1000 PROGRAMADORES

Introducción a la
Programación en Python

MÓDULO 5

Funciones. Funciones integradas.
Retorno y envío de valores.
Argumentos, parámetros, valor y referencia.
Args vs kwargs. Argumentos indeterminados.
Manejo de excepciones.

Ministerio de Educación
Cultura, Ciencia y Tecnología
Gobierno de Salta

Ministerio de Economía
y Servicios Públicos
Gobierno de Salta

Universidad
Nacional de Salta

INTRODUCCIÓN

Las funciones en Python, y en cualquier lenguaje de programación, son estructuras esenciales de código.

Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto.

Como unidades lógicas de un programa tienen un doble objetivo:

- Dividir y organizar el código en partes más sencillas.
- Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Python define un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones, se denominan ***nativas*** o ***predefinidas*** o ***integradas***. Ya hemos utilizado algunas de ellas, como la función **`len()`**, que obtiene el número de elementos de un objeto contenedor como una lista, una tupla, un diccionario o un conjunto. También hemos visto la función **`print()`**, que muestra por consola un texto.

Sin embargo, como programadores, podemos definir nuestras propias funciones para estructurar el código de manera que sea más legible y para reutilizar aquellas partes que se repiten a lo largo de una aplicación. Esto es una tarea fundamental a medida que va creciendo el número de líneas de un programa.

El diseño de funciones parte de los siguientes principios:

- El **principio de reusabilidad**, si tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función. Nos evitaría tener código repetido y modificarlo sería más fácil, ya que bastaría con cambiar la función una vez.
- Y el **principio de modularidad**, en vez de escribir largos trozos de código, es mejor crear módulos o funciones que agrupen fragmentos en funcionalidades específicas, haciendo que el código resultante sea más fácil de leer.

FUNCIONES PREDEFINIDAS EN PYTHON

Vamos a describir algunas funciones internas más utilizadas y conocidas. Indicaremos su nombre y los argumentos necesarios para invocarlas acompañadas de un ejemplo de uso.

Funciones de Cadena

Función	Descripción	Ejemplo	Resultado
print()	Imprime en pantalla el argumento.	**print ("Hola")**	"Hola"
len()	Determina la longitud en caracteres de una cadena.	**len("Hola Python")**	11
join()	Convierte en cadena utilizando una separación	**Lista = ['Python', 'es']** **'-'.join(Lista)**	'Python-es'
split()	Convierte una cadena con un separador en una lista	**a = ("hola esto sera una lista")** **Lista2 = a.split()** **print (Lista2)**	['hola', 'esto', 'sera', 'una', 'lista']
replace()	Reemplaza una cadena por otra	**texto = "Manuel es mi amigo"** **print (texto.replace ('es', 'era'))**	Manuel era mi amigo
upper()	Convierte una cadena en Mayúsculas	**texto = "Manuel es mi amigo"** **texto.upper()**	'MANUEL ES MI AMIGO'
lower()	Convierte una cadena en Minúsculas	**texto = "MaNueL eS mI AmIgO"** **texto.lower()**	'manuel es mi amigo'

Funciones Numéricas

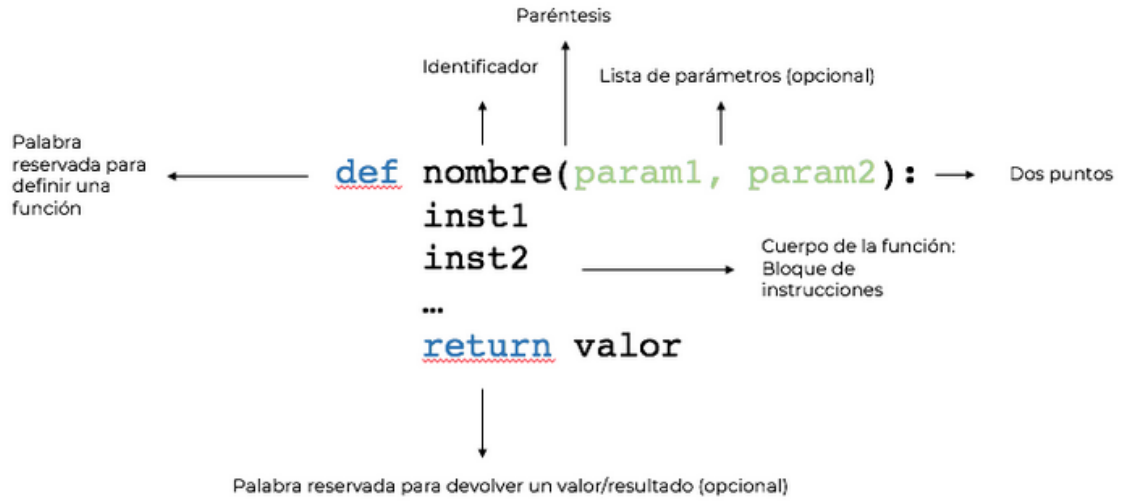
Función	Descripción	Ejemplo	Resultado
range()	Crea un rango de números	**x = range (5)** **print (list(x))**	[0, 1, 2, 3, 4]
str()	Convierte un valor numérico a texto	**str(22)**	'22'
int()	Convierte a valor entero	**int('22')**	22
float()	Convierte un valor a decimal	**float('2.22')**	2.22
max()	Determina el máximo entre un grupo de números	**x = [0, 1, 2]** **print (max(x))**	2
min()	Determina el mínimo entre un grupo de números	**x = [0, 1, 2]** **print (min(x))**	0
sum()	Suma el total de una lista de números	**x = [0, 1, 2]** **print (sum(x))**	3

