Explorando Funciones en Python

Tiempo estimado necesario: 15 minutos

Objetivos:

Al final de esta lectura, deberías ser capaz de:

1. Describir el concepto de función y la importancia de las funciones en la programación
2. Escribir una función que tome entradas y realice tareas
3. Usar funciones integradas como len(), sum() y otras de manera efectiva
4. Definir y usar tus funciones en Python
5. Diferenciar entre los ámbitos de variables globales y locales
6. Usar bucles dentro de la función
7. Modificar estructuras de datos utilizando funciones

Introducción a las funciones

Una función es un componente fundamental que encapsula acciones o cálculos específicos. Al igual que en matemáticas, donde las funciones reciben entradas y producen salidas, las funciones de programación funcionan de manera similar. Toman entradas, ejecutan acciones o cálculos predefinidos y luego devuelven una salida.

Propósito de las funciones

Las funciones promueven la modularidad y la reutilización del código. Imagina que tienes una tarea que necesita ser realizada múltiples veces dentro de un programa. En lugar de duplicar el mismo código en varios lugares, puedes definir una función una vez y llamarla cada vez que necesites realizar esa tarea. Esto reduce la redundancia y hace que el código sea más fácil de gestionar y mantener.

Beneficios de usar funciones

Modularidad: Las funciones descomponen tareas complejas en componentes manejables
Reutilización: Las funciones se pueden usar múltiples veces sin reescribir código
Legibilidad: Las funciones con nombres significativos mejoran la comprensión del código
Depuración: Aislar funciones facilita la resolución de problemas y la corrección de errores
Abstracción: Las funciones simplifican procesos complejos detrás de una interfaz fácil de usar
Colaboración: Los miembros del equipo pueden trabajar en diferentes funciones de manera concurrente
Mantenimiento: Los cambios realizados en una función se aplican automáticamente dondequiera que se use

Cómo las funciones toman entradas, realizan tareas y producen salidas

Entradas (Parámetros)

Las funciones operan sobre datos, y pueden recibir datos como entrada. Estas entradas se conocen como *parámetros* o *argumentos*. Los parámetros proporcionan a las funciones la información necesaria que necesitan para realizar sus tareas. Considera los parámetros como los valores que pasas a una función, permitiéndole trabajar con datos específicos.

Realizando tareas

Una vez que una función recibe su entrada (parámetros), ejecuta acciones o cálculos predefinidos. Estas acciones pueden incluir cálculos, operaciones sobre datos o incluso tareas más complejas. El propósito de una función determina las tareas que realiza. Por ejemplo, una función podría calcular la suma de números, ordenar una lista, formatear texto o recuperar datos de una base de datos.

Produciendo salidas

Después de realizar sus tareas, una función puede producir una salida. Esta salida es el resultado de las operaciones llevadas a cabo dentro de la función. Es el valor que la función "devuelve" al código que la llamó. Piensa en la salida como el producto final del trabajo de la función. Puedes usar esta salida en tu código, asignarla a variables, pasarla a otras funciones o incluso imprimirla para mostrarla.

Ejemplo:

Considera una función llamada `calculate_total` que toma dos números como entrada (parámetros), los suma y luego produce la suma como salida. Así es 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
```

Funciones incorporadas de Python

Python tiene un conjunto rico de funciones incorporadas que ofrecen una amplia gama de funcionalidades. Estas funciones están disponibles para que las uses, y no necesitas preocuparte por cómo están implementadas internamente. En su lugar, puedes concentrarte en entender qué hace cada función y cómo usarla de manera efectiva.

Uso de funciones integradas o funciones predefinidas

Para usar una función integrada, simplemente llamas al nombre de la función seguido de paréntesis. Cualquier argumento o parámetro requerido se pasa a la función dentro de estos paréntesis. La función luego realiza su tarea predefinida y puede devolver una salida que puedes usar en tu código.

Aquí hay algunos ejemplos de funciones integradas comúnmente utilizadas:

len(): Calcula la longitud de una secuencia o colección

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

sum(): Suma los elementos en un iterable (lista, tupla, etc.)

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

max(): Devuelve el valor máximo en un iterable

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

min(): Devuelve el valor mínimo en un iterable

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

Las funciones integradas de Python ofrecen una amplia gama de funcionalidades, desde operaciones básicas como len() y sum() hasta tareas más especializadas.

Definiendo tus funciones

Definir una función es como crear tu mini-programa:

1. Usa `def` seguido del nombre de la función y paréntesis

Aquí está la sintaxis para definir una función:

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

Una declaración `"pass"` en una función de programación es un marcador de posición o una declaración no operativa (no operation). Úsala cuando quieras definir una función o un bloque de código sintácticamente, pero no desees especificar ninguna funcionalidad o implementación en ese momento.

- **Marcador de posición:** `"pass"` actúa como un marcador de posición temporal para el código futuro que pretendes escribir dentro de una función o un bloque de código.
- **Requisito de sintaxis:** En muchos lenguajes de programación como Python, usar `"pass"` es necesario cuando defines una función o un bloque condicional. Asegura que el código permanezca sintácticamente correcto, incluso si aún no hace nada.
- **No operación:** `"pass"` en sí no realiza ninguna acción significativa. Cuando el intérprete encuentra “pass”, simplemente pasa a la siguiente declaración sin ejecutar ningún código.

Parámetros de Función:

- Los parámetros son como entradas para las funciones
- Van dentro de paréntesis al definir la función
- Las funciones pueden tener múltiples parámetros

Ejemplo:

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

Docstrings (Cadenas de Documentación)

- Las docstrings explican qué hace una función
- Se colocan dentro de comillas triples debajo de la definición de la función
- Ayudan a otros desarrolladores a entender tu función

Ejemplo:

```
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)
```

Declaración de retorno

- Return devuelve un valor de una función
- Finaliza la ejecución de la función y envía el resultado
- Una función puede devolver varios tipos de datos

Ejemplo:

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

Entendiendo los ámbitos y las variables

El ámbito es donde se puede ver y usar una variable:

- **Ámbito Global:** Variables definidas fuera de las funciones; accesibles en todas partes
- **Ámbito Local:** Variables dentro de las funciones; solo utilizables dentro de esa función

Ejemplo:

Parte 1: Declaración de variable global

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

Esta línea inicializa una variable global llamada `global_variable` y le asigna el valor “I’m global”.

> Las variables globales son accesibles en todo el programa, tanto dentro como fuera de las funciones.

Parte 2: Definición de función

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

Aquí, defines una función llamada `example_function()`.

Dentro de esta función:

- Se declara una variable local llamada local_variable y se inicializa con el valor de cadena “I'm local.” Esta variable es local a la función y solo se puede acceder dentro del ámbito de la función.
- La función luego imprime los valores tanto de la **variable global (global_variable) como de la variable local (local_variable)**. Demuestra que puedes acceder a variables globales y locales dentro de una función.

Parte 3: Llamada a función

```
example_function()
```

En esta parte, llamas a la `example_function()` invocándola. Esto resulta en la ejecución del código de la función.
Como resultado de esta llamada a la función, imprimirá los valores de las variables globales y locales dentro de la función.

Parte 4: Accediendo a la variable global fuera de la función

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

Después de llamar a la función, imprimes el valor de la variable global `global_variable` fuera de la función. **Esto demuestra que las variables globales son accesibles dentro y fuera de las funciones.**

Parte 5: Intentando acceder a una variable local fuera de la función

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

En esta parte, estás intentando imprimir el valor de la variable local `local_variable` fuera de la función. Sin embargo, esta línea resultaría en un error.

> Las variables locales solo son visibles y accesibles dentro del ámbito de la función donde se definen.

Intentar acceder a ellas fuera de ese ámbito generaría un `"NameError"`.

Uso de funciones con bucles

Funciones y bucles juntos

1. Las funciones pueden contener código con bucles
2. Esto hace que las tareas complejas sean más organizadas
3. El código del bucle se convierte en una función reutilizable

Ejemplo:

```
def print_numbers(limit):
    for i in range(1, limit+1):
        print(i)
print_numbers(5)  # Output: 1 2 3 4 5
```

Mejorando la organización y reutilización del código

1. Las funciones agrupan acciones similares para una fácil comprensión
2. El bucle dentro de las funciones mantiene el código limpio
3. Puedes reutilizar una función para repetir acciones

Ejemplo

```
def greet(name):
```

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

Modificando la estructura de datos utilizando funciones

Utilizarás Python y una lista como la estructura de datos para esta ilustración. En este ejemplo, crearás funciones para agregar y eliminar elementos de una lista.

Parte 1: Inicializar una lista vacía

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

En esta parte, comienzas creando una lista vacía llamada `my_list`. Esta lista vacía sirve como la estructura de datos que modificarás a lo largo del código.

Parte 2: Definir una función para agregar elementos

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

Aquí, defines una función llamada `add_element`. Esta función toma dos parámetros:

- `data_structure`: Este parámetro representa la lista a la que deseas agregar un elemento.
- `element`: Este parámetro representa el elemento que deseas agregar a la lista.

Dentro de la función, utilizas el método `append` para agregar el elemento proporcionado a la data_structure, que se asume que es una lista.

Parte 3: Definir una función para eliminar 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.")
```

En esta parte, defines otra función llamada `remove_element`. También toma dos parámetros:

- `data_structure`: La lista de la que queremos eliminar un elemento
- `element`: El elemento que queremos eliminar de la lista

Dentro de la función, utilizas sentencias condicionales para verificar si el elemento está presente en la data_structure. Si lo está, utilizas el método `remove` para eliminar la primera ocurrencia del elemento. Si no se encuentra, imprimes un mensaje indicando que el elemento no se encontró en la lista.

Parte 4: Agregar elementos a la 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)
```

Aquí, utilizas la función `add_element` para agregar tres elementos (42, 17 y 99) a `my_list`. Estos se añaden uno a la vez utilizando llamadas a la función.

Parte 5: Imprimir la lista actual

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

Esta parte simplemente imprime el estado actual de `my_list` en la consola, lo que nos permite ver los elementos que se han añadido hasta ahora.

Parte 6: Eliminar elementos de la 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
```

En esta parte, utilizas la función `remove_element` para eliminar elementos de my_list. Primero, intentas eliminar 17 (que está en la lista), y luego intentas eliminar 55 (que no está en la lista). **La segunda llamada a `remove_element` imprimirá un mensaje indicando que 55 no fue encontrado.**

Parte 7: Imprimir la lista actualizada

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

Finalmente, imprimes la `my_list` actualizada en la consola. Esto nos permite observar las modificaciones realizadas en la lista al agregar y eliminar elementos utilizando las funciones definidas.

Conclusión

¡Felicidades! Has completado el Laboratorio de Instrucción de Lectura sobre funciones en Python. Has adquirido una comprensión sólida de las funciones, su importancia y cómo crearlas y utilizarlas de manera efectiva. Estas habilidades te permitirán escribir código más organizado, modular y potente en tus proyectos de Python.


Skills Network