FUNCIONES

Introducción

En programación algunos términos se pueden confundir y es el caso de las **funciones** y los **métodos.**

La principal diferencia es que un método es parte de una clase, es decir, es parte de la funcionalidad que le damos a un objeto. Por tanto, siempre va a estar asociado a un objeto. Sin embargo, las funciones en Python, están definidas por si mismas y no pertenecen a ninguna clase.

En el siguiente ejemplo ¿Reconoces la diferencia, sabes quién es quién?

```
x = "abc"
print(x.upper())
```

En Python contamos con 4 tipos de funciones:

1. **Funciones integradas**
2. Funciones en módulos preinstalados
3. **Funciones definidas por el usuario**
4. Funciones Lambda

Funciones integradas

Son funciones que vienen incorporadas en Python y están disponibles para su uso sin necesidad de importar ningún módulo adicional.

Estas funciones son parte del núcleo del lenguaje y se utilizan para realizar tareas comunes, como operaciones matemáticas, manipulación de cadenas, entrada/salida de datos, entre otras.

Consulta aquí la Lista de funciones integradas

Ya hemos utilizado algunas de ellas como por ejemplo print(), len(), type(), input(), str(), int(), float(), max(), min(), sum(), entre otras.

Utilizando funciones integradas, termina los siguientes programas:

1. Dada la siguiente lista de frutas, utilice una funcion que permita seleccionar la cantidad de frutas que hay en la lista.

```
frutas = ['manzana', 'mango', 'papaya', 'banano', 'melon', 'pera', 'uva', 'piña']
```

2. Dada una lista de números, devolver el menor y el mayor de ellos

```
numeros =[5, 6, 2, 8, 3]
```

Funciones en módulos preinstalados

Son funciones que están disponibles en módulos preinstalados de Python. Estas funciones ofrecen funcionalidades adicionales más especializadas que no se encuentran en las funciones integradas. Para utilizar estas funciones, es necesario importar el módulo correspondiente usando la sentencia import.

Ejemplos de módulos preinstalados y sus funciones son math (por ejemplo, math.sqrt() para calcular la raíz cuadrada), random (por ejemplo, random.randint() para generar números aleatorios), os (por ejemplo, os.listdir() para obtener la lista de archivos en un directorio), entre otros.

Dada la siguiente lista de frutas, utilice una funcion en módulos preinstalados que permita seleccionar de manera aleatoria 3 frutas de la lista.

```
frutas = ['manzana', 'mango', 'papaya', 'banano', 'melon', 'pera', 'uva', 'piña']
```

```
import random
frutas = ['manzana', 'mango', 'papaya', 'banano', 'melon', 'pera', 'uva', 'piña']
random.sample(frutas,3)
```

Funciones definidas por el usuario

Son funciones que el programador define para realizar tareas específicas dentro de un programa. Estas funciones se crean utilizando la palabra clave **def** seguida del nombre de la función y sus parámetros, seguido de dos puntos : y el bloque de código que realiza la tarea deseada. El uso de funciones definidas por el usuario permite organizar y reutilizar el código de manera eficiente.

Definición e invocación de funciones

La sintaxis para definir una función es la siguiente:

```
def nombre_funcion():
    codigo_funcion
```

para invocar a una función previamente definida basta con escribir su nombre y en caso de tener parámetros enviarlos entre paréntesis.

```
# invocación de una función
nombre_funcion()
```

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

    suma(5,8)
```

Parámetros y Argumentos

En Python, cuando defines una función, puedes incluir variables llamadas parámetros dentro de los paréntesis, los parámetros son como marcadores de posición para los datos que la función espera recibir cuando es llamada.

Cuando llamas a la función y proporcionas valores concretos para esos parámetros, esos valores se llaman argumentos.

Una función puede recibir cero parametros

```
    def tabla_del_5():
        for i in range (1,11):
            print(f"5 x {i} = {5 * i}")

    tabla_del_5()
```

o varios parámetros.

```
    def tabla_del_numero(num, limite):
        for i in range (1,limite+1):
```

```
            print(f"{num} x {i} = {num * i}")

    tabla_del_numero(int(input("numero:")),int(input("limite")))
```

Argumentos por nombre y parámetros por defecto

En Python existen distintos tipos de argumentos que debemos de conocer ya que entenderlos nos ayudará muchísimo a la hora de programar y de consultar la documentación. En concreto debemos de distinguir entre:

a. Los argumentos posicionales: son argumentos que se pueden llamar por su posición en la definición de la función.

b. Los argumentos de palabras clave: son argumentos que se pueden llamar por su nombre.

NOTA:

Los argumentos obligatorios: son argumentos que se deben pasar a la función.

Los argumentos opcionales: son argumentos que no es necesario especificar. En Python, los argumentos opcionales son argumentos que tienen un valor predeterminado.

Miremos el siguiente código. ¿Puedes predecir los resultados antes de ejecutarlo?

