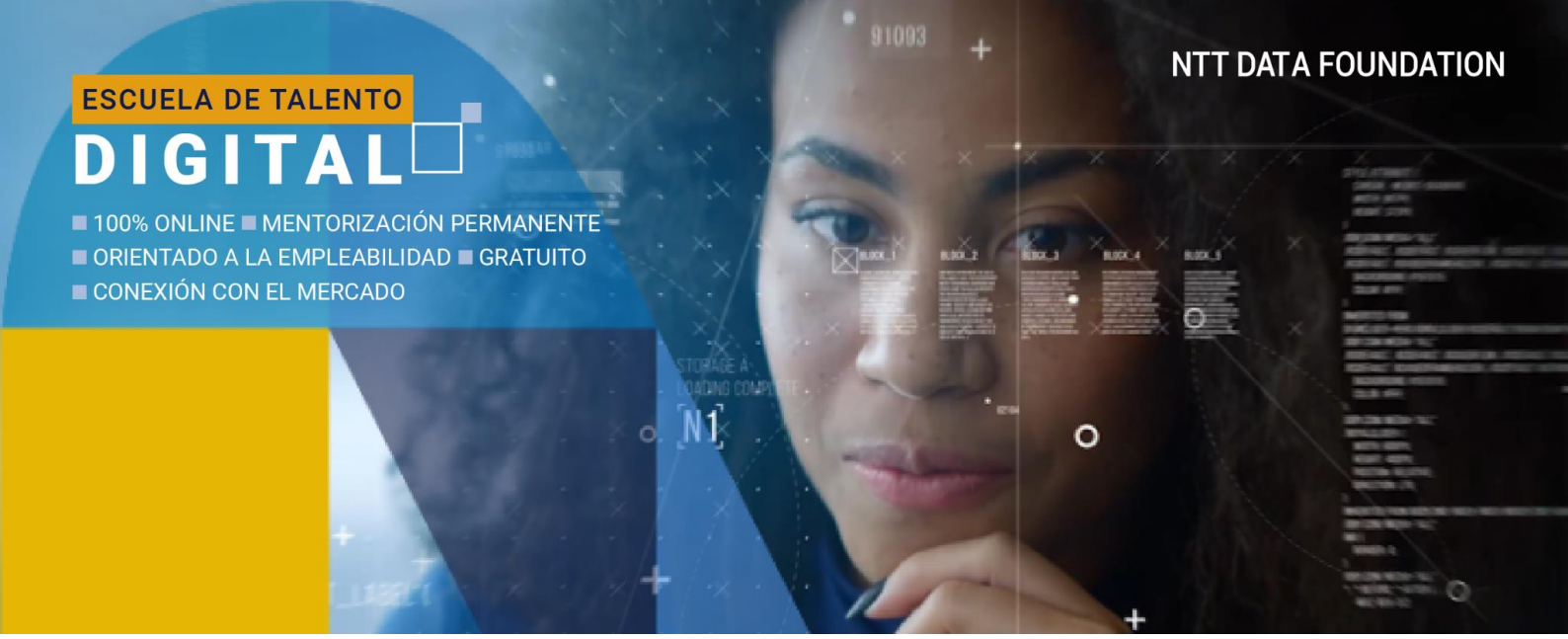

ESCUELA DE TALENTO DIGITAL

NTT DATA FOUNDATION

FUNCIONES

ÍNDICE

1. ¿QUÉ SON?

Las funciones son bloques de código reutilizables que se ejecutan dentro de un programa cuando se les llama. La forma en la que se define una función es la siguiente:

```
def nombre de la función ():

      código a ejecutar
```

Esto quiere decir que nuestras funciones tendrán un nombre y cuando en un programa las llamemos por ese nombre, se ejecutará el código de esas funciones sin necesidad de escribirlo de nuevo.

