Funciones

Las funciones nos permiten agrupar el código en bloques reutilizables. De este modo evitamos repetir innecesariamente el código y, además, podemos reutilizarlo en diferentes partes del programa.

1. Definición de funciones

A la hora de definir una función en Python, comenzamos con la palabra `def` seguida del nombre de la función y los parámetros que tendrá, entre paréntesis. Para cada parámetro sólo debemos especificar su nombre (recuerda que en Python no se especifican los tipos de datos explícitamente).

Igual que ocurre con otras estructuras como *if* o *while*, el código perteneciente a una función debe estar tabulado. Además, si queremos que la función devuelva algún valor, podemos emplear la cláusula `return` como en otros lenguajes, aunque no es obligatoria si no queremos devolver nada. También podemos definir un *return* vacío para indicar que no se devuelve nada.

Veamos algunos ejemplos.

- Esta función toma dos parámetros y devuelve el mayor de ellos

```
def maximo(num1, num2):
    if num1 > num2:
        return num1
    else:
        return num2
```

- Esta función toma un texto como parámetro y lo saca por pantalla:

```
def imprimeTexto(texto):
    print(texto)
    return # Esta línea se podría omitir
```

A la hora de llamar a estas funciones desde otras partes del código, se hace igual que en muchos otros lenguajes:

```
print ("Escribe dos números")
n1 = int(input())
n2 = int(input())
print ("El máximo es", maximo(n1, n2))

texto = input("Escribe un texto:")
imprimeTexto(texto)
```

1.1. Más sobre el valor de retorno

Hemos visto que con la instrucción `return` podemos hacer que la función devuelva un resultado. Este resultado puede ser un valor simple (por ejemplo, un número) o un dato complejo, o compuesto de varios elementos. En este último caso, podemos hacer que la función devuelva:

- Una lista de valores
- Un mapa o diccionario de datos
- Una tupla
- ...

La siguiente función devuelve una lista con los datos que recibe como parámetros:

```
def lista(n1, n2, n3):
    return [n1, n2, n3]

datos = lista(1, 2, 3)
print(datos[1]) # 2
```

Un uso curioso de esta particularidad es el trabajo con tuplas: podemos hacer que una función devuelva una secuencia de datos, separados por comas, y asignar el resultado de la llamada a un conjunto de variables, también separados por comas. Por ejemplo, la siguiente función devuelve un número y un nombre de persona. Al llamarla, podemos obtener de golpe los dos valores devueltos, y asignarlos a dos variables independientes:

```
def mi_funcion():
    return 20, "Nacho"

numero, nombre = mi_funcion()
print(numero)   # 20
print(nombre)   # Nacho
```

1.2. La instrucción *pass*

En algunas ocasiones nos puede interesar definir la cabecera de una función y no implementar (aún) su código. En este caso, para no dejar la función vacía (lo que daría un error de ejecución) podemos emplear la instrucción vacía `pass` como única instrucción de la función (que no tiene ningún efecto), y ya la completaremos más adelante:

```
def mi_funcion():
    pass
```

2. Sobre los parámetros

Veamos a continuación algunos aspectos relevantes sobre los parámetros que pasamos a las funciones.

2.1. Paso por valor y por referencia

En Python todos los parámetros simples (números, booleanos y textos) se pasan por valor, con lo que no podemos modificar el valor original del dato (se pasa una copia del mismo), y todos los tipos complejos (listas, u objetos) se pasan por referencia. Esto último implica que, siempre que se mantenga la referencia original, podemos modificar el valor del parámetro de forma persistente (se aplica a la variable original utilizada como parámetro). Por ejemplo, si empleamos esta función:

```
def anyadirValores(lista):
    lista.append(30)
    print ("Valores en la función:", lista)
    return
```

y llamamos a la función de este modo:

```
lista1 = [10, 20]
anyadirValores(lista1)
print ("Valores fuera de la función:", lista1)
```

Entonces la variable *lista1* y el parámetro *lista* almacenan los mismos valores finales: `[10, 20, 30]`. Sin embargo, si usamos esta otra función:

```
def anyadirValores(lista):
    lista = [30, 40]
    print ("Valores en la función:", lista)
    return
```

y llamamos a la función del mismo modo que antes, entonces la variable original *lista1* tendrá los valores `[10, 20]` al finalizar, y el parámetro *lista* tendrá los valores `[30, 40]` dentro de la función, pero este cambio se pierde fuera de la misma, porque se ha modificado la referencia de la variable (la hemos reasignado entera en la función), y por tanto hemos creado una nueva referencia distinta a la original, que no modifica entonces su contenido.

2.2. Tipos de parámetros

Los parámetros definidos en una función pueden ser de distintos tipos, y se pueden especificar de distintas formas. Veremos aquí algunos ejemplos.

