MÓDULO 3. FUNCIONES EN PYTHON

Loreto Pelegrín Castillo

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
Región de Murcia
Sef
Servicio Regional DE EMPLEO y FORMACIÓN
AVANZA

Índice:

1

Introducción

1. Introducción

- En las unidades anteriores hemos escrito programas sencillos que se ejecutan secuencialmente, es decir, el programa empezaba por las instrucciones del principio y se iba ejecutando línea por línea hasta el final, utilizando instrucciones condicionales y bucles. Esto se conoce como **programación estructurada**.

1. Introducción

- Cada vez los programas se iban haciendo mas complejos y se desarrollo un nuevo tipo de programación: **Programación modular**.
  - Consiste en dividir el programa en subprogramas para hacerlo mas legible, manejable y ahorrar código, pues estos programas son reutilizables.
  - Estos subprogramas pueden estar compuestos a su vez de unos componentes, que es lo que conocemos como **Función**.

2

Estudio de las funciones

2. Funciones

- ¿Qué es una función?
  - Es un bloque de código reutilizable que puede ser ejecutado las veces que se estime oportuno en un proyecto.
  - **Ventajas:** Reduce el numero de líneas de código en un proyecto.
- Anteriormente hemos usado **funciones nativas** que vienen con Python como **len()** para calcular la longitud de una lista, pero al igual que en otros lenguajes de programación, también podemos definir nuestras propias funciones.

2.1. Definición de funciones

Para ello hacemos uso de **def.** Y de la siguiente manera se declararía una **función**:

```
def nombre_funcion(argumentos):
        código
        return retorno
```

Los argumentos son datos opcionales que se introducen la funcion para poder operar com ellos.

2.1. Definición de funciones

- Empecemos por la función más sencilla de todas. Una función sin parámetros de entrada ni parámetros de salida.

```
def hola():
    print("Hola Mundo")

hola()
```

- Hemos declarado o definido la función. El siguiente paso es llamarla con **hola().** Si lo realizamos veremos que se imprime Hola Mundo.

2.1. Definición de funciones

- Vamos a complicar un poco las cosas pasando un argumento de entrada. Ahora si pasamos como entrada un nombre, se imprimirá Hola y el nombre.

```
def hola(nombre):
    print("Hola", nombre)
hola("Loreto")
# Hola Loreto
```

2.1. Definición de funciones

- Los argumentos por posición o posicionales son la forma más básica e intuitiva de pasar parámetros. Si tenemos una función sumar() que acepta dos parámetros, se puede llamar como se muestra a continuación.

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

resultado = sumar(3, 5)
print(resultado)  # 8
```

- Al tratarse de parámetros posicionales, se interpretará que el primer número es la a y el segundo la b. El número de parámetros es fijo, por lo que si intentamos llamar a la función con solo uno, dará error.

2.1. Definición de funciones

- Se pueden definir valores predeterminados para los parámetros, interpretando que el valor de ese parámetro es el predeterminado si no se le proporciona otro.

```
def suma(a, b=3):
    return a+b
# Solo se le pasa 1 parametro.
# El parametro a toma el valor 1 mientras que el b esta definido como 3
resultado = suma(1) #4
print(resultado)
# Por el contrario al pasarle valor al parametro b, se anula lo declarado
# en la definicion de la funcion
resultado = suma(1,1) #2
print(resultado)
```

2.1. Definición de funciones

- Se puede pasar los parámetros en el orden que se desee, mientras se declaren con el nombre del parámetro.

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

resultado = suma(b=2,a=2)
print(resultado)
```

2.1. Definición de funciones

- Se puede pasar los parámetros en el orden que se desee, mientras se declaren con el nombre del parámetro.