Por ejemplo, si tenemos una función que se llama `escribir` que cuando la llamas escribe, por ejemplo, `“Bienvenido”`, nuestra función sería:

```
def escribir ():

      print (“Bienvenid@”)
```

Cuando la llamemos dentro de un programa, lo haremos solo por su nombre:

```
nombre = input ("Introduce tu nombre: ")

print("Hola", nombre)

escribir()
```

Al ejecutar este programa obtendremos lo siguiente:

```
Introduce tu nombre: Juan

Hola Juan

Bienvenid@
```

Es decir, al ejecutar `escribir()` en el programa, automáticamente se ejecuta el código contenido en la función, sin necesidad de volver a escribir en el código del programa `print(“Bienvenid@”).`

2. PARÁMETROS

Normalmente, cuando creamos una función, utilizamos también parámetros, que son datos que le damos a la función para que pueda utilizarlos al ejecutar el código.

Por ejemplo, si tenemos una función que suma dos números, le tendremos que decir qué dos números son los que queremos que sume. Pues esos dos números son dos parámetros de entrada a la función.

Los parámetros se definen de forma genérica en el nombre de la función y les damos valores cuando llamamos a dicha función. Siguen la estructura:

```
def nombre de la función (parámetro1, parámetro2, …, parámetroN)

      código a ejecutar
```

Podemos incluir tantos parámetros como necesitemos. En el ejemplo anterior, la función `escribir ()` no tenía ningún parámetro, pero podríamos modificarla para que tuviera uno. Entonces nuestra función sería:

```
def escribir (nombre):

    print("Hola", nombre)

    print("Bienvenid@")
```

Cuando llamemos a nuestra función será cuando le demos un valor al parámetro genérico nombre, y lo haremos así:

```
escribir("Juan")
```

Que mostrará por pantalla:

```
Hola Juan

Bienvenid@
```

Veamos ahora el ejemplo de la suma de dos números que comentábamos anteriormente. Esta función tendría dos parámetros y la definiríamos así:

```
def suma (a, b):

      print(a+b)
```

Cuando la llamemos dentro de un programa, daremos valor a los dos parámetros. Por ejemplo, si dentro de un programa queremos que sume los números 4 y 5, llamaremos a la función así:

```
suma (4, 5)
```

Lo que mostrará por pantalla `9`.

2.1. Obligatoriedad

Cuando introducimos un parámetro en la definición de la función, también podemos indicarle si ese parámetro será obligatorio u opcional. Normalmente, cuando solo le damos un nombre genérico y no lo inicializamos, el parámetro será obligatorio, y cuando lo inicializamos con un valor concreto, será opcional. Es decir, como no tiene valor, es obligatorio que yo le introduzca un valor, pero si ya lo tiene, no es necesario que lo incluya. Eso sí, si lo hago, tomo el nuevo valor.

Veámoslo con unos ejemplos. En nuestra función `escribir` anterior, teníamos, como hemos visto, que darle un valor a nombre.

```
def escribir (nombre):
    print("Hola", nombre)
    print("Bienvenid@")
escribir (“Juan”)
```

Sin embargo, si nosotros definimos la función de la siguiente manera:

```
def escribir (nombre=“ ”):
    print("Hola", nombre)
    print("Bienvenid@")
```

Si no damos ningún valor a `nombre` y llamamos a la función `escribir(),` como `nombre` ya está inicializado a vacío, se mostrará por pantalla:

```
Hola
Bienvenid@
```

Sin embargo, si nosotros llamamos a la función de esta manera:

```
escribir(“Juan”)
```

Mostrará por pantalla:

```
Hola Juan
Bienvenid@
```

Porque ha sustituido el valor vacío por `Juan`.

Veámoslo también con la función `suma`. Si nosotros la definimos así:

```
def suma (a, b=0):
      print(a+b)
```

Le estamos diciendo que, si solo incluimos un parámetro en la llamada, la función tomará para `b` el valor `0`. Es decir:

`suma (4)` nos devolverá `4` porque 4 + 0 es igual a 4, mientras que `suma (4, 5)` nos devolverá `9` porque `b` ya no vale `0` sino que vale `5`.

