PRO

UT3: Diseño de programas en Python

José Antonio
Benítez Chacón

Índice

1. Funciones

Las funciones son bloques de código o secuencias de instrucciones que se agrupan con un identificador, para poder reutilizar ese código en diferentes partes del código sin tener que duplicarlo.

Los valores que necesita una función como entrada para ejecutarse se llaman parámetros. Además, en Python, una función puede o no devolver un resultado.

Para definirla se utiliza la palabra reservada **def**, que precede al indicador de la función y los parámetros de la misma. Esta definición indica el comportamiento que tendrá la función cuando sea llamada, pero de por sí, sin ser llamada, no se ejecuta ninguna vez.

```
def genera_lista(a,b,n):
    lista = random.sample( range(a,b),n)
    lista.sort()
    return lista

print(genera_lista( 100,120,5))
```

1. Funciones

Las funciones, pueden no recibir parámetros de entrada. En este caso hay dos opciones: bien que la función siempre ejecute el mismo código y realice exactamente lo mismo o bien que aunque no tenga parámetros de entrada, internamente accede a ficheros, fuentes de datos o variables globales que hayan podido ser modificadas por otras funciones.

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

def saludo():
    print("Buenas tardes.")

def saludo2():
    print("Buenas tardes, "+nombre+".")

saludo()
saludo2()
```

1. Funciones

La forma más común de llamar a una función es con un número determinado de parámetros, de forma que la posición de los mismos indican a qué se refieren:

```
from datetime import datetime

def imprimeFecha(anyo:int, mes:str, dia:int):
   mes_lower=mes.lower()
   mes_num = 0
   match mes_lower:
       case 'enero': mes_num = 1;
       case 'febrero': mes_num = 2
       case 'marzo': mes_num = 3
       case 'abril': mes_num = 4
       case 'mayo': mes_num = 5
       case 'junio': mes_num = 6
       case 'julio': mes_num = 7
       case 'agosto': mes_num = 8
       case 'septiembre': mes_num = 9
       case 'octubre': mes_num = 10
       case 'noviembre': mes_num = 11
       case 'diciembre': mes_num = 12
   fecha = datetime(month=mes_num, day=dia, year=anyo)
   print(fecha)

imprimeFecha(2013,"MAYO",3)
```

1. Funciones

Otra forma de pasar los parámetros es por nombre, en este caso se pueden pasar en orden diferente al que aparecen en la definición de la función:

```
from datetime import datetime


def imprimeFecha(anyo:int, mes:str, dia:int):
   mes_lower=mes.lower()
   mes_num = 0
   match mes lower:
       case 'enero': mes num = 1;
       case 'febrero': mes_num = 2
       case 'marzo': mes_num = 3
       case 'abril': mes_num = 4
       case 'mayo': mes num = 5
       case 'junio': mes num = 6
       case 'julio': mes_num = 7
       case 'agosto': mes_num = 8
       case 'septiembre': mes_num = 9
       case 'octubre': mes num = 10
       case 'noviembre': mes num = 11
       case 'diciembre': mes_num = 12
   fecha = datetime(month=mes_num, day=dia, year=anyo)
   print(fecha)

imprimeFecha(mes="Junio", dia=17, anyo=2020)
```

1. Funciones

Además, las funciones pueden tener unos valores establecidos por defecto para los parámetros, de manera que si los parámetros requeridos no se pasan, toman los valores por defecto.

```
alumnos = {
   "Ruiz Rocasolano, Juan" : {"localidad": "Sevilla", "ciclo": "DAM", "curso": 1},
   "García Pérez, Isabel" : {"localidad": "Brenes", "ciclo": "DAW", "curso": 2},
   "García Pérez, Emilia" : {"localidad": "Brenes", "ciclo": "DAW", "curso": 1},
   "Vázquez Fonseca, Martín" : {"localidad": "San Juan de Aznalfarache" , "ciclo": "DAM", "curso": 2}
}