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

resultado = suma(b=2,a=2)
print(resultado)
```

2.1. Definición de funciones

- Sin embargo no es posible pasar un argumento predeterminado antes que uno que no lo sea.

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

resultado = suma(b=2,3)
```

```
    resultado = suma(b=2,3)
                        ^
SyntaxError: positional argument follows keyword argument
```

2.1. Definición de funciones

- Se le puede asignar una función a una variable y usar esa variable como función:

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

s = suma
resultado = s(1,2)
print(resultado)
```

2.1. Definición de funciones

- Para crear una función en la cual no conozcamos el número de argumentos no lo conocemos, se declara el argumento de la función con un *.

```
def media_aritmetica(*valores):
    suma = 0
    for num in valores:
        suma +=num
    print (f"La media aritmetica de los numeros introducidos es: {suma/len(valores)}")

media_aritmetica(3,6,2,6,1,6,3,3,3) #La media aritmetica de los numeros introducidos es: 3.666666666

media_aritmetica(1,2,3) #La media aritmetica de los numeros introducidos es: 2.0
```

2.1. Definición de funciones

- Para ingresar un numero indeterminado de argumentos, pero asociado a un nombre de variable, se utilizará ** delante del argumento.
- Se puede pensar en la manera de introducir los datos como si fuera un diccionario.

```
def coche(marca,**caracteristicas):
    print(f"El coche es de la marca {marca} y tiene las siguentes caracteristicas: ")
    for valores in caracteristicas.items():
        print(f"{valores[0]}: {valores[1]}")

coche ("Mercedes",Modelo = "GLA220", Puertas = 5, Color = "Rojo")
```

```
El coche es de la marca Mercedes y tiene las siguentes caracteristicas:
Modelo: GLA220
Puertas: 5
Color: Rojo
```

2.1. Definición de funciones

- Ejercicio de clase:
  - Crea una función que pregunte el nombre y la edad del usuario, y la muestre por pantalla.
  - Crear una función que imprima la tabla de multiplicar de una número.

2.2. Alcance de las funciones

- Otro factor a tener en cuenta es el alcance de las variables. Esto no es más que el ámbito o el entorno en el que la variable se puede usar por parte del programa. Hay 2 tipos de ámbitos:

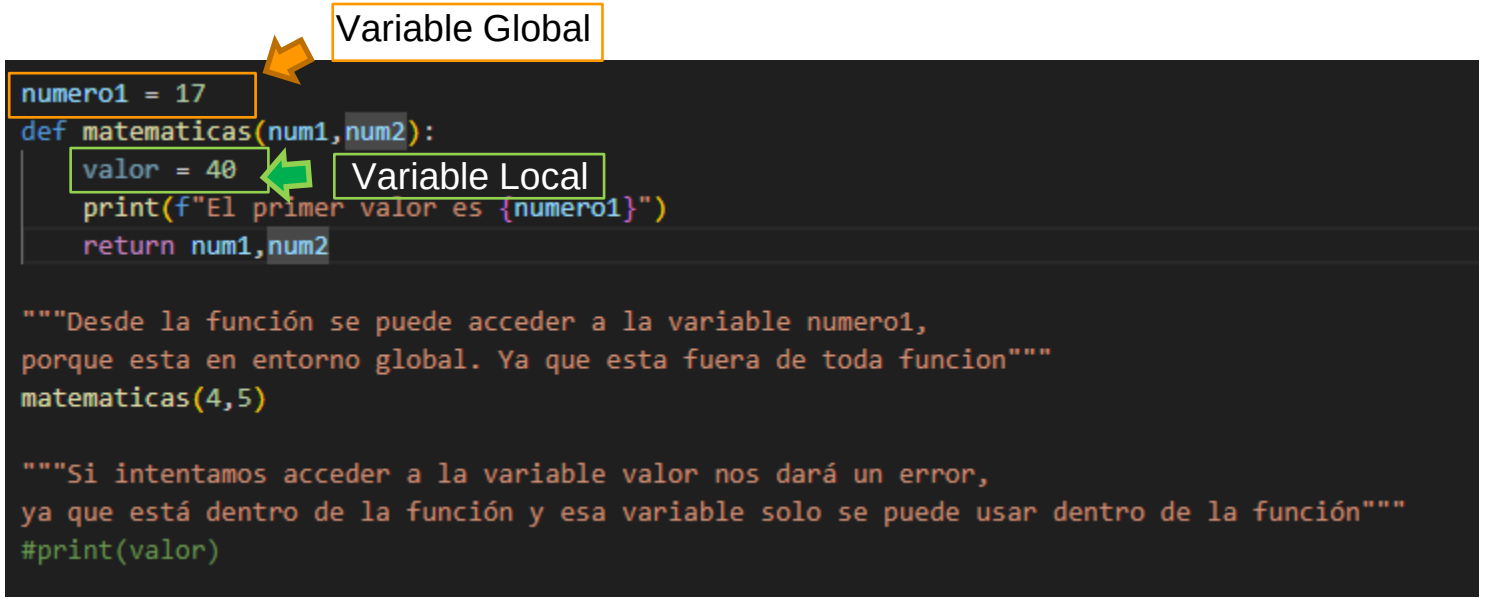
  - **Global:** La variable es accesible desde cualquier punto del programa.
  - **Local:** A la variable solo se puede acceder estando en el mismo ámbito que ella.

2.2. Alcance de las funciones

- Otro factor a tener en cuenta es el alcance de las variables. Esto no es más que el ámbito o el entorno en el que la variable se puede usar por parte del programa. Hay 2 tipos de ámbitos:

  - **Global:** La variable es accesible desde cualquier punto del programa.
  - **Local:** A la variable solo se puede acceder estando en el mismo ámbito que ella.

2.2. Alcance de las funciones



2.2. Alcance de las funciones

- Cuando tenemos 2 variables que se llaman igual, pero en ámbitos diferentes, modificar una, no modifica a la otra.

