Programador de Videojuegos

CLASE 08/08/2025

Funciones

Funciones

Una función es un bloque de código que se puede usar muchas veces. Sirve para organizar y reutilizar instrucciones que hacen una tarea específica.

Sirve para:

- Organizar el código
- Evitar repetir instrucciones
- Dividir el programa en partes pequeñas y fáciles de entender

Sintaxis:

```
def nombre_funcion(parámetros):
    # Bloque de código que realiza una tarea

    # Retorna un valor (opcional)
    return valor
```

Explicación

- ➔ def: Palabra clave para definir la función.
- ➔ nombre_de_funcion: El nombre de la función (por convención, se usa minúsculas y palabras separadas por guiones bajos).
- ➔ parámetros: Son los valores de entrada (opcional), que la función puede utilizar para realizar su tarea.
- ➔ return: Es el valor que la función devuelve (opcional).

Funciones sin parámetros

- Función sin parámetros

Paso 1: Definir una función

Ejemplo:

```
def saludar():
    print("Hola jugador")
```

Paso 2: Ejecutar la función
Para ejecutar una función solo se escribe su nombre seguido de paréntesis.

```
saludar()
```

Salida: Hola jugador

Funciones con parámetros

¿Qué es un parámetro?
Un parámetro es una variable que se define entre los paréntesis de una función, y que recibe un valor cuando la función es llamada. En otras palabras, es como una “entrada” que la función necesita para trabajar. Permiten que una función sea más flexible y reutilizable, porque no depende de datos fijos.

Ejemplo:

```
def saludar(nombre):   # 'nombre' es un parámetro
    print(f"Hola, {nombre}")

#Ejecutamos la función
saludar("Pablo")      # "Pablo" es un argumento
saludar("Ana")      # "Ana" es un argumento
```

Salida: Hola, Pablo
Hola, Ana

Términos:

- **Parámetro**: Variable que se define en la función, permite recibir un valor de entrada.
- **Argumento**: Valor que se le da a un parámetro al llamar a la función.
- **Valor por defecto**: Parámetro con un valor predefinido.

- **Se puede definir más de un parámetro**

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

resultado = sumar(5, 3)
print(resultado)
```

Salida: 8

Funciones con parámetros

- **Parámetros opcionales o con valor por defecto**

Un parámetro opcional (también llamado parámetro con valor por defecto) es un parámetro que no es obligatorio al llamar a la función, porque ya tiene asignado un valor predeterminado.
Sirve para que la función pueda funcionar con o sin ese dato. Si no se pasa el argumento, se usa el valor por defecto.

Sintaxis:

```
def funcion(param="valor"):
    #codigo dentro de la función
```

Ejemplo:

```
def saludar(nombre="jugador"):
    print(f"Hola, {nombre}")
```

Explicación:

➔ nombre: es el parámetro
➔ "jugador": es el valor por defecto

```
saludar()        # Usa el valor por defecto
saludar("Rocío")    # Usa el valor pasado
```

Salida:
Hola, jugador
Hola, Rocío

Funciones con parámetros

Reglas importantes

1. Los parámetros opcionales van al final.
2. Se pueden combinar parámetros obligatorios y opcionales.

Esto da error:

```
def ejemplo(a=5, b):  # b no puede estar después de un valor por defecto
```

Forma correcta:

```
def ejemplo(b, a=5):
```

Ejemplo aplicado a videojuegos

```
def atacar(jugador, enemigo="Enemigo genérico"):
    print(f"{jugador} ataca a {enemigo}")
```

Llamadas posibles:

```
atacar("Mario")              # Usa enemigo por defecto
atacar("Link", "Ganon")      # Usa los dos argumentos
```

Salida:
Mario ataca a Enemigo genérico
Link ataca a Ganon

Funciones con parámetros

En Python, un parámetro opcional *debe* tener un valor predeterminado.

¿Y si queremos que el valor predeterminado sea “vacío”?

Podés usar None, una cadena vacía "", una lista vacía [] dependiendo de la función:

Ejemplo:

```
def mostrar_mensaje(mensaje=None):
    if mensaje is None:
        print("No hay mensaje para mostrar.")
    else:
        print(mensaje)
```

Funciones con valor de retorno

- **Función con valor de retorno**

Las funciones pueden también retornar valores. Esto significa que después de ejecutar el bloque de código, la función puede devolver un resultado que puede ser usado en otros lugares de tu programa.

return valor

Se utiliza la palabra *return* seguida de un valor, para devolver ese valor desde la función hacia el código que la llama.

Ejemplo:

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

resultado = suma(3, 5)  # Llamada a la función y almacenamiento del resultado

print(resultado)  # Imprime 8
```

Paso por valor o por referencia

Dependiendo del tipo de dato que enviemos a la función, podemos diferenciar dos comportamientos:

- **Paso por valor:**
  - Se crea una copia del valor de los argumentos en los respectivos parámetros de la función.
  - Las modificaciones en los valores de los parámetros no afectan a las variables externas. Si se hace una asignación dentro de la función, se crea un nuevo objeto y no afecta al original.
  - En Python se pasan por valor los tipos de datos simples: enteros, flotantes, cadenas, lógicos.

