Explorando Funções em Python

Tempo estimado necessário: 15 minutos

Objetivos:

Ao final desta leitura, você deve ser capaz de:

1. Descrever o conceito de função e a importância das funções na programação
2. Escrever uma função que recebe entradas e realiza tarefas
3. Usar funções embutidas como len(), sum() e outras de forma eficaz
4. Definir e usar suas funções em Python
5. Diferenciar entre os escopos de variáveis globais e locais
6. Usar loops dentro da função
7. Modificar estruturas de dados usando funções

Introdução às funções

Uma função é um bloco de construção fundamental que encapsula ações ou cálculos específicos. Assim como na matemática, onde as funções recebem entradas e produzem saídas, as funções de programação funcionam de maneira semelhante. Elas recebem entradas, executam ações ou cálculos predefinidos e, em seguida, retornam uma saída.

Propósito das funções

As funções promovem a modularidade e a reutilização do código. Imagine que você tem uma tarefa que precisa ser executada várias vezes dentro de um programa. Em vez de duplicar o mesmo código em vários lugares, você pode definir uma função uma vez e chamá-la sempre que precisar dessa tarefa. Isso reduz a redundância e torna o código mais fácil de gerenciar e manter.

Benefícios de usar funções

Modularidade: Funções dividem tarefas complexas em componentes gerenciáveis
Reutilização: Funções podem ser usadas várias vezes sem reescrever o código
Legibilidade: Funções com nomes significativos aumentam a compreensão do código
Depuração: Isolar funções facilita a resolução de problemas e a correção de falhas
Abstração: Funções simplificam processos complexos por trás de uma interface amigável
Colaboração: Membros da equipe podem trabalhar em diferentes funções simultaneamente
Manutenção: Mudanças feitas em uma função se aplicam automaticamente onde quer que seja usada

Como as funções recebem entradas, realizam tarefas e produzem saídas

Entradas (Parâmetros)

Funções operam sobre dados e podem receber dados como entrada. Essas entradas são conhecidas como *parâmetros* ou *argumentos*. Parâmetros fornecem às funções as informações necessárias que elas precisam para realizar suas tarefas. Considere os parâmetros como valores que você passa para uma função, permitindo que ela trabalhe com dados específicos.

Realizando tarefas

Uma vez que uma função recebe sua entrada (parâmetros), ela executa ações ou cálculos predefinidos. Essas ações podem incluir cálculos, operações em dados ou até mesmo tarefas mais complexas. O propósito de uma função determina as tarefas que ela realiza. Por exemplo, uma função pode calcular a soma de números, ordenar uma lista, formatar texto ou buscar dados em um banco de dados.

Produzindo saídas

Após realizar suas tarefas, uma função pode produzir uma saída. Essa saída é o resultado das operações realizadas dentro da função. É o valor que a função "retorna" para o código que a chamou. Pense na saída como o produto final do trabalho da função. Você pode usar essa saída em seu código, atribuí-la a variáveis, passá-la para outras funções ou até mesmo imprimi-la para exibição.

Exemplo:

Considere uma função chamada `calculate_total` que recebe dois números como entrada (parâmetros), os soma e, em seguida, produz a soma como a saída. Aqui está como funciona:

```
def calculate_total(a, b):  # Parameters: a and b
    total = a + b           # Task: Addition
    return total            # Output: Sum of a and b
result = calculate_total(5, 7)  # Calling the function with inputs 5 and 7
print(result)  # Output: 12
```

Funções embutidas do Python

O Python possui um conjunto rico de funções embutidas que oferecem uma ampla gama de funcionalidades. Essas funções estão prontamente disponíveis para você usar, e você não precisa se preocupar com como elas são implementadas internamente. Em vez disso, você pode se concentrar em entender o que cada função faz e como usá-la de forma eficaz.

Usando funções internas ou funções pré-definidas

Para usar uma função interna, você simplesmente chama o nome da função seguido de parênteses. Quaisquer argumentos ou parâmetros necessários são passados para a função dentro desses parênteses. A função então executa sua tarefa predefinida e pode retornar uma saída que você pode usar em seu código.