2.2. Orden

Los parámetros no tienen por qué llamarse siempre, desde un programa, de forma ordenada, eso sí, si no los llamamos de forma ordenada, habrá que indicarle al programa a qué parámetro le damos cada valor. Veámoslo con un ejemplo:

Si tenemos la función `división` definida así:

```
def division (numerador, denominador):
      print(numerador/denominador)
```

Si la llamamos dentro de un programa así:

`division (5, 4)`, no tendremos ningún problema y nos devolverá `1.25`.

Sin embargo, si la llamamos así:

`division (4, 5)`, tampoco tendremos problema, pero el resultado será `0.8`.

Es decir, siempre toma como numerador el primer número y como denominador el segundo.

Por lo tanto, en el caso de que nosotros la llamemos así:

`division (5, 0)`, nos dará un error porque no existe la división por 0, pero si llamamos como `division (0, 5)` si tendrá un resultado, `0.0`.

En estos casos, para llamar a los parámetros de forma desordenada, lo tendremos que hacer así:

`division (denominador = 5, numerador = 0)` que nos devolverá `0.0` porque toma como numerador 0 y como denominador 5.

Y la función `division (denominador = 4, numerador = 5)` nos devolverá `1.25` porque, aunque hemos invertido el orden de los valores, al indicar cuál queremos que sea su valor, hace el cálculo igual que si llamamos `division (5,4)`.

2.3. Número de parámetros sin definir

En ocasiones, puede darse el caso de que no sepamos, de antemano, cuántos parámetros vamos a tener en nuestra función.

En estas situaciones tenemos dos opciones:

- Podemos indicar con un asterisco, *, que no sabemos cuántos parámetros vamos a recibir pero que, sean los que sean, los guarde en una lista y trabaje con ella.
- Podemos indicar con dos asteriscos, `**`, que no sabemos cuántos parámetros vamos a recibir pero que, sean los que sean, los guarde en un diccionario y trabaje con él.

Veámoslo con ejemplos.

Si tenemos la siguiente función:

```
def suma(*numeros):
    acumula = 0
    for i in numeros:
```

```
        acumula = acumula + i

    print(acumula)
```

Si la llamamos así `suma (1, 3)` nos mostrará por pantalla el valor `4`, pero si la llamamos como `suma (4, 3, 6, 1)` nos devolverá el valor `14`.

¿Cómo funciona?

Cuando la llamamos con dos parámetros, por ejemplo, 1 y 3, guarda todos los parámetros en una lista interna, `[1, 3]`. Y ejecuta el código:

- Inicializa la variable `acumula` a 0.
- Para `i = 1`, que es el primer elemento de la lista `numeros`, `acumula` será igual `acumula` más `i`, es decir, acumula = 0 +1 = 1.
- Vuelve al `for`, y ahora `i = 3`, el segundo elemento de la lista `numeros`, por lo tanto, acumula = 1 + 3 = 4.
- Vuelve al `for`, como no hay más elementos, sale del `for` sin ejecutar nada.
- Muestra por pantalla el valor de `acumula`, en este caso `4`.

Si llamamos a nuestra función con más números, los guardará en la lista y ejecutará el bucle hasta que no haya más elementos en ella.