```
    def dividir (x=2, y=2):
      res = x/y
      print (res)

    dividir (4, 3) #paso de argumentos por posición
    dividir(8)     #paso de argumento por defecto
    dividir(y=3, x=4) #paso de argumento por nombre
    dividir(5, y=5) #paso de argumento por posición y palabra clave
```

- Los parámetros pueden tener valores por defecto que sólo se tomarán si no se envía un argumento asociado al parámetro.
- Los argumentos son posicionales por defecto.
- Si se envían parámetros por palabra clave, el orden no importa.
- Si se mezclan argumentos por posición y por palabra clave, estos últimos deben ir al final.

Número de argumentos indeterminados

Algunas veces nos pueden llamar a una función con un número variable de argumentos. En ese caso, debemos preparar a la función para que pueda tomarlos y gestionarlos en su interior.

*args permite pasar un número variable de argumentos posicionales. Los argumentos se pasan como una tupla.

```
    def suma (*numeros):
      print (type(numeros))
```

```
  return sum(numeros)

print ("La suma de los numeros es :", suma (5, 2.4,5,3,8,9))
print ("La suma de los numeros es :", suma (5, 2))
print ("La suma de los numeros es :", suma (5))
```

Si además de recibir valores variables, queremos recibir los nombres de las variables podemos definir nuestro parámetro como un diccionario variable.

**kwargs permite pasar un número variable de argumentos de palabra clave. Los argumentos se pasan como un diccionario.

Veamos:

```
def mostrar_info(**datos):
    print(type(datos))
    for k in datos.items():
        print (k)

mostrar_info (nombre="Julian", apellido="Valencia")
#mostrar_info (nombre="Juan", apellido="Velez", edad = 32, tel ="32323")
```

Retorno de valores

Como en la mayoría de lenguajes de programación en Python empleamos la palabra reservada **return** para devolver un valor o resultado desde la función hacia la parte del programa donde fue invocada.

```
def sumar (x, y):
  return x + y

print (sumar(4,3))
```

En Python podemos tener funciones que retornen varios valores. En ese caso los valores retornados se tratarán como una Tupla. Miremos:

```
def mi_funcion():
    return 1, "a", 3.5

x = mi_funcion()
print (type(x))
print (x)
```

Al ser una tupla, podemos acceder a los valores devueltos a partir de su posición e incluso podemos realizar cualquier operación permitida para las tuplas..

```
def mi_funcion():
    return 1, "a", 3.5
```

```
for x in mi_funcion():
  print (x)
```

En este caso tomamos los valores de la tupla y se los asignamo a variables.

```
def procesar_numeros(numeros):

    maximo = max(numeros)
    minimo = min(numeros)
    suma = sum(numeros)
    promedio = suma/(len(numeros))
    return maximo, minimo, suma, promedio

# Ejemplo de uso de la función
numeros = [10, 5, 8, 20, 3]

maximo, minimo, suma, promedio = procesar_numeros(numeros)
x=procesar_numeros(numeros)
print("Máximo:", maximo)
print("Mínimo:", minimo)
print("Suma:", suma)
print("Promedio:", promedio)
```

¿Podrías predecir qué se imprime en los siguientes códigos?

```
def doblar_valor(x):
     x = x * 2
     print ("dentro de la funcion ", x)
x = 10
doblar_valor(x)
print(x)
```

- **Los tipos simples se pasan por valor**: *Enteros, flotantes, cadenas, lógicos...* Eso significa que cuando pasas un tipo simple a una función se crea una copia de ese valor y la función trabaja con esa copia, cualquier modificación hecha dentro de la función no afectará al valor original fuera de la función.

```
def doblar_valores(ns):
    ns [0] = ns[0] * 2
    ns [1] = ns[1] * 2
    ns [2] = ns[2] * 2
    print(f"Dentro de la función {ns}")

ns = [10,50,100]
doblar_valores(ns)
print(ns)
```

- **Los tipos compuestos se pasan por referencia:** *Listas, diccionarios, conjuntos...* Eso significa que cuando pasas un tipo compuesto a una función, se pasa una referencia al objeto original, por lo tanto cualquier modificación hecha dentro de la función afectará al objeto original fuera de la función.

Si no quisieramos que se modificar nuestra lista podemos pasar una copia de esta

```
def doblar_valores(ns):
    ns [0] = ns[0] * 2
    ns [1] = ns[1] * 2
    ns [2] = ns[2] * 2
    print(f"Dentro de la función {ns}")

ns = [10,50,100]
doblar_valores(ns[:])
print(ns)
```

Ámbito de las variables en una función

Con relación a las variables, los cambios en las variables al interior de una función no tienen efecto por fuera de ella. Veamos

```
def cambio_valor():
  x = 3

x = 8
cambio_valor()
print (x)
```

Sin embargo, con la declaración de una variable al interior de una función como **global** su valor se conservará por fuera de la función. Veamos

```
def cambio_valor():
  global x
  x = 3
x = 8
cambio_valor()
print (x)
```

Funciones recursivas

Una función recursiva es aquella que en su interior se invoca a sí misma.