Otras funciones útiles

Función	Descripción	Ejemplo	Resultado
list()	Crea una lista a partir de un elemento	**x = range (5)** **print (list(x))**	[0, 1, 2, 3, 4]
tuple()	Crea o convierte en una tupla	**print(tuple(x))**	(0, 1, 2, 3, 4)
ord()	Devuelve el valor ASCII de una cadena o carácter.	**print(ord('A'))**	65
round()	Redondea después de la coma de un decimal	**print (round(12.723))**	13
type()	Devuelve el tipo de un elemento	**type(x)**	<class 'range'>
input()	Permite la entrada de datos al usuario	**y = int(input("Ingrese el número"))** **print (y)**	3 3

DEFINICIÓN DE UNA FUNCIÓN

La siguiente imagen muestra el esquema de una función en Python:



Para definir una función se utiliza la palabra reservada def. A continuación el nombre o identificador de la función que es el que se utiliza para invocarla. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. La cabecera o definición de la función termina con dos puntos.

Tras los dos puntos se incluye el cuerpo de la función (identado) que es el conjunto de instrucciones que se encapsulan en dicha función y que le dan significado.

En último lugar y de manera opcional, se añade la instrucción con la palabra reservada return para devolver un resultado.

Para usar o invocar a una función, simplemente hay que escribir su nombre pasando los argumentos necesarios según los parámetros que defina la función.

Por lo tanto, los identificadores que se colocan al DEFINIR la función se denominan PARÁMETROS y los identificadores que se utilizan en la INVOCACIÓN de la función se llaman ARGUMENTOS. Es decir que los argumentos son los valores que se actualizan en el cuerpo de la función a través de los parámetros.

Veamos un ejemplo, donde simulamos una calculadora que realiza las operaciones básicas, el usuario ingresa dos operandos y puede elegir cuál operación realizar y además decidir si quiere realizar otro cálculo o no..

```
# Sección 1 ################################################
def sumar(a,b):                 #Definimos la función sumar
      x = a + b
      return x
def restar(a,b):                #Definimos la función restar
      x = a - b
      return x
def multiplicar(a,b):           #Definimos la función multiplicar
      x = a * b
      return x
def dividir(a,b):               #Definimos la función dividir
      x = a / b
      return x
# Sección 2 ################################################
operar = True
while operar:                   #Creamos un ciclo
      a = int(input("Ingresa el primer numero: \n"))
      b = int(input("Ingresa el segundo numero: \n"))
      op = input ("""Elegir cálculo
            1- Sumar
            2- Restar
            3- Multiplicar
            4- Dividir \n""")
# Sección 3 ################################################
      if op == '1':             #Llamamos a función sumar
         res = sumar(a,b)
         print("El resultado es: ",res)
      elif op == '2':           #Llamamos a función restar
         res = restar(a,b)
         print("El resultado es: ",res)
      elif op == '3':           #Llamamos a función multiplicar
         res = multiplicar(a,b)
         print("El resultado es: ",res)
      elif op == '4':           #Llamamos a función dividir
         res = dividir(a,b)
         print("El resultado es: ",res)
      else:
         print("""Número de opción erróneo""")
```

```
# Sección 4 ################################################
      s = input("Quieres hacer otra operación? S/N \n")
      if (s == 'N'):
         operar = False      # Para finalizar el ciclo
print("FIN de calculadora")  # Mensaje luego que corta el ciclo
```

```
Ingresa el primer numero:
1
Ingresa el segundo numero:
1
Elegir cálculo
        1- Sumar
        2- Restar
        3- Multiplicar
        4- Dividir
2
El resultado es:  0
Quieres hacer otra operación? S/N
N
FIN de calculadora
```

---

salida

En la **sección 1** se definen las cuatro funciones correspondientes al cálculo que habrá disponible, pero aún no son llamadas, osea no son utilizadas.

