Talento Digital para Chile:

Módulo 2 Programación Básica en Python

nodovirtual.awakelab.cl

jELOU futurejob by Adalid Chile

2.6.- Contenido 6: Codificar un programa utilizando funciones para la reutilización de código acorde al lenguaje Python

Objetivo de la jornada

1. Explica el sentido de utilizar funciones dentro de un programa distinguiendo su definición versus su invocación
2. Define funciones que utilizan parámetros de entrada y que producen un retorno para resolver un problema
3. Explica el alcance de una variable dentro y fuera de una función distinguiendo el concepto de variable local y global
4. Utiliza funciones preconstruidas y personalizadas por el usuario con paso de parámetros y que obtienen un retorno

2.6.1.- Funciones en Python

2.6.1.1. Qué es una función

Usted puede estar familiarizado con el concepto matemático de una función. Una función es una relación o mapeo entre una o más entradas y un conjunto de salidas. En matemáticas, una función generalmente se representa así:

```
z = f(x, y)
```

Aquí, **f** es una función que opera en las entradas **x** e **y**. La salida de la función es **z**. Sin embargo, las funciones de programación son mucho más generalizadas y versátiles que esta definición matemática. De hecho, la definición y el uso adecuados de las funciones son tan críticos para el desarrollo adecuado del software que prácticamente todos los lenguajes de programación modernos admiten funciones integradas y definidas por el usuario.

En Python, una función es un grupo de declaraciones relacionadas que realiza una tarea específica.

Prácticamente todos los lenguajes de programación utilizados en la actualidad admiten una forma de funciones definidas por el usuario, aunque no siempre se denominan funciones. En otros lenguajes, se pueden referidos como:

- Subrutinas
- Procedimientos
- Métodos (esto se revisará más adelante en el capítulo de orientación a objetos)
- Subprogramas

2.6.1.2. Para qué sirve una función

Las funciones ayudan a dividir nuestro programa en fragmentos más pequeños y modulares. A medida que nuestro programa se hace más y más grande, las funciones lo hacen más organizado y manejable. Además, evita la repetición y hace que el código sea reutilizable.

En otras palabras, se utilizan funciones en la programación para agrupar un conjunto de instrucciones que desea usar repetidamente o que, debido a su complejidad, están mejor autocontenidas en un subprograma y se llaman cuando es necesario. Eso significa que una función es un código escrito para llevar a cabo una tarea específica. Para llevar a cabo esa tarea específica, la función podría o no necesitar múltiples entradas. Cuando se lleva a cabo la tarea, la función puede o no devolver uno o más valores.

2.6.1.3. Sintaxis de funciones en Python

Sintaxis de Función

```
def nombre_de_funcion(parametros):
        """docstring"""
        sentencia(s)(parametros)
```

Detallamos los contenidos de esta sintaxis como parte de la siguiente sección.

2.6.1.4. Definir funciones

En la sección anterior se muestra una definición de función que consta de los siguientes componentes.

- Palabra clave **def** que marca el inicio del encabezado de la función.
- Un nombre de función para identificar de forma exclusiva la función. La denominación de funciones sigue las mismas reglas de escritura de identificadores en Python.

- Parámetros (argumentos) a través de los cuales pasamos valores a una función. Son opcionales.
- Dos puntos (:) para marcar el final del encabezado de la función.
- String de documentación opcional (**docstring**) para describir lo que hace la función.
- Una o más sentencias de Python válidas que componen el cuerpo de la función