```
def contar_hacia_atras(numero):
    if numero == 0:  # Caso base: cuando el número es 0, detener la recursión
```

```
            print("¡Boom!")
        else:
            print(numero)
            contar_hacia_atras(numero - 1)  # Llamar a la función con un número más pequeño

    contar_hacia_atras(5)
```

Aplicación

Imagina que estás desarrollando una aplicación de búsqueda de datos y necesitas buscar un elemento específico en una lista de elementos. Cada elemento puede ser una cadena de texto o una lista de elementos. Quieres encontrar el elemento sin importar cuán profundo esté en la lista.

```
    #x=0
    def buscar_elemento(elemento, lista):
        #global x
        #x += 1
        for item in lista:
            if type(item) == list:
                print("Elemento no encontrado:", item)
                encontrado = buscar_elemento(elemento, item)
                if encontrado:
                    return True
            elif item == elemento:
                print("Elemento encontrado:", elemento)
                return True
            else:
               print("Elemento no encontrado:", item)
        return False


    # Ejemplo de uso
    lista_datos = ["a", ["b", "c", ["d", "e"]], "f", ["g", "h"]]
    elemento_buscado = "h"
    if not buscar_elemento(elemento_buscado, lista_datos):
        print("Elemento no encontrado")
    #print(f"cantidad {x}")
```

Funciones Lambda

Son funciones anónimas y pequeñas que se definen sin necesidad de utilizar la palabra clave def.

Se definen utilizando la palabra clave lambda, seguida de los parámetros, luego dos puntos : y la expresión que define lo que hace la función.

Las funciones lambda son útiles cuando se necesita una función rápida para una operación simple y no es necesario reutilizarla en otras partes del código. Ejemplo:

```
cuadrado = lambda x: x ** 2
cuadrado(5)
```

Apropiación

1. Realiza una función que tome una lista de números enteros y devuelva dos listas ordenadas. La primera con los números pares y la segunda con los números impares.

```
Empieza a programar o a crear código con IA.
```

2. Realiza una función que reciba una palabra y devuelva un diccionario donde las claves sean las diferentes letras que se encuentran en la palabra y los valores sean el número de veces que se repite cada letra en la palabra

```

```

3. Realiza una función que reciba un número y devuelva una cadena con los nombres de los números recibidos, separando cada número con un guión medio. Por ejemplo, si el número recibido es 134, la función devolverá la cadena "uno - tres - cuatro"

4. ¡Madame Adivinadona te ha contratado para que le realices una aplicación en donde la persona ingresa su nombre y le va a decir cómo es y qué le depara el destino. Debes crear dos listas: una con adjetivos positivos y otra con frases que expresen lo que le depara el destino. Luego, debes usar estas listas en la función generar_mensaje_animo. Si la persona no ingresa el nombre debera iniciar con un 'Hola'

5. Se requiere una aplicación que oriente al usuario sobre la cantidad de pizzas que debe pedir de acuerdo al numero de personas y al tamaño de la pizza que este solicitando. En la pizeria hay pizzas pequeñas que alcanzan para 6 personas, medianas para 8 y grandez para 10, el programa debe informar cuantas rebanadas sobran y cuantos no alcanzan a repetir. Si no especifica el tamaño de la pizza se asume que es grande la que esta solicitando.

6. Se requiere una funcion que simule tirar los dado, deberá solicitar cuantas personas van a jugar, en cada turno se pedirá el nombre de la persona y la aplicación lanzará los dados, debe indicar cuanto saqué con cada dado y la suma de los dos, deberá guardar el nombre de la persona y el valor de la suma de los dos dados, al final deberá indicar qué jugador tuvo el mayor valor y ganó el juego.

7. Construir una función recursiva que lleve la cuenta de las veces que un aspirante intentó ingresar al SENA hasta que lo logra. La función tendrá definido en el código el puntaje mínimo para aprobar el examen de ingreso (70 puntos) y deberá recibir como argumento el puntaje que el aspirante sacó (valor aleatorio), mientras el aspirante no apruebe el examen, la función se invocará así misma hasta cuando por fin lo apruebe la función se detendrá y mostrará el número de intentos realizados por el aspirante.