Aqui estão alguns exemplos de funções internas comumente usadas:

len(): Calcula o comprimento de uma sequência ou coleção

```
string_length = len("Hello, World!")  # Output: 13
list_length = len([1, 2, 3, 4, 5])   # Output: 5
```

sum(): Soma os elementos em um iterável (lista, tupla, e assim por diante)

```
total = sum([10, 20, 30, 40, 50])  # Output: 150
```

max(): Retorna o valor máximo em um iterável

```
highest = max([5, 12, 8, 23, 16])  # Output: 23
```

min(): Retorna o valor mínimo em um iterável

```
lowest = min([5, 12, 8, 23, 16])  # Output: 5
```

As funções embutidas do Python oferecem uma ampla gama de funcionalidades, desde operações básicas como len() e sum() até tarefas mais especializadas.

Definindo suas funções

Definir uma função é como criar seu mini-programa:

1. Use `def` seguido pelo nome da função e parênteses

Aqui está a sintaxe para definir uma função:

```
def function_name():
    pass
```

Uma declaração `"pass"` em uma função de programação é um espaço reservado ou uma declaração de no-op (sem operação). Use-a quando você quiser definir uma função ou um bloco de código sintaticamente, mas não quiser especificar nenhuma funcionalidade ou implementação naquele momento.

- **Espaço Reservado:** `"pass"` atua como um espaço reservado temporário para o código futuro que você pretende escrever dentro de uma função ou de um bloco de código.
- **Requisito de Sintaxe:** Em muitas linguagens de programação, como Python, usar `"pass"` é necessário quando você define uma função ou um bloco condicional. Isso garante que o código permaneça sintaticamente correto, mesmo que ainda não faça nada.
- **Sem Operação:** `"pass"` em si não realiza nenhuma ação significativa. Quando o interpretador encontra “pass”, ele simplesmente avança para a próxima declaração sem executar nenhum código.

Parâmetros da Função:

- Parâmetros são como entradas para funções
- Eles vão dentro de parênteses ao definir a função
- Funções podem ter múltiplos parâmetros

Exemplo:

```
def greet(name):
    return "Hello, " + name
result = greet("Alice")
print(result)  # Output: Hello, Alice
```

Docstrings (Strings de Documentação)

- Docstrings explicam o que uma função faz
- Colocadas dentro de aspas triplas sob a definição da função
- Ajudam outros desenvolvedores a entender sua função

Exemplo:

```
def multiply(a, b):
    """
    This function multiplies two numbers.
    Input: a (number), b (number)
    Output: Product of a and b
    """
    print(a * b)
multiply(2,6)
```

Instrução de retorno

- O retorno devolve um valor de uma função
- Finaliza a execução da função e envia o resultado
- Uma função pode retornar vários tipos de dados

Exemplo:

```
def add(a, b):
    return a + b
sum_result = add(3, 5)  # sum_result gets the value 8
```

Compreendendo escopos e variáveis

Escopo é onde uma variável pode ser vista e utilizada:

- **Escopo Global:** Variáveis definidas fora das funções; acessíveis em qualquer lugar
- **Escopo Local:** Variáveis dentro das funções; utilizáveis apenas dentro daquela função

Exemplo:

Parte 1: Declaração de variável global

```
global_variable = "I'm global"
```

Esta linha inicializa uma variável global chamada `global_variable` e atribui a ela o valor “Eu sou global”.

Variáveis globais são acessíveis em todo o programa, tanto dentro quanto fora das funções.

Parte 2: Definição de função

```
def example_function():
    local_variable = "I'm local"
    print(global_variable)  # Accessing global variable
    print(local_variable)   # Accessing local variable
```

Aqui, você define uma função chamada `example_function()`.

Dentro dessa função:

- Uma variável local chamada local_variable é declarada e inicializada com o valor de string “Eu sou local.” Esta variável é local à função e só pode ser acessada dentro do escopo da função.
- A função então imprime os valores tanto da **variável global (global_variable) quanto da variável local (local_variable)**. Isso demonstra que você pode acessar variáveis globais e locais dentro de uma função.

Parte 3: Chamada de função

```
example_function()
```

Nesta parte, você chama a `example_function()` invocando-a. Isso resulta na execução do código da função.
Como resultado dessa chamada de função, ela imprimirá os valores das variáveis globais e locais dentro da função.

Parte 4: Acessando variáveis globais fora da função

```
print(global_variable)  # Accessible outside the function
```

Após chamar a função, você imprime o valor da variável global `global_variable` fora da função. **Isso demonstra que variáveis globais são acessíveis dentro e fora das funções.**

Parte 5: Tentando acessar variável local fora da função

```
# print(local_variable)  # Error, local variable not visible here
```

Nesta parte, você está tentando imprimir o valor da variável local `local_variable` fora da função. No entanto, essa linha resultaria em um erro.