def incluyeAlumnoEnDic (nombre, localidad= "Sevilla", curso=1, ciclo="DAM"):
   existeAlumno =  True
   if nombre not in alumnos:
       existeAlumno =  False
       alumnos[nombre] = { "localidad": localidad, "ciclo": ciclo, "curso": curso}
   else:
       alumnos[nombre][ "localidad"] = localidad
       alumnos[nombre][ "ciclo"] = ciclo
       alumnos[nombre][ "curso"] = curso

   cadenaSalida =  f"Alumno: {nombre}. Localidad: {localidad}. Cursa {curso}º de {ciclo}."
   if existeAlumno:
       print("ACTUALIZACIÓN:", cadenaSalida)
   else:
       print("ALTA:", cadenaSalida)


incluyeAlumnoEnDic( "Guzmán Rodríguez, Pedro" )
incluyeAlumnoEnDic( "Chacón Gutiérrez, Alba" , ciclo="DAW")
incluyeAlumnoEnDic( ciclo="DAM", nombre="García Pérez, Isabel" )
incluyeAlumnoEnDic( localidad="Camas", ciclo="DAW", curso=2, nombre="Tatay López, Cayetano" )
print(alumnos)
```

1. Funciones

Si queremos utilizar una función para un número indeterminado de parámetros, utilizaremos lo que se conoce como funciones *args (en los que se le pondrá de entrada un único parámetro con un asterisco delante, que significará una lista de parámetros de entrada. Esta lista podremos recorrerla o utilizar operaciones de agregación sobre la misma.

```
def calculaMedia(*lista):
   return sum(lista) / len(lista)

media1 = calculaMedia(1, 2, 3, 4)
print(f"La primera media calculada es {media1:.2f}")
print(f"La segunda media calculada es {calculaMedia(11, 2.4, 27.1, 40.4, 23.6, 130, -7, -2, 31.4):.2f}")
```

```
def cuentaPares(*numeros):
   contador = 0
   for numero in numeros:
       if numero % 2 == 0:
           contador +=1
   return contador

lista = [0,4,3,1,6,2,9,178,73,21,104]
lista2 = [1,3,5,7]
#NOTA: con el operador * descomponemos las listas en valores individuales
print(f"En la lista {lista} hay {cuentaPares(*lista)} números pares")
print(f"En la lista {lista2} hay {cuentaPares(*lista2)} números pares")
print(f"En la lista [3, 3, 2, 0, 12] hay {cuentaPares(3,3,2,0,12)} números pares")
```

1. Funciones

Si una función puede tener un número indeterminado de parámetros, pero vamos a pasarlos/recuperarlos por nombre, en tal caso utilizamos una función **kwargs. En este caso, los dos asteriscos indican que se pasa un mapa de claves:valores, que se puede recuperar para que funcione.

```
import math
def opera(**argumentos):
   resultado = 0
   if "operador" in argumentos:
       if argumentos["operador"] == "suma":
           if( "elemento1" in argumentos and "elemento2" in argumentos):
              resultado = argumentos[ "elemento1"] + argumentos["elemento2"]
           else:
               print("Para sumar se necesita definir elemento1 y elemento2" )
       elif argumentos["operador"] == "resta":
           if( "elemento1" in argumentos and "elemento2" in argumentos):
              resultado = argumentos[ "elemento1"] -argumentos["elemento2"]
           else:
               print("Para restar se necesita definir elemento1 y elemento2" )
       elif argumentos["operador"] == "factorial":
           if( "elemento1" in argumentos):
              resultado = math.factorial(argumentos[ "elemento1"])
           else:
               print("Para hacer el factorial se necesita definir elemento1" )
       else:
           print("Operador no válido")
   else:
       print("Ha de pasarse un operador")
   return resultado

print(opera(operador="suma",elemento1=3, elemento2=13.2))
print(opera(elemento1=30, elemento2=5.72, operador="resta"))
print(opera(elemento1=7, operador="factorial"))
print(opera(elemento2=5, operador="factorial"))
```

1. Funciones

Como ya se ha comentado, una función puede devolver (o no) un valor. Si la función devuelve un valor (tipo básico o estructura), este puede utilizarse directamente o almacenarse en una variable para su uso posterior (especialmente si vamos a utilizarla en más de una ocasión)

```
import random

def imprimeAletorio(min, max):
    print("Genero un número aleatorio: " +str(random.randint(min, max)))

def devuelveAleatorio(min, max):
    return random.randint(min, max)