En la **sección 2** se crea un bucle o ciclo (que se repite mientras el usuario ingresa una opción distinta a “N”). Se solicitan los valores para ser operados y la operación deseada.

En la **sección 3** se determina cuál operación eligió el usuario y se invoca a una de las cuatro funciones diseñadas.

En la **sección 4** se da la posibilidad al usuario de realizar otra operación con otros elementos. Si ingresa “N”, se detiene el ciclo y termina el programa, si ingresa cualquier otro carácter el ciclo vuelve a iterar.

Argumento de entrada

Empecemos por una función sencilla sin parámetros de entrada ni salida.

```
def di_hola():          #Definición de la función
   print("Hola")
```

Luego de definir la función, el siguiente paso es invocarla y ver los resultados.

```
def di_hola():          #Definición de la función
   print("Hola")
di_hola()               #Invocación de la función
```

```
Hola
```

---

salida

Ahora usaremos un argumento de entrada, modificaremos un poco la salida.

```
def di_hola(nombre):          #Definición de la función
   print("Hola", nombre)
di_hola("Juan")               #Invocación de la función
```

```
Hola Juan
```

---

salida

Por defecto, los valores de los argumentos se asignan a los parámetros en el mismo ***orden*** en el que se los pasa al llamar a la función. Luego veremos que esto puede cambiar. Veamos un ejemplo de en qué afecta el orden de los parámetros.

```
def resta(a, b):              #Definición de la función
   return a-b
print(resta(5, 3))            #Invocación 1 de la función
print(resta(3, 5))            #Invocación 2 de la función
```

```
2                             #Resultado de 5-3
-2                            #Resultado de 3-5
```

---

salida

PARÁMETROS POR VALOR Y PARÁMETROS POR REFERENCIA

Una función puede definir, opcionalmente, una secuencia de parámetros que poseen una característica muy importante.

- **Paso por valor:** Un lenguaje de programación que lo utiliza, lo que hace es copiar el valor de las variables en los respectivos parámetros. Cualquier modificación del valor del parámetro, no afecta a la variable externa correspondiente.
- **Paso por referencia:** Un lenguaje de programación que lo utiliza, lo que hace es copiar en los parámetros la dirección de memoria de las variables que se usan como argumento. Hacen referencia al mismo objeto y cualquier modificación del valor en el parámetro afectará a la variable externa correspondiente.

En Python, si el tipo de dato que se pasa como argumento es inmutable, cualquier modificación en el valor del parámetro no afectará a la variable externa pero, si es mutable (como una lista o diccionario), sí se verá afectado por las modificaciones.

Veamos un ejemplo de paso por valor.

```
x = 10                          #La variable x toma el valor 10
def funcion(entrada):           #El parámetro entrada toma el valor 10
   entrada = entrada + 1        #El parámetro entrada toma el valor 11
funcion(x)                      #Invocación a la función con x = 10
print(x)
```

```
10    #La modificación en la función no repercute en el valor de x
```

---

salida

Veamos un ejemplo de paso por referencia.

```
x = [10, 20, 30]                #La variable x toma el valor 10
def funcion(entrada):           #El parámetro entrada toma el valor 10
   entrada[0]=entrada[0]+1      #El parámetro entrada toma el valor 11
funcion(x)                      #Invocación a la función con x = 10
print(x)
```

```
[11, 20, 30]      #La modificación en la función SI repercute en el
valor de x
```

---

salida

ARGS vs KWARGS

Python nos permite crear funciones que acepten un número indefinido de parámetros sin necesidad de que todos ellos aparezcan en la cabecera de la función. Los operadores * y ** son los que se utilizan para esta funcionalidad.

*Args

El parámetro especial *args en una función se usa para pasar, de forma opcional, un número variable de argumentos posicionales.

- Quien indica que el parámetro es de este tipo es el símbolo '*', el nombre args se usa por convención.
- El parámetro recibe los argumentos como una tupla.
- Es un parámetro opcional.
- El número de argumentos al invocar a la función es variable.
- Son parámetros posicionales, es decir, su valor depende de la posición en la que se pasen a la función.