Ejemplo:

```
def cambiar_valor(x):
    x = 100  # Crea una nueva referencia local

a = 10
cambiar_valor(a)
print(a)  # 10 (no cambió)
```

Para modificar los tipos de datos simples, se puede devolverlos modificados y reasignarlos:

```
def cambiar_valor(x):
    x = 100  # Crea una nueva referencia local y le asigna 100
    return x

a = 10
print("Antes de ejecutar la función, a = ", a )  # 10
a = cambiar_valor(a) # Al ejecutar la función retorna el valor modificado y lo asigna
print("Después de ejecutar la función, a = ", a )  # 100
```

Paso por valor o por referencia

- **Paso por referencia:**
  - Se pasa un puntero a la posición de memoria donde se aloja el dato.
  - Cualquier cambio que se haga en su valor dentro de la función afecta el contenido de la variable en el resto del programa.
  - En Python se pasan por referencia los tipos de datos compuestos: listas, diccionarios, conjuntos.

Ejemplo:

```python
def modificar_lista(lista):
    lista.append(4)  # Modifica el contenido de la lista

mi_lista = [1, 2, 3]
modificar_lista(mi_lista)
print(mi_lista)  # [1, 2, 3, 4]
```

Se puede evitar la modificación de los valores enviados por referencia, pasando como argumento una copia del dato compuesto:

```python
def modificar_lista(lista):
    lista.append(4)  # Modifica el contenido de la lista
    print("Imprimo la copia de la lista dentro de la función", lista)  # [1, 2, 3, 4]


mi_lista = [1, 2, 3]
modificar_lista(mi_lista[:]) # Se pasa una copia de la lista original
print("Fuera de la función, la lista original no se modificó", mi_lista)  # [1, 2, 3]
```

Paso por valor o por referencia

Otro ejemplo:

```python
# Defino una función
# Primer parámetro: paso por referencia
# Segundo parámetro: paso por valor
def ejemplo(lista, numero):
    lista.append(99)  # Modifica la lista
    numero += 1       # Crea una nueva variable local a la cual le incrementa el valor recibido por parámetro

valores = [1, 2, 3]
x = 5

ejemplo(valores, x) # Ejecuta la función
print(valores)  # [1, 2, 3, 99]
print(x)        # 5
```

Ámbito de las variables

Ámbito de las variables

El ámbito (o scope) de una variable hace referencia a la región del programa en la que una variable es visible y accesible. Es decir, las variables están definidas dentro de un ámbito que determina dónde la variable puede ser utilizada. Este alcance es determinado por la ubicación en el código donde se define (por ejemplo, dentro de una función, fuera de cualquier función, etc.).

A su vez, ciclo de vida de una variable se refiere al tiempo en que una variable permanece en memoria.

Alcance y visibilidad de las variables:

- **Variables locales:** Sólo son visibles en el ámbito donde fueron declaradas.
- **Variables globales:** Se pueden acceder en cualquier parte del programa.

Ámbito Local (Local Scope)

El ámbito local se refiere a las variables que están definidas dentro de una función. Es decir, las variables que se crean dentro de una función sólo son accesibles dentro de esa función.

- **Alcance de las variables locales:** Están disponibles solo dentro de la función o bloque en el que se crean. No pueden ser accedidas fuera de la función.

- **Ciclo de vida de las variables locales:** Existen sólo mientras se ejecuta la función o el bloque de código.

Ejemplo:

```
1   #Defino una función
2   def mi_funcion():
3       x = 10  # 'x' es una variable local
4       print(x)
5
6   mi_funcion()  # Al ejecutar la función imprime el valor de x = 10
7
8   #Si quiero imprimir el valor de x fuera de la función dará error
9   # print(x)  # Error: 'x' no está definida fuera de la función
```

En este ejemplo, la variable x es local en la función mi_funcion(). Por lo cual, no se puede acceder a la variable x fuera de esa función.

Ámbito Global (Global Scope)

El ámbito global se refiere a las variables que están definidas fuera de cualquier función. Son accesibles desde cualquier parte del programa, incluidas las funciones.

- **Alcance de las variables globales:**
  - Están disponibles a nivel global (fuera de todas las funciones).
  - Pueden ser leídas dentro de las funciones.
  - Pueden ser modificadas dentro de las funciones si se usa la palabra clave global.
- **Ciclo de vida de las variables globales:** Existen durante toda la ejecución del programa.