#No devuelve nada, solo imprime
imprimeAletorio(1, 100)

#Devuelve un aleatorio que guardamos para uso posterior
num_ale=devuelveAleatorio(1,100)
if(num_ale < 50):
    print(f"El número aleatorio {num_ale} es menor que 50")
else:
    print(f"El número aleatorio {num_ale} es mayor o igual que 50")
```

1. Funciones

Otro ejemplo:

```
import random

def generaListaAleatorios (min, max, num):
    return random.choices( range(min, max), k=num)

listaAle = generaListaAleatorios( 1,100,10)
numPares = 0
listaCuadrados = list()
for i in range(len(listaAle)):
    if listaAle[i] % 2 == 0:
        numPares += 1
    listaCuadrados.append(listaAle[i]** 2)
print(f"Para la lista de aleatorios:  {listaAle}")
print(f"hay {numPares} elementos pares")
print(f"La lista de los cuadrados de los elementos es:  {listaCuadrados}")
```

1. Funciones

Una función tiene visibilidad al resto de variables definidas por encima de ella (y aquellas definidas como globales en otros puntos). Puede utilizar su valor y modificarlo, pero solo dentro del ámbito de la función. Luego se respetaría el valor que tuviese.

```
variable1 = 100
print(variable1)

def devuelveValorEnFuncion():
   variable1 = 200
   return variable1

print(devuelveValorEnFuncion())
print(variable1)
```

```
100
200
100
```

Si queremos que en la función se modifique el valor de la variable externa, tenemos que definirla dentro como global:

```
variable1 = 100
print(variable1)

def devuelveValorEnFuncion():
   global variable1
   variable1 = 200
   return variable1
print(devuelveValorEnFuncion())
print(variable1)
```

```
100
200
200
```

2. Excepciones (try-except)

Las excepciones son eventos o errores inesperados que ocurren durante la ejecución de un programa y que interrumpe su flujo normal. En Python, el programa se detiene salvo que dicha excepción se capture y se maneje el error. Las excepciones pueden producirse por diversos motivos: divisiones por cero, acceso a archivos que no existen, accesos a posiciones de estructuras fuera de rango, etc.

```
def divide(numerador, denominador):
      resultado = numerador / denominador
      print(f"El resultado de la división es:  {resultado}")


print(divide(2,3))
print(divide(10, 0))
```

2. Excepciones (try-except)

Podemos controlar las diferentes excepciones que puedan producirse utilizando un bloque try-except para tratarlas:

```
def divide(numerador, denominador):
    resultado = 0
    try:
        resultado = numerador / denominador
        print(f"El resultado de la división es: {resultado}")
    except ZeroDivisionError:
        print("Error: No se puede dividir entre cero.")
    except TypeError:
        print("Error: Los valores deben ser numéricos.")
    except Exception as e:
        print(f"Se ha producido un error inesperado: {e}")

# Ejemplos de uso
divide(10, 2)
divide(10, 0)
divide(10, "a")
```

2. Excepciones (try-except)

Otro ejemplo:

```
cadena = "María Jiménzez"
print(f"La posición de la letra o en  \"{cadena}\" es {cadena.index("o")}")
```

Controlando la excepción:

```
cadena = "María Jiménzez"
try:
   print(f"La posición de la letra o en  \"{cadena}\" es {cadena.index("o")}")
except ValueError:
   print(f"La letra  \"o\" no se encuentra en  \"{cadena}\"")
```

2. Excepciones (try-except)

Y otro ejemplo más:

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

for i in range(10):
   print(lista[i])
```

Controlando la excepción:

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

for i in range(10):
   try:
       print(lista[i])
   except IndexError:
       print(f"No existe índice {i} en la lista que se está recorriendo" )
```

2. Excepciones (try-except)

Y otro:

```
try:
   ejercicio = int(input("Elige un número de ejercicio para ejecutar" ))
   if ejercicio == 1:
       print("Ejercicio 1")
   elif ejercicio == 2:
       print("Ejercicio 2")
   else:
       print("El número de ejercicio seleccionado no está disponible" )
except Exception as e:
   print(f"Error: {e}")
```

2. Excepciones (try-except)

Nosotros también podemos crear excepciones en las funciones o programas que desarrollemos (y que podrán capturarse en otra parte):