La siguiente función toma dos parámetros y devuelve la suma de los mismos:

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

Si llamamos a la función con los valores 2 y 3, el resultado devuelto será 5.

```
sum(2, 3)
```

```
5
```

---

salida

Pero, ¿si necesitamos sumar un valor más?

```
sum(2, 3, 4)
```

```
TypeError: sum() takes 2 positional arguments but 3 were given
```

---

salida

Por supuesto que esperábamos que la llamada falle. Una posible solución sería agregar parámetros, pero tampoco sabríamos cuántos.

La mejor solución y más elegante es hacer uso de **args* en la definición de esta función. De este modo, podemos pasar tantos argumentos como queramos.

Primero hay que rediseñar la función *sum* y con esa nueva implementación, podemos llamar a la función con cualquier número variable de valores:

```
#Definición de la función
def sum(*args):
    value = 0
    for n in args:
        value += n
    return value

#Invocaciones de la función
print(sum())
print(sum(4,5))
print(sum(4,5,24,11))
```

```
0     #Resultado de la invocación sin argumentos
9     #Resultado de la invocación con 2 argumentos
44    #Resultado de la invocación con 4 argumentos
```

salida

El comportamiento de la función siempre es el mismo, con independencia del número de argumentos que pasamos.

Con esto resolvemos nuestro problema inicial, en el que necesitábamos un número variable de argumentos.

*Kwargs

El parámetro especial **kwargs en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

Las principales diferencias con respecto **args* son:

- Lo que indica que el parámetro es de este tipo es el símbolo '**', el nombre kwargs se usa por convención.
- El parámetro recibe los argumentos como un diccionario.
- Al tratarse de un diccionario, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves del diccionario.

Veamos un ejemplo sencillo de cómo ** se asocia a un diccionario

```
def fun(**params):
  print(params)
fun(x=5, y=8)
fun(x=5, y=8, z=4)
```

```
{'x': 5, 'y': 8}
{'x': 5, 'y': 8, 'z': 4}
```

salida

También es posible que la cabecera de una función utilice uno o varios argumentos posicionales, seguidos del operador * o **. Esto nos proporciona bastante flexibilidad a la hora de invocar a una función.

En el siguiente ejemplo tenemos 2 parámetros "fijos" y uno que permite agregar tantos como necesitemos.

```
def print_record(nombre, apellido, **rec):
  print("Nombre: ", nombre)
  print("Apellidos:", apellido)
  for k in rec:
    print("{0}: {1}".format(k, rec[k]))

print_record("Juan", "Coll", edad=43, localidad="Madrid")
```

```
Nombre:  Juan
Apellidos: Coll
edad: 43
localidad: Madrid
```

salida

Veamos otra invocación de la función:

```
print_record("Juan", "Coll", localidad="Madrid", edad=43)
```

```
Nombre:  Juan
Apellidos: Coll
localidad: Madrid      #Se invirtió el orden los argumentos
edad: 43
```

---

salida

Este tipo de parámetros es muy útil al realizar consultas sobre una base de datos, tema que veremos y ejemplificaremos más adelante.

Al combinar parámetros en la definición de una función hay que tener muy en cuenta el orden. Tanto *args como **kwargs pueden aparecer de forma conjunta o individual, pero siempre al final y de la siguiente manera:

```
def ejemplo(arg1, arg2, *args, **kwargs)
```

En resumen:

- Utiliza *args para pasar de forma opcional a una función un número variable de argumentos posicionales.
- El parámetro *args recibe los argumentos como una tupla.
- Emplea **kwargs para pasar de forma opcional a una función un número variable de argumentos con nombre.
- El parámetro **kwargs recibe los argumentos como un diccionario.

Por último, utilizar *args y **kwargs ahorra tiempo y esfuerzo en cuanto a análisis e implementación dado que flexibiliza el diseño de funciones, pero su uso indiscriminado puede originar resultados inesperados cuya revisión y arreglo nos podría llevar más tiempo aún que el diseño estratégico original.

MANEJO DE EXCEPCIONES

Llamamos excepciones en Python a los errores generados por nuestro código fuente. Si alguna función de nuestro programa genera un error y esta no lo maneja, el mismo se propaga hasta llegar a la función principal que la invocó y genera que nuestro programa se detenga.

Manejar los errores nos va a permitir evitar que nuestro programa deje de funcionar de golpe y nos va a dar la posibilidad de mostrar un error personalizado al usuario en vez de los clásicos errores del intérprete Python.

Para ello recurrimos a ciertas palabras reservadas que nos van a permitir realizar algunas acciones antes de detener nuestro programa por completo. Normalmente los errores pueden provenir de un error de cálculos o del ingreso de un dato del usuario que nuestro código no es capaz de procesar, entre otros.