Ejemplo:

```python
x = 20  # 'x' es una variable global

#Defino una función
def mi_funcion():
    print(x)  # Accede a 'x' que es global

mi_funcion()  # Al ejecutar la función imprime el valor de x = 20

# A su vez se puede utilizar la variable x fuera de la función
print(x)  # Imprimo el valor de x = 20 (acceso desde el ámbito global)
```

En este caso, la variable x es global y puede ser utilizada dentro de la función mi_funcion() y también fuera de ella.

Ámbito Global (Global Scope)

Modificar una variable global dentro de una función

Si se desea modificar una variable global dentro de una función, se debe usar la palabra clave ***global*** para indicar que se refiere a la variable global, no a una local. Sin la palabra clave global, Python crearía una variable local con el mismo nombre dentro de la función, lo que causaría que el cambio no se refleje en la variable global.

Ejemplo:

```
x = 30

def modificar_global():
    global x
    print("Antes de modificar el valor, x = ", x) #Imprime:30
    x = 40  # Se modifica la variable global

modificar_global()
print(x)  # Imprime: 40
```

Ámbito

Es posible definir, dentro de una función, una variable local con el mismo nombre que tiene una declarada en el programa principal. Pero se trata de otra variable con el mismo nombre, no comparten su contenido.

```
# 'x' es una variable global
x=10

#Defino una función
def mi_funcion():
    x = 20  # 'x' es una variable local
    print("Dentro de la función, la variable local x vale ", x)

mi_funcion()  # Al ejecutar la función imprime el valor de la variable local x = 20

print("Fuera de la función la variable global x vale ", x) # variable global x = 10
```

Ámbito Envolvente (Enclosing Scope)

El ámbito envolvente se refiere a las variables de una función externa que rodea a una función interna. Esto ocurre en funciones anidadas (una función dentro de otra).

Las variables de una función externa son accesibles en una función interna, pero no se pueden modificar directamente sin usar ***nonlocal***.

Ejemplo:

```
1   def externa():
2       x = 10  # Defino la variable local x
3       print("Antes de ejecutar la función interna, x = ", x) # Imprime:10
4
5       def interna():
6           """
7           La variable x es accesible dentro de la función interna()
8           porque interna() está dentro de la función externa()
9           """
10          nonlocal x # Se usa nonlocal para modificar una variable del ámbito envolvente
11          x = 20 # Modifico el valor de x
12
13      interna() # La función externa, ejecuta la función interna
14      print("Después de ejecutar la función interna, x = ",x)
15
16  externa()  # Ejecuta la función externa
```

Ejercitación

Ejercitación

CONSIGNA 1: Inventario básico (Listas + Funciones)

Objetivo: Crear un sistema de inventario usando listas y funciones.

Instrucciones:

- Crear una lista que represente el inventario del jugador.
- Escribir funciones para:
  - Agregar un elemento.
  - Eliminar un elemento.
  - Mostrar el inventario
  - Verificar si un objeto está en el inventario

Ejercitación

CONSIGNA 2: Control de enemigos (Diccionarios + Iteración)

Objetivo: Gestionar enemigos mediante un diccionario y recorrerlos con un bucle.

Instrucciones:

- Crear un diccionario donde la clave sea el nombre del enemigo y el valor un sub diccionario con su vida y daño.
- Hacer funciones para:
  - Mostrar todos los enemigos
  - Restar la vida a un enemigo específico, pasando por parámetro el daño a impactar.
  - Eliminar al enemigo si su vida llega a 0

Desafío

Mapa del juego

Representar un mapa como una matriz (lista de listas) y permitir que el jugador interactúe con este mapa, moviéndose y detectando enemigos y metas. El mapa estará representado por una lista de listas. En cada celda, se utilizará:

- "P": Posición del jugador.
- "M": Meta (donde el jugador debe llegar).
- "E": Enemigo (el jugador pierde si pisa esta celda).
- " ": Espacio vacío.

```
mapa = [
    [" ", " ", "E", " "],
    ["P", " ", " ", "M"],
    [" ", " ", "E", " "],
    [" ", " ", " ", " "]
]
```

Se pide crear un mapa que esté representado por una lista 2D, donde cada elemento de la lista interna representará una celda del mapa. El jugador (P) se moverá a través del mapa, que tendrá ciertos obstáculos o metas representadas por caracteres como M (meta) o E (enemigo).

- Crear una función mostrar_mapa(mapa) que imprima el mapa.
- Agregar funciones para:
    - Mover al jugador (P) en la matriz
    - Detectar si llega a una meta (M)
    - o pisa un enemigo (E)