En el caso de crear nuestra función con dos asteriscos, pasará lo mismo, pero, en lugar de en listas en diccionarios. Es decir:

```
def suma (**numeros):

    acumula = 0

    for clave in numeros:

        acumula = acumula + numeros[clave]

    print(acumula)
```

Ahora, para llamar a nuestra función, debemos declarar claves y valores, por lo que lo haríamos así:

```
suma (numero1 = 1, numero2 = 3)
```

Esta función nos mostraría por pantalla el número `4`, e internamente funcionaría así:

- Guarda en un diccionario `{“numero1”:1, “numero2”:3}`
- Inicializa `acumula` a 0.
- Para el valor de la primera clave en el diccionario, es decir, `“numero1”`, `acumula` será igual a `acumula` más el valor que hay en esa clave, esto es, acumula = 0 + 1 = 1
- Para el valor de la segunda clave en el diccionario, `“numero2”`, acumula = 1 + 3 = 4
- Como no hay más claves, muestra por pantalla `4.`

Y funcionaría igual para cualquier número de elementos. Por ejemplo, `suma(numero1 = 1,numero2 = 3,numero3 = 6,numero4 = 4)` devolverá `14`.

3. REUTILIZACIÓN

Como comentábamos al principio, la ventaja que ofrecen las funciones es que no tenemos que escribir el código que ejecutan una y otra vez, que solo con llamarlas ese código ya está implícito, además, podemos llamarlas tantas veces como consideremos necesario.

Por ejemplo, si tenemos el siguiente código:

```
def escribir (nombre):

    print("Hola", nombre)

    print("Bienvenid@")

nombres = ["Juan", "Ana", "Pablo", "Andrea"]

for i in nombres:

    escribir(i)
```

Nos mostrará por pantalla:

```
Hola Juan

Bienvenid@

Hola Ana

Bienvenid@

Hola Pablo

Bienvenid@

Hola Andrea

Bienvenid@
```

Es decir, el bucle `for` llamará a la función y ejecutará su código para cada uno de los elementos de la lista que hemos llamado `nombres` sin tener que escribirlo una y otra vez.

4. RETURN

Cuando no queremos solo mostrar el resultado por pantalla, sino que queremos que la función nos devuelva un resultado, utilizamos la sentencia `return`. Esta sentencia suele almacenar los resultados en una variable. Se utiliza de la siguiente manera:

```
def nombre de la función ():

      código a ejecutar

      return resultado que queremos que devuelva la función
```

Por ejemplo, la función `suma` que ya hemos estudiado tenía la forma siguiente:

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

Y esto mostraba la suma de a + b por pantalla sin guardar ese resultado en ningún sitio.

Sin embargo, si la definimos así:

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

Ahora la función nos devuelve un resultado que podremos guardar en cualquier variable.

Como veis no siempre es necesario que haya un código a ejecutar, el código se puede ejecutar directamente en la sentencia `return`.

Veamos otro ejemplo de uso de `return`. Si tenemos un programa que nos devuelve `True` si la suma de dos números es mayor que 15 y `False` en caso contrario, podríamos tener el siguiente código:

```
numero1 = int(input("Introduce un número: "))
numero2 = int(input("Introduce otro número: "))
suma = suma (numero1, numero2)
if suma > 15:
    print("True")
else:
    print("False")
```

En este caso, la función `suma (numero1, numero2)` guardará en una variable llamada `suma` el resultado de sumar los números `numero1` y `numero2`.

También podríamos utilizar el valor directamente sin necesidad de guardarlo en una variable. Es decir, el siguiente código realizará exactamente lo mismo que el anterior:

```
numero1 = int(input("Introduce un número: "))
numero2 = int(input("Introduce otro número: "))
if suma (numero1, numero2) > 15:
    print("True")
else:
    print("False")
```

Debéis tener en cuenta que cuando la función se encuentra con la sentencia `return`, automáticamente finaliza. Por lo tanto, hay situaciones, en que esta sentencia se puede utilizar para evitar que se ejecuten instrucciones innecesarias una vez alcanzado el objetivo deseado.

Por ejemplo, si queremos definir una función que cuente el número de caracteres que hay hasta la primera a, tendríamos el siguiente código:

```
def cuantos_hasta_a (texto):
    contador = 0
    for i in texto:
        if i == "a":
            return contador
        else:
            contador += 1
    return contador
```

Si la llamamos con `cuantos_hasta_a (“Hola me llamo Sandra”)`, cuando `i` está en la posición 3, el carácter en esa posición es la “a” y como `i = “a”`, entonces nos devuelve el contador y la función finaliza, no sigue mirando el resto de texto introducido como parámetro porque ha llegado a `return`.