```
def verificarEdad(edad):
    if edad < 0:
        raise ValueError("La edad no puede ser negativa")
    elif edad < 18:
        print("Menor de edad.")
    else:
        print("Mayor de edad.")

try:
    verificarEdad(23)
    verificarEdad(-5)
except ValueError as e:
    print(f"Error: {e}")
```

3. Expresiones Lambda y funciones de orden superior

Las **expresiones lambda** son funciones anónimas (sin nombre) en Python que se utilizan para crear funciones pequeñas y rápidas en una sola línea de código. Estas expresiones se definen con la palabra clave **lambda** y se usan comúnmente para operaciones sencillas y temporales, donde crear una función completa resultaría poco práctico.

En general, se emplean para funciones que tienen un propósito muy específico y no requieren más de una o dos líneas de código.

Sintaxis básica

```
lambda parámetros: expresión
```

Ejemplo simple

```
suma = lambda x, y: x + y
print(suma(5, 3))
```

3. Expresiones Lambda y funciones de orden superior

Estructura de una función lambda

Las **funciones lambda**:

- se definen usando **lambda** en vez de **def**
- no tienen nombre explícito (son funciones anónimas)
- la lista de parámetros va justo después de lambda, seguida de dos puntos ":"
- la expresión de retorno se coloca directamente después de los dos puntos

NOTA:
Al ser una función en una sola línea, no lleva instrucciones de control como **if**, **for** o **while**, pero sí admite operadores ternarios como:
`(opción_si_condicion_true if condicion else opcion_si_condicion_false).`

¿Cuándo Usar Lambdas?

- Para funciones pequeñas y temporales.
- En lugar de funciones definidas cuando se necesita una función rápida que se usa solo una vez.
- En **funciones de orden superior** como **map()**, **filter()**, y **sorted()**, que permiten aplicar una función en colecciones.

3. Expresiones Lambda y funciones de orden superior

Lambdas como funciones básicas

```
doble = lambda x: x * 2
print(doble(4))
```

Lambdas en una operación condicional

```
mayor = lambda x, y: x if x >
y else y
print(mayor(10, 20))
```

3. Expresiones Lambda y funciones de orden superior

Uso de lambdas con map()

map(función, iterable) aplica una función a cada elemento de una colección iterable, generando una nueva colección con los resultados.

Ejemplo:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)
```

Otro ejemplo:

```
palabras = ["lata", "pera", "nieve", "lluvia", "abrigo"]
sustitucion = list(map(lambda elem: elem.replace("a","*"),
palabras))
print(sustitucion)
```

3. Expresiones Lambda y funciones de orden superior

Uso de lambdas con filter()

filter(función, iterable) aplica filtra una colección iterable en base al resultado de una función, generando una nueva colección con los resultados.

Ejemplo:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)
```

Otro ejemplo:

```
colores = ["rojo", "naranja", "amarillo", "verde", "azul", "añil",
"violeta"]
coloresLargos = list(filter(lambda color: len(color)> 4, colores))
print(coloresLargos)
```

3. Expresiones Lambda y funciones de orden superior

Uso de lambdas con sorted()

sorted(iterable, key=ordenacion, reversed) aplica una ordenación a una colección iterable en base a una clave de ordenación (generalmente indicada por una función lambda) y que puede estar invertida (por defecto, si no se indica tercer parámetro, es equivalente a reversed=False).

Ejemplo:

```
parejas = [(1, 'b'), (2, 'a'), (4, 'd'), (3, 'c')]
ordenadas = sorted(parejas, key=lambda x: x[1])
print(ordenadas)
```

Otro ejemplo:

```
colores = ["rojo", "naranja", "amarillo", "verde", "azul", "añil", "violeta", "rosa chicle", "blanco
roto"]
coloresOrden = sorted(colores, key=lambda color: len(color), reverse=True)
print(coloresOrden)
```

3. Expresiones Lambda y funciones de orden superior

Uso de lambdas con reduce()

reduce(función, iterable) aplica una función acumulativa a los elementos de una colección iterable.

Ejemplo:

```
from functools import reduce

suma_total = reduce( lambda x, y: x + y, [1, 2, 3, 4])
print(suma_total)
```

Otro ejemplo:

```
from functools import reduce

palabras = ["Python", "es", "un", "lenguaje", "poderoso", "y", "popular"]
palabra_mas_larga = reduce( lambda x, y: x if len(x) > len(y) else y, palabras)
print(palabra_mas_larga)
```

4. Flujos de datos y Sistemas de Ficheros

En python podemos considerar dos tipos de flujos de datos: bytes y caracteres.

Según cómo se vaya a tratar un flujo de datos y cómo queremos abrirlo, hay que indicar las siguientes opciones:
t para abrir en modo texto y **b** para abrir en modo binario.
r (read) si queremos abrir el fichero solo para lectura, **w** (write) si queremos abrir el fichero solo para escritura eliminando el contenido previo (si lo hubiese), **a** (append) para abrir solo en modo escritura añadiendo contenido al final, sin eliminar el existente y **x** (exclusive creation) para asegurarnos que el fichero no existe previamente.

Modos Combinados con Texto/Binario:
rt: Lectura en modo texto (equivalente a r).
rb: Lectura en modo binario. Este es el modo que usamos para leer datos binarios como imágenes o archivos de audio.
wt: Escritura en modo texto (equivalente a w).
wb: Escritura en modo binario. Usado para escribir datos binarios, también común en archivos multimedia.
at: Agregar en modo texto (equivalente a a).
ab: Agregar en modo binario, para añadir datos binarios al final de un archivo.
r+: Lectura y escritura en el mismo archivo, sin eliminar el contenido existente. Requiere que el archivo exista.
w+: Escritura y lectura, eliminando el contenido existente. Si el archivo no existe, se crea uno nuevo.
a+: Agregar y lectura. Añade contenido al final del archivo y permite lectura. Si el archivo no existe, se crea uno nuevo.
rb+ y **wb+**: Lectura y escritura en modo binario.

4. Flujos de datos y Sistemas de Ficheros

Los flujos de datos de **bytes** se utilizan para manejar datos binarios (imágenes, audio, vídeo, etc.).

```
def eliminar_azul_bmp(input_path, output_path):
    with open(input_path, 'rb') as f:
        contenido = bytearray(f.read())  # Leer el archivo en formato binario

    # La cabecera de un archivo BMP tiene un offset donde empieza la información de los
píxeles
    # Normalmente, para BMP estándar, los primeros 54 bytes son de la cabecera
    offset = int.from_bytes(contenido[10:14], byteorder='little')

    # Modificar la sección de datos eliminando el color azul
    for i in range(offset, len(contenido), 3):
        contenido[i] = 0  # Establecer el componente azul (B) a 0

    # Guardar la imagen modificada
    with open(output_path, 'wb') as f:
        f.write(contenido)

# Ruta de la imagen de entrada y salida
input path = "OnePiece.bmp"
output_path = "imagen_sin_azul.bmp"

eliminar_azul_bmp(input_path, output_path)
```

4. Flujos de datos y Sistemas de Ficheros

Los flujos de datos de **caracteres** se utilizan para manejar texto.

```
# Definir los nombres de los archivos
archivo entrada =  "entrada.txt"
archivo_salida =  "salida.txt"

texto a reemplazar =  "Python"
nuevo_texto =  "JavaScript"

# Se abre el archivo de entrada en modo lectura
with open(archivo entrada,  "rt") as entrada:
    # Se leen todas las líneas y se guardan en una lista (cada elemento es una línea)
    contenido = entrada.readlines()

# Se inicializa la lista para el contenido modificado
contenido_modificado = []

# Se recorre cada línea y se modifica
for linea in contenido:
    linea modificada = linea.replace(texto a reemplazar, nuevo_texto)
    contenido_modificado.append(linea_modificada)

