programa executa:

   ```
   Olá, Maria
   ```

5. Ao chamar `saudacao("João")`, o argumento `"João"` substitui o parâmetro, e a saída é:

   ```
   Olá, João
   ```

---

2) Funções com múltiplos parâmetros

Uma função pode receber várias informações ao mesmo tempo. Cada parâmetro corresponde a um valor que será fornecido na chamada da função. Quando usamos mais de um parâmetro, os argumentos devem ser passados na mesma ordem em que os parâmetros foram definidos.

```python
def apresentar(nome, idade):
    print("Nome:", nome, "| Idade:", idade)

apresentar("João", 65)
apresentar("Denise", 72)
```

Passo a passo:

1. A função `apresentar` foi definida com dois parâmetros: `nome` e `idade`.
2. Ao chamar `apresentar("João", 65)`, o argumento `"João"` é colocado em `nome` e o argumento `65` em `idade`.
3. O resultado é:

   ```
   Nome: João | Idade: 65
   ```

4. A mesma função pode ser chamada de novo com outros valores, como `"Denise"` e `72`.

Isso permite que uma mesma função trate diferentes conjuntos de dados sem precisar reescrever o código.

---

3) Parâmetros com valores padrão

Em algumas situações, queremos que um parâmetro tenha um valor já definido, mesmo que o usuário não informe nada. Esse valor é chamado de valor padrão. Se o argumento for passado na chamada da função, o valor padrão é substituído. Se não for passado, o valor padrão será usado automaticamente.

```python
def saudacao(nome, mensagem="Bem-vindo!"):
    print("Olá", nome, "-", mensagem)

saudacao("Carlos")
saudacao("Carlos", "Boa aula de Python!")
```

Passo a passo:

1. A função `saudacao` foi definida com dois parâmetros: `nome` e `mensagem`.
2. O parâmetro `mensagem` tem um valor padrão `"Bem-vindo!"`.
3. Na chamada `saudacao("Carlos")`, apenas o nome é informado. Como `mensagem` não foi fornecida, o programa usa o valor padrão. Saída:

```
Olá Carlos - Bem-vindo!
```

4. Na chamada `saudacao("Carlos", "Boa aula de Python!")`, os dois argumentos foram informados. O valor padrão é substituído. Saída:

```
Olá Carlos - Boa aula de Python!
```

Regra importante: quando uma função tem parâmetros com valores padrão, eles devem aparecer depois dos parâmetros obrigatórios. Caso contrário, o Python gera erro de sintaxe.

---

4) Argumentos nomeados

Normalmente o Python associa cada argumento ao parâmetro pela ordem em que aparecem. Mas também é possível indicar explicitamente qual parâmetro deve receber cada valor. Isso é feito escrevendo `nome_do_parametro=valor` na chamada da função.

```python
def relatorio(nome, idade, cidade="Goiânia"):
    print(f"{nome}, {idade} anos, mora em {cidade}")

relatorio("Joana", 70)
relatorio(idade=80, nome="Pedro")
relatorio(nome="Maria", idade=75, cidade="Anápolis")
```

Passo a passo:

1. A função `relatorio` foi definida com três parâmetros: `nome`, `idade` e `cidade`. O parâmetro `cidade` tem valor padrão `"Goiânia"`.
2. Na chamada `relatorio("Joana", 70)`, o Python usa a ordem dos parâmetros. Resultado:

```
Joana, 70 anos, mora em Goiânia
```

3. Na chamada `relatorio(idade=80, nome="Pedro")`, os parâmetros são informados pelo nome. A ordem não importa porque foi indicado explicitamente. Resultado:

```
Pedro, 80 anos, mora em Goiânia
```

4. Na chamada `relatorio(nome="Maria", idade=75, cidade="Anápolis")`, todos os argumentos foram passados de forma nomeada, incluindo a substituição do valor padrão de `cidade`. Resultado:

```
Maria, 75 anos, mora em Anápolis
```

Esse recurso ajuda a tornar o código mais claro e evita erros de posição quando uma função tem muitos parâmetros.