```
numero = 7
def modificar_valor(valor):
    numero = valor
    print(f"Este es el valor del numero dentro de la funcion {numero}")

print(f"Este es el valor del numero fuera de la funcion {numero}")
# Este es el valor del numero fuera de la funcion 7

modificar_valor(9)
# Este es el valor del numero dentro de la funcion 9
```

2.2. Alcance de las funciones

- Si queremos hacer una variable global y acceder desde cualquier lugar del programa utilizaremos la palabra reservada **global** delante de la variable.

```
valor = 5
def operacion(num):
    global valor
    valor += num
    return valor
```

```
print(valor) #5

print(operacion(7)) #12

print(valor) #12

print(operacion(2)) #14
#Aqui no se modifica el valor solo se le suma 5
print(valor +5) # 19

print(valor)#14
```

2.2. Alcance de las funciones

- En el siguiente bloque de código, la asignación de la x no forma parte de la función ya que no esta identada.

```
def miFuncion():
    print("Esta es")
    print("Mi primera funcion")

x = 7 #La asignacion de la x no forma farte de la función ya que no esta identada
```

2.3. Paso por valor y paso por referencia

- Dependiendo del tipo de dato que enviemos a la función como argumento, este se puede pasar a la función de 2 maneras:
  - Paso por valor
  - Paso por referencia

2.3.1. Paso por valor

- En este caso se crea una copia local del valor del dato dentro de la función. El valor que se modifica dentro de la función solo afecta a la variable dentro de esta, no al valor del exterior.
- Este comportamiento ocurre generalmente con los tipos de datos simples como enteros, flotantes, cadenas, …

```
numero = 9
def sumatorio(num):
    return numero + num

print(numero) #9

print(sumatorio(5)) #14

print(numero) #9
```

2.3.1. Paso por referencia

- Para este caso se maneja directamente el valor de la variable. Los cambios que se realizan a esta dentro de la función afectará directamente al valor global de la variable. Afecta a las listas, diccionarios, …
- Para que no ocurra, se debe de pasar una copia de la variable por la función.