# Se guarda el contenido modificado en un nuevo archivo
with open(archivo salida,  "wt") as salida:
    # Escribir cada línea modificada
    salida.writelines(contenido_modificado)

print("Modificación completada. El archivo modificado se ha guardado en"  , archivo_salida)
```

NOTA: Usamos .bmp (bit map) porque si usásemos otros formatos (.jpg, .png, .gif, etc.) estos estarían comprimidos y necesitaríamos otras librerías (como Pillow) para tratarlos.

4. Flujos de datos y Sistemas de Ficheros

Clases y Funciones Relacionadas

Funciones integradas de Python: **open**, **read**, **write**, **close**.

Módulos útiles: io (para manejar flujos de bytes y texto de forma avanzada) y os (para operaciones con el sistema de archivos).

NOTA: Los ficheros SIEMPRE hay que cerrarlos. En los ejemplos que se han visto no es necesario cerrar manualmente los descriptores de archivo debido al uso de la instrucción **with** en Python.
La sintaxis **with open(...) as ...:** crea un contexto seguro que garantiza que los archivos se cierren automáticamente cuando el bloque de código dentro de with finaliza, incluso si ocurre una excepción.

Ejemplo con cierre manual:

```
archivo_entrada = open("entrada.txt", "r")
contenido = archivo_entrada.readlines()
archivo_entrada.close()  # Cerrar manualmente

contenido_modificado = []
for linea in contenido:
    linea_modificada = linea.replace("Python", "JavaScript")
    contenido_modificado.append(linea_modificada)

archivo_salida = open("salida.txt", "w")
archivo_salida.writelines(contenido_modificado)
archivo_salida.close()
```

4. Flujos de datos y Sistemas de Ficheros

Para crear y eliminar directorios podemos importar **os**.

Creación:

```
import os

os.makedirs('nueva_carpeta', exist_ok=True)  # Crear
directorio
with open('nueva carpeta/archivo.txt', 'w') as f:
    f.write('Contenido del archivo')
```

Borrado:

```
import os

os.remove('nueva carpeta/archivo.txt')  # Eliminar archivo
os.rmdir('nueva_carpeta')  # Eliminar directorio
```

4. Flujos de datos y Sistemas de Ficheros

Para navegar por directorios podemos importar **os** o **pathlib**.

Opción 1:

```
import os

directorio = "./nueva_carpeta"

# Listar archivos y directorios con detalles
with os.scandir(directorio) as contenido:
    for entrada in contenido:
        tipo = "Directorio" if entrada.is_dir() else "Archivo"
        print(f"{entrada.name} - {tipo}")
```

Opción 2:

```
import os

# Especificar el directorio
directorio = "./nueva_carpeta"

# Listar archivos y directorios
contenido = os.listdir(directorio)

# Mostrar el contenido
for item in contenido:
    print(item)
```

4. Flujos de datos y Sistemas de Ficheros

Para navegar por directorios podemos importar **os** o **pathlib**.

Opción 3:

```
from pathlib import Path

# Especificar el directorio
directorio = Path(".")

# Listar archivos y directorios
for entrada in directorio.iterdir():
    tipo = "Directorio" if entrada.is_dir() else
"Archivo"
    print(f"{entrada.name} - {tipo}")
```

4. Flujos de datos y Sistemas de Ficheros

Para tratar otros tipos de ficheros probablemente necesitemos librerías (si no necesariamente, al menos para facilitar el trabajo).

Por ejemplo, para escribir información en un fichero .csv:

```
import csv

# Datos de ejemplo para escribir en el CSV
datos = [
    ["Nombre", "Edad", "Ciudad"],
    ["Ana", "25", "Madrid"],
    ["Luis", "30", "Barcelona"],
    ["Carlos", "28", "Valencia"]
]

# Escritura en el archivo CSV con separador ;
with open("datos.csv", mode="w", newline="") as archivo_csv:
    escritor = csv.writer(archivo_csv,  delimiter=";")
    escritor.writerows(datos)

print("Archivo CSV creado con éxito." )
```

4. Flujos de datos y Sistemas de Ficheros

Y para leerla (de fichero .csv):

```
import csv

# Lectura del archivo CSV
with open("datos.csv", mode="r") as archivo csv:
   lector = csv.reader(archivo_csv,  delimiter=";")
   for fila in lector:
       print(fila)
```