Por un lado tenemos los parámetros **obligatorios**. Es el modo normal de pasar parámetros; si simplemente definimos el nombre de cada parámetro, entonces ese parámetro es obligatorio, y debemos darles valor al llamarles, en el mismo orden en que están definidos. Aquí podemos ver un ejemplo (el mismo visto anteriormente):

```
def maximo(num1, num2):
    if num1 > num2:
        return num1
    else:
        return num2
```

También podemos invocar a una función usando los nombres de los parámetros como **palabras clave**. De este modo no tenemos por qué seguir el mismo orden que cuando se definió dicha función. Por ejemplo:

```
def imprimirDatos(nombre, edad):
    print ("Tu nombre es", nombre, "y tu edad es", edad)
    return
...
imprimirDatos(edad = 28, nombre = "Juan")
```

Además, podemos asignar **valores por defecto** a los parámetros que queramos. Así, si queremos llamar a la función, podemos omitir los parámetros que tengan un valor por defecto asignado, si queremos. Por ejemplo:

```
def imprimirDatos(nombre, edad = 0):
    print ("Tu nombre es", nombre, "y tu edad es", edad)
    return
    ...
imprimirDatos("Juan") # Imprime "Tu nombre es Juan y tu edad es 0"
```

> **NOTA**: es importante que los parámetros que tengan valores por defecto se coloquen todos al final de la lista de parámetros (tras los obligatorios), para que así no queden huecos si queremos llamar a la

> función omitiendo parámetros. También es importante que, cuando omitamos un parámetro, los que vayan detrás también se omitan para que no se desplace el orden y se asignen por error a otros parámetros.

2.2.1. Funciones con un número variable de parámetros

La funciones en Python también admiten un **número variable de parámetros**. Esto lo podemos especificar como último parámetro de la función un tipo especial que permite pasar tantos parámetros como necesitemos. Por ejemplo:

```
def imprimirTodo(num1, *numeros):
    print("Primer número:", num1)
    for num in numeros:
        print num
    return
```

Lo que hará la función en este caso es recibir un primer parámetro obligatorio (*num1*) y el resto, opcionales, se recibirán en forma de **tupla** con sus valores. Podemos invocar a la función así:

```
imprimirTodo(1, 2, 3, 4)
# La tupla sería (2, 3, 4) en este caso
```

De forma alternativa podemos indicar un doble asterisco en ese último parámetro:

```
def imprimirTodo(num1, **valores):
    print("Primer número:", num1)
    for num in valores:
        print valores[num]
    return
```

En este otro caso lo que se recibe como parámetro adicional es un **mapa** donde a cada parámetro (valor) se le asocia un nombre (clave). Podríamos invocar a la función de este modo:

```
imprimirTodo(1, a = 2, b = 3)
```

Podemos **combinar ambas cosas** en una función que admita primero una secuencia de valores y luego una secuencia de valores con nombre asociado, por ejemplo:

```
def imprimirTodo(*numeros, **valores):
    ...
```

Y la podríamos invocar así:

```
imprimirTodo(1, 2, 3, a = 4, b = 5)
# El primer parámetro recogería la tupla (1, 2, 3)
# El segundo parámetro recogería el mapa {"a": 4, "b": 5}
```

Ejercicio 1:

Crea un programa llamado `Funciones.py` con las siguientes funciones:

1. Una función llamada `mcd` que reciba dos enteros *a* y *b* como parámetros y devuelva el máximo común divisor de esos parámetros. El máximo común divisor es el número más alto por el que se pueden dividir los dos números.
2. Una función llamada `esPrimo` que reciba un número como parámetro y devuelva un booleano indicando si el número es primo o no

Desde el programa principal, llama a la función `mcd` para calcular el máximo común divisor de 20 y 12 (debería dar 4), y usa la función `esPrimo` para sacar los números primos que haya del 1 al 50.

AYUDA: vídeo con la solución del ejercicio

2.3. Paso de parámetros al programa principal

A pesar de que en Python no existe una función principal `main` como la que sí existe en otros lenguajes como C, Java, C#... sí es posible pasar parámetros al programa desde el terminal cuando lo ejecutamos. Para ello, importamos el elemento `sys`, que hace referencia al sistema sobre el que se ejecuta el programa. Dentro, disponemos de un array predefinido llamado `argv`, similar al que existe en C o C++, con los datos que le llegan al programa. El primero de ellos, igual que ocurre en C o C++ es el nombre del propio ejecutable, y el resto son los parámetros adicionales.

```
import sys

for i in range(1, len(sys.argv)):
    # Recorremos los parámetros quitando el 0 (ejecutable)
    print(sys.argv[i])
```

Ejercicio 2:

> Crea un programa llamado `Contar.py` que reciba como parámetros del programa principal dos datos (numéricos) y realice un conteo desde el primero hasta el segundo. Si no se reciben los dos datos mostraremos un mensaje de error y finalizaremos.

3. Algunas operaciones avanzadas con listas

En este apartado veremos algunas operaciones algo más complejas que se pueden realizar con listas, y que requieren definir alguna función adicional.