La sentencia `return` también permite devolver varios datos al mismo tiempo. Para ello tendrá la siguiente estructura:

```
def nombre de función (parámetros):
      código a ejecutar
      return dato1, dato2, …, datoN
```

Por ejemplo, si queremos una función que nos devuelva el máximo de una lista de números, y cuál es el valor de ese máximo menos 1, podríamos tener el siguiente código:

```
def max_y_anterior (*numeros):
    max = 0
    for i in numeros:
        if i > max:
            max = i
    return max, max-1
```

Si llamamos a esta función `max_y_anterior (1, 6, 3)` nos devolverá el resultado `(6, 5)`, el máximo que es 6 y el valor de 6 - 1 que es 5.

También podríamos incluirla en el siguiente programa:

```
max, max_anterior = max_y_anterior (1, 6, 3)

print("El máximo es:", max)

print("El máximo menos 1 es:", max_anterior)
```

Que nos devolverá por pantalla:

```
El máximo es: 6

El máximo menos 1 es: 5
```

5. RECURSIVIDAD

Como hemos visto anteriormente, una función puede llamarse todas las veces que necesitemos dentro de un programa, pero también puede llamarse a sí misma desde dentro de su propio código. Esto se utiliza para iterar llamando a la misma función hasta que obtengamos el resultado que estamos buscando.

Por ejemplo, si tenemos el siguiente código:

```
def recursividad (x):
      if x > 0:
            print (“El valor de x es:”, x)
            recursividad (x-1)
      else:
            print (“Fin de las iteraciones”)
      return
```

La función se va a ejecutar hasta que el valor de `x` sea menor que 0, y se llamará a sí misma hasta entonces. Con la llamada `recursividad (3)` obtendremos los siguiente:

```
El valor de x es: 3

El valor de x es: 2

El valor de x es: 1

Fin de las iteraciones
```

Fijaos también en que aquí `return` no devuelve realmente nada, se utiliza, como comentábamos anteriormente, para detener la ejecución de la función.

6. USO DE VARIABLES

Otro punto que debéis tener en cuenta cuando utilicéis funciones es que las variables que se definen en un programa, fuera de una función, puede utilizarse en la función, pero, las variables que se definan dentro de la función no podrán usarse fuera de esta.

Por ejemplo, si utilizamos la función creada anteriormente, `cuantos_hasta_a`, como la variable `contador` está declarada dentro de la función, no la podemos llamar desde el programa. Es decir, con este código:

```
def cuantos_hasta_a (texto):
    contador = 0
    for i in texto:
        if i == "a":
            return contador
        else:
            contador += 1
    return contador
```

Si tenemos el programa:

```
texto = input (“Introduce un texto: “)
cuantos_hasta_a (texto)
print (contador)
```

Nos lanzará un error y nos dirá que `contador` no está definido.

Sin embargo, si tenemos el siguiente código:

```
contador = 0
def cuantos_hasta_a (texto):
    aux = contador
    for i in texto:
        if i == "a":
            return aux
        else:
            aux += 1
```

```
    return aux

texto = input ("Introduce un texto: ")

print(cuantos_hasta_a (texto))

print (contador)
```

Si ejecutamos el programa tendremos:

```
Introduce un texto: Hola me llamo Sandra

3

0
```

Donde, en el texto introducido, hay 3 letras hasta la primera a y `contador` tiene el valor 0.

7. MÁS INFORMACIÓN

En el siguiente vídeo podréis ver algunos ejemplos de cómo se puede trabajar con funciones:

https://www.youtube.com/watch?v=N_-YhYH_DyU&list=PLb_E6BNMg5j7-MJ0ctjvKQlv2PU7qbMDb&index=29