```
lista_numeros= [1,2,3,4,5]

def multi_cinco(valores):
    for i,n in enumerate(valores):
        valores[i]=n*5
    return valores

print(lista_numeros) #[1, 2, 3, 4, 5]

print(multi_cinco(lista_numeros)) #[5, 10, 15, 20, 25]

print(lista_numeros) #[5, 10, 15, 20, 25]
```

2.4. Funciones anidadas

- En Python es posible usar lo que se conoce como funciones anidadas o internas, que no es mas que una función dentro de otra. La función principal se conoce como externa.

```
def operacion_cadena(texto):
    def mayusculas(letras):
        total = letras.upper()
        return total
    return mayusculas(texto).split()

print(operacion_cadena("Hola, me llamo Loreto"))
#['HOLA,', 'ME', 'LLAMO', 'LORETO']
```

3

Funciones nativas de Python

3. Funciones nativas

- Las funciones integradas en Python son un conjunto de funciones predefinidas que vienen incorporadas en el lenguaje.

- Estas ofrecen una amplia gama de utilidades y se pueden utilizar directamente sin necesidad de importar ningún módulo adicional.

3.1. Funciones numéricas

- La función **abs()** se utiliza para calcular el **valor absoluto** de un número. Devuelve el valor absoluto de x, es decir, el valor numérico sin signo.

```
print(abs(-10)) #10
```

3.1. Funciones numéricas

- La función **max()** se utiliza para encontrar el valor máximo en un iterable, como una lista o una tupla.
- Por ejemplo, puede funcionar para buscar la edad máxima en una lista de edades.

```
edades = [9, 12, 23, 14, 5]
print(max(edades)) #23
```

3.1. Funciones numéricas

- La función **min()** se utiliza para encontrar el valor mínimo en un iterable.
- La podemos utilizar para saber la edad mínima en una lista de edades.

```
edades = [9, 12, 23, 14, 5]
min(edades)  # 5
```

3.1. Funciones numéricas

- La función **sum()** se usa para calcular la suma de todos los elementos en un iterable. Si deseamos sumar todas las ventas de un día en una lista, podemos usarla.

```
ventas = [15, 22, 30, 47, 50]
sum(ventas)  # 164
```

3.1. Funciones numéricas

- La función **round()** se utiliza para imprimir el número de decimales especificado por n-dígitos. Pero no redondea el valor del número.
- Funciona cuando queremos asegurarnos de recortar los decimales en una variable.

```
pi = 3.14159
round(pi, 2)  # 3.14
```

3.1. Funciones numéricas

- La función **pow(base, exp)** se utiliza para elevar números, a partir de la base elevándolo con el exponente que se introduce en la función.

```
print(pow(5,2)) #25
```

3.1. Funciones numéricas

- La función **divmod(a,b)** devuelve el cociente y el resto al dividir el numero a del primer argumento por el argumento b de su segundo argumento.
  - Para los argumentos enteros, el valor de retorno será el mismo que (a//b,a%b)
  - Para los argumentos float, el valor de retorno será: (a/b,a%b)

```
print(divmod(5,2)) #(2,1)
# Se puede asignar a variables
cociente,resto =divmod(5,2)
print(cociente)
print(resto)
print(divmod(13.5,2.5)) #(5.0, 1.0)
```

3.1. Funciones numéricas

- La función **len()** se usa para obtener la longitud tanto de una cadena de texto, lista, tupla, …
- En la cadena devuelve el número de caracteres que contiene la cadena.

```
texto = "Hola, mundo!"
print(len("Hola, mundo!"))
```

3.1. Funciones numéricas

- Cuando se utiliza la función **len()** en una lista, tupla, .., cuenta el numero de elementos que hay.

```
ventas = [15, 22, 30, 47, 50]
print(len(ventas))
```

- En este caso hay 5 elementos.

¡Gracias!