3.1. Ordenación de listas

Ya hemos visto en documentos anteriores que la instrucción `sort` permite ordenar automáticamente listas simples. Pero si queremos ordenar algunos datos más complejos (como objetos, o tuplas) debemos proporcionar una función que indique el criterio de comparación. Por ejemplo, esta lista de tuplas queda ordenada ascendentemente por su edad (segundo campo):

```
def ordenarPorEdad(persona):
    return (persona[1])

gente = [("John Doe", 36, "611223344"),
         ("Mary Stewart", 54, "733445566"),
         ...]
gente.sort(key=ordenarPorEdad)
```

3.2. Mapeo de listas

La instrucción `map` aplica una función de transformación a una lista y devuelve los elementos transformados. Estos elementos pueden formar de nuevo una lista usando la instrucción `list`. Este ejemplo obtiene una lista con los cuadrados de los números de la lista original:

```
def cuadrado(x):
    return (x * x)

lista = [1, 2, 3, 4]
cuadrados = list(map(cuadrado, lista))
```

3.3. Filtrado de listas

La instrucción `filter` aplica una función de filtrado a una lista y devuelve los elementos que cumplen la condición o pasan el filtro. Como ocurre con *map*, podemos formar una nueva lista con ellos usando la instrucción `list`. El siguiente ejemplo se queda con los números pares de una colección:

```
def par(x):
    return x % 2 == 0

lista = [1, 2, 3, 4]
pares = list(filter(par, lista))
```

4. Nociones sobre programación funcional

Algunas funciones son bastante simples y ocupan una simple línea de código. Por ejemplo, echemos un vistazo a un fragmento de código anterior que transforma un array de números en sus cuadrados:

```
def cuadrado(x):
    return (x * x)

lista = [1, 2, 3, 4]
cuadrados = list(map(cuadrado, lista))
```

Cuando la función es así de simple, se puede reemplazar por una **expresión lambda**. Estas expresiones se utilizan en muchos lenguajes para implementar funciones normalmente cortas, de forma que ocupan una línea de código. Además, tienen la peculiaridad de que se pueden definir en el mismo punto en que se quieren utilizar. Empleando una expresión lambda, el código anterior quedaría así:

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

Como vemos, la notación consiste en utilizar la palabra `lambda` seguida de los parámetros que necesita la función (separados por comas), dos puntos y el resultado a devolver.

Aplicando esto mismo al ejemplo anterior que filtra los números pares usando *filter*, podría quedar un código así:

```
lista = [1, 2, 3, 4]
pares = list(filter(lambda x: x % 2 == 0, lista))
```

A la hora de ordenar colecciones de datos, también podemos definir con lambdas el criterio de ordenación. El siguiente ejemplo ordena una lista de tuplas por el segundo campo de dichas tuplas (edad numérica):

```
gente = [("John Doe", 36, "611223344"),
         ("Mary Stewart", 54, "733445566"),
         ...]
gente.sort(key=lambda x: x[1])
```

> **Ejercicio 3**:
>
> Crea un programa llamado `Loteria.py` que le pida al usuario que introduzca los 6 números que juega a la lotería (separados por espacios). Entonces, deberá crear una lista con ellos, ordenarla ascendentemente e imprimirla en pantalla. Además, el programa debe indicar si es una lista válida (es decir, los números deben estar entre 1 y 49, inclusive, sin repetirse). Por ejemplo:

```
Introduce los 6 números de la lotería separados por espacios
1 20 12 20 6 50
[1, 6, 12, 20, 20, 50]
La lista NO es válida:
Hay números repetidos
Hay números menores que 1 o mayores que 49
```

5. Gestión de errores mediante excepciones

En ocasiones, algunas operaciones que pedimos que haga un programa pueden provocar un error que no habíamos previsto. Por ejemplo, intentar convertir a entero un dato introducido por el usuario que no lo sea. Si no controlamos estos errores, el programa puede terminar de forma abrupta, mostrando un mensaje de error incomprensible para el usuario.

Para evitar esto, podemos encapsular el código que puede provocar el error en una cláusula `try` de modo que, si falla, automáticamente se saltará a una cláusula contigua llamada `except`, donde podremos mostrar un mensaje de error adecuado y seguir ejecutando el programa. Por ejemplo, el siguiente código puede fallar si lo que escribe el usuario no es un dato entero. En este caso, mostraremos un mensaje de error y seguiremos ejecutando el resto del programa:

```
try:
    numero = int(input("Escribe un número:"))
except:
    print("Error procesando el dato introducido")
print("Resto del programa desde aquí")
```

> **Ejercicio 4:**

> Adapta el ejercicio 2 anterior en un archivo llamado `ContarExcepciones.py`, y controla el caso en que los datos que escriba el usuario como parámetros no sean números enteros, mostrando el mensaje "Datos incorrectos" en ese caso.