---

5) Argumentos variáveis

Em alguns casos, não sabemos de antemão quantos argumentos a função vai receber. O Python permite capturar esses argumentos de duas maneiras:

- `*args` → reúne todos os argumentos posicionais em uma tupla.
- `**kwargs` → reúne todos os argumentos nomeados em um dicionário.

Exemplo com `*args`

```
def listar_idosos(*nomes):
    for nome in nomes:
        print("Nome do idoso:", nome)

listar_idosos("Ana", "Carlos", "João")
```

Passo a passo:

1. A função `listar_idosos` usa `*nomes`. O asterisco indica que todos os argumentos recebidos serão agrupados em uma tupla chamada `nomes`.
2. Na chamada, foram passados três nomes. Dentro da função, `nomes` se torna a tupla `("Ana", "Carlos", "João")`.
3. O laço `for` percorre essa tupla e imprime cada nome.

Saída:

```
Nome do idoso: Ana
Nome do idoso: Carlos
Nome do idoso: João
```

Exemplo com `**kwargs`

```
def ficha(**dados):
    for chave, valor in dados.items():
        print(chave, ":", valor)

ficha(nome="Paulo", idade=68, cidade="Goiânia")
```

Passo a passo:

1. A função `ficha` usa `**dados`. Os dois asteriscos indicam que todos os argumentos nomeados serão reunidos em um dicionário chamado `dados`.
2. Na chamada, foram passados três pares nomeados. Dentro da função, `dados` se torna:

   ```
   {"nome": "Paulo", "idade": 68, "cidade": "Goiânia"}
   ```

3. O laço `for` percorre o dicionário e imprime cada chave e valor.

Saída:

```
nome : Paulo
idade : 68
cidade : Goiânia
```

Esse recurso permite criar funções mais flexíveis, que podem lidar com diferentes quantidades e tipos de dados sem precisar alterar sua definição.

---

6) Funções anônimas (lambda)

Quando criamos uma função em Python, normalmente usamos a palavra `def`, damos um nome para a função e escrevemos seu conteúdo. Isso é útil quando a função será usada várias vezes em diferentes partes do programa.

Porém, existem situações em que precisamos de uma função muito simples, que será usada apenas em um ponto específico. Criar uma função completa com `def` nesses casos pode deixar o código mais longo do que deveria.

Para resolver isso, o Python permite a criação de **funções anônimas**, chamadas de **lambda**. Elas recebem esse nome porque não precisam de um nome definido.

Estrutura

A forma geral de escrever uma função `lambda` é:

```
lambda parâmetros: expressão
```

- O que está antes dos dois pontos (`:`) são os parâmetros, como em qualquer função.
- O que está depois dos dois pontos é uma expressão que será calculada.
- O resultado da expressão é retornado automaticamente.
- A função sempre deve ter apenas uma expressão.

Exemplo 1 – função para calcular o quadrado

Com `def`:

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

print(quadrado(5))  # 25
```

Com `lambda`:

```
quadrado = lambda x: x * x
print(quadrado(5))  # 25
```

Os dois códigos fazem a mesma coisa. A diferença é que o `lambda` permite escrever a função em uma única linha.

Exemplo 2 – função para somar dois números

```
soma = lambda a, b: a + b
print(soma(3, 4))  # 7
```

Aqui a função `lambda` recebe dois parâmetros, `a` e `b`, e devolve a soma.

Onde o `lambda` costuma ser usado

O `lambda` é útil quando precisamos passar uma função como argumento para outra função. Por exemplo, ao ordenar uma lista de números:

```
juntar = lambda a, b: a + " " + b
print(juntar("Ciência", "Dados"))
```

Quando evitar o uso de `lambda`

Apesar de ser prático, o `lambda` deve ser usado apenas em situações simples. Não é adequado quando:

- a lógica da função precisa de várias etapas;
- o código deve ser reutilizado em mais de um lugar;
- é importante dar um nome claro para a função, facilitando a leitura.

Nesses casos, prefira usar `def` para criar uma função nomeada.

As funções `lambda` tornam o código mais compacto em casos simples, especialmente quando a função será usada apenas uma vez. Elas são comuns em tarefas de manipulação de listas e dados, mas não substituem o uso de funções nomeadas em situações mais complexas.

---

7) Escopo de variáveis

Quando criamos variáveis em Python, elas não estão disponíveis em todo o programa de forma automática. O lugar onde uma variável é criada define onde ela pode ser acessada. Esse lugar é chamado de **escopo**.

Entender escopo é importante porque evita confusões sobre qual valor uma variável tem em cada parte do código. Se não soubermos diferenciar, podemos criar erros difíceis de identificar.

Variáveis globais

Uma variável criada **fora de funções** é chamada de **global**. Ela pode ser acessada em qualquer parte do programa, inclusive dentro de funções (desde que não seja modificada lá dentro).

```python
x = 10  # variável global

def mostrar():
    print("Dentro da função:", x)

mostrar()
print("Fora da função:", x)
```

Saída:

```
Dentro da função: 10
Fora da função: 10
```

Aqui, a mesma variável `x` foi acessada tanto dentro quanto fora da função.

Variáveis locais

Quando criamos uma variável **dentro de uma função**, ela existe apenas naquele espaço. Essa variável é chamada de **local**.

```python
def exemplo():
    y = 5  # variável local
    print("Dentro da função:", y)

exemplo()
print("Fora da função:", y)  # erro
```

Saída:

```
Dentro da função: 5
NameError: name 'y' is not defined
```

O erro acontece porque `y` só existe dentro da função `exemplo`. Fora dela, essa variável não está definida.

Variáveis locais e globais com o mesmo nome

Se criarmos uma variável local com o mesmo nome de uma global, a local "esconde" a global enquanto a função estiver sendo executada.

```python
x = 10  # global

def exemplo():
    x = 5  # local
    print("Dentro da função:", x)

exemplo()
print("Fora da função:", x)
```

Saída:

```
Dentro da função: 5
Fora da função: 10
```

O Python diferencia as duas variáveis. Elas não são a mesma, apesar de terem o mesmo nome.

Modificando variáveis globais dentro de funções

Por padrão, uma função não pode alterar diretamente uma variável global. Se tentarmos, o Python cria uma variável local com o mesmo nome, o que pode causar confusão. Para realmente modificar uma variável global dentro de uma função, usamos a palavra-chave **global**.

```python
contador = 0

def incrementar():
    global contador
    contador += 1  # altera a variável global

incrementar()
print(contador)
```

Saída:

```
1
```

Sem a palavra `global`, o Python entenderia que `contador` dentro da função é uma nova variável local.

Funções dentro de funções e o uso de `nonlocal`

Também podemos ter funções definidas dentro de outras funções. Nesses casos, as variáveis da função externa podem ser acessadas pela função interna. Se quisermos **alterar** essas variáveis externas (mas que não são globais), usamos a palavra-chave **nonlocal**.

```
def externa():
    valor = 10

    def interna():
        nonlocal valor
        valor += 5
        print("Dentro da interna:", valor)

    interna()
    print("Dentro da externa:", valor)

externa()
```

Saída:

```
Dentro da interna: 15
Dentro da externa: 15
```

A palavra `nonlocal` indica que `valor` não é local à função interna, mas pertence a um escopo acima (no caso, a função `externa`).

Resumo sobre escopo

- Escopo é o limite de acesso de uma variável.
- Variável **global** é criada fora das funções e pode ser acessada de qualquer parte do programa.
- Variável **local** é criada dentro de uma função e só existe ali.
- Se uma variável local tiver o mesmo nome de uma global, a local tem prioridade dentro da função.
- Para alterar variáveis globais em funções, usamos `global`.
- Para alterar variáveis de funções externas em funções internas, usamos `nonlocal`.

Resumo

- Parâmetro é o nome usado na função, argumento é o valor passado.
- Funções podem ter múltiplos parâmetros, valores padrão e argumentos nomeados.

- `*args` e `**kwargs` permitem lidar com quantidades variáveis de argumentos.
- `lambda` cria funções simples e anônimas.
- O escopo define onde variáveis existem (local, global, não local).