Imaginemos que queremos dividir dos números a y b. Si, por ejemplo, b tiene el valor cero, este programa dará un error y su ejecución terminará de manera abrupta.

```
a = 5
b = 0
print("a/b = ",a/b)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-90888ba7a852> in <module>()
      1 a = 5
      2 b = 0
----> 3 print("a/b = ",a/b)
ZeroDivisionError: division by zero
```

salida

Ese “error” que decimos que ha ocurrido es lanzado por Python (raise en Inglés) ya que la división entre cero es una operación que matemáticamente no está definida. Se trata de la excepción `ZeroDivisionError`.

Una primera aproximación al control de excepciones, en nuestro primer ejemplo, podría ser realizar una comprobación manual de que no estamos dividiendo por cero, para así evitar el error.

```
a = 5
b = 0
if b!=0:
   print(a/b)
else:
   print("No se puede dividir!")
```

```
No se puede dividir!
```

salida

Veamos un ejemplo con otra excepción. Intentemos sumar un número con un texto. Python define una excepción para esto llamada `TypeError`.

```
print(2 + "2")
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-2-96788054896b> in <module>()
----> 1 print(2 + "2")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

salida

Un ejemplo más. Si intentamos acceder a la lista utilizando el valor de índice 2, ocurrirá un error:

```
li = [0, 1]
print(li[2])
```

```
IndexError                                Traceback (most recent call last)
<ipython-input-20-fcd9bda3ce0e> in <module>()
      1 li = [0, 1]
----> 2 print(li[2])
IndexError: list index out of range
```

salida

Suponiendo que las líneas anteriores de código forman parte de un programa, lo que ocurriría al ejecutar la última de ellas, es que el intérprete detendría automáticamente

la ejecución del programa. Para evitar esto, podemos capturar la excepción IndexError, la cual ha sido lanzada al detectar el error:

```
li = [0, 1]
try:
   print(li[2])
except IndexError:
   print("Error: índice no válido")
```

```
Error: índice no válido
```

salida

Empleando el código anterior la ejecución no será detenida y el intérprete continuará ejecutando el programa.

El bloque **`try/except`** es el utilizado para capturar excepciones.

Justo después de la palabra clave **`except`** debemos indicar el tipo de excepción que deseamos detectar. Por defecto, si no indicamos ninguna, cualquier excepción será capturada.

A partir de la sentencia **`try`**, se tendrá en cuenta cualquier línea de código y si, como consecuencia de la ejecución de una de ellas, se produce una excepción serán ejecutadas las sentencias de código que aparecen dentro de la sentencia except. En concreto, la sintaxis de la cláusula **`try/except`** es como sigue:

```
try:
   <sentencias_susceptibles_de_lanzar_error>
except [<Nombre_excepcion>]:
   <sentencias_ejecutadas_cuando_error>
finally:
   <sentencias_ejecutadas_siempre>
else:
   <sentencias_ejecutadas_si_no_error>
```

Como ejemplo para ilustrar la sintaxis de **`try/except`** modificaremos el último ejemplo de código:

```
li = [0, 1]
try:
   li [2]
except:
   print("Error: índice no válido")
else:
   print("Sin error")
finally:
   print("Bloque ejecutado")
```

```
Error: índice no válido
Bloque ejecutado
```

salida

La sentencia **`raise`** se encarga de lanzar excepciones. La manera más simple es invocarla directamente seguida del nombre de la excepción que deseamos lanzar:

```
li = [0, 1]
try:
   li [2]
   raise IndexError
except:
   print("Error de Indice")
```

```
Error de Indice
```

salida

Excepciones definidas por el usuario

El programador puede crear sus propios tipos de excepciones y lanzarlas cuando sea necesario.
El nombre de excepción definido por el usuario debe corresponder a una clase que herede de la clase **`Exception`**, la cual está definida en la librería estándar de Python. La clase en cuestión contendrá las acciones que deben ser ejecutadas cuando la excepción es lanzada.
Veremos ejemplos de este tema luego de desarrollar el módulo correspondiente a Programación Orientada a Objetos (POO) por la vinculación de contenidos existente.

BIBLIOGRAFÍA

[1] Python para todos. R. Gonzalez Duque. Licencia Creative Commons Reconocimiento 2.5 España.
Descarga gratuita de versión más reciente en http://mundogeek.net/tutorial-python/

[2] Python 3 al descubierto 2da Edición. A. Fernández Montoro. Alfaomega. Año 2013

[3] Tutoriales Web varios.