Variáveis locais são visíveis e acessíveis apenas dentro do escopo da função onde foram definidas.

Tentar acessá-las fora desse escopo levantaria um `"NameError"`.

Usando funções com loops

Funções e loops juntos

1. Funções podem conter código com loops
2. Isso torna tarefas complexas mais organizadas
3. O código do loop se torna uma função reutilizável

Exemplo:

```
def print_numbers(limit):
    for i in range(1, limit+1):
```

```
        print(i)
print_numbers(5)  # Output: 1 2 3 4 5
```

Melhorando a organização e a reutilização do código

1. Funções agrupam ações semelhantes para fácil compreensão
2. O uso de loops dentro das funções mantém o código limpo
3. Você pode reutilizar uma função para repetir ações

Exemplo

```
def greet(name):
    return "Hello, " + name
for _ in range(3):
    print(greet("Alice"))
```

Modificando a estrutura de dados usando funções

Você usará Python e uma lista como a estrutura de dados para esta ilustração. Neste exemplo, você criará funções para adicionar e remover elementos de uma lista.

Parte 1: Inicializar uma lista vazia

```
# Define an empty list as the initial data structure
my_list = []
```

Nesta parte, você começa criando uma lista vazia chamada `my_list`. Esta lista vazia serve como a estrutura de dados que você modificará ao longo do código.

Parte 2: Defina uma função para adicionar elementos

```
# Function to add an element to the list
def add_element(data_structure, element):
    data_structure.append(element)
```

Aqui, você define uma função chamada `add_element`. Esta função recebe dois parâmetros:

- `data_structure`: Este parâmetro representa a lista à qual você deseja adicionar um elemento
- `element`: Este parâmetro representa o elemento que você deseja adicionar à lista

Dentro da função, você usa o método `append` para adicionar o elemento fornecido à data_structure, que se assume ser uma lista.

Parte 3: Defina uma função para remover elementos

```
# Function to remove an element from the list
def remove_element(data_structure, element):
    if element in data_structure:
        data_structure.remove(element)
    else:
        print(f"{element} not found in the list.")
```

Nesta parte, você define outra função chamada `remove_element`. Ela também recebe dois parâmetros:

- `data_structure`: A lista da qual queremos remover um elemento
- `element`: O elemento que queremos remover da lista

Dentro da função, você usa instruções condicionais para verificar se o elemento está presente na data_structure. Se estiver, você usa o método `remove` para remover a primeira ocorrência do elemento. Se não for encontrado, você imprime uma mensagem indicando que o elemento não foi encontrado na lista.

Parte 4: Adicionar elementos à lista

```
# Add elements to the list using the add_element function
add_element(my_list, 42)
add_element(my_list, 17)
add_element(my_list, 99)
```

Aqui, você usa a função `add_element` para adicionar três elementos (42, 17 e 99) à `my_list`. Esses elementos são adicionados um de cada vez usando chamadas de função.

Parte 5: Imprimir a lista atual

```
# Print the current list
print("Current list:", my_list)
```

Esta parte simplesmente imprime o estado atual da `my_list` no console, permitindo-nos ver os elementos que foram adicionados até agora.

Parte 6: Remover elementos da lista

```
# Remove an element from the list using the remove_element function
remove_element(my_list, 17)
remove_element(my_list, 55)  # This will print a message since 55 is not in the list
```

Nesta parte, você usa a função `remove_element` para remover elementos da my_list. Primeiro, você tenta remover 17 (que está na lista), e depois tenta remover 55 (que não está na lista). **A segunda chamada para `remove_element` imprimirá uma mensagem indicando que 55 não foi encontrado.**

Parte 7: Imprimir a lista atualizada

```
# Print the updated list
print("Updated list:", my_list)
```

Finalmente, você imprime a `my_list` atualizada no console. Isso nos permite observar as modificações feitas na lista ao adicionar e remover elementos usando as funções definidas.

Conclusão

Parabéns! Você completou o Laboratório de Instrução de Leitura sobre funções em Python. Você adquiriu uma compreensão sólida das funções, sua importância e como criá-las e usá-las de forma eficaz. Essas habilidades permitirão que você escreva um código mais organizado, modular e poderoso em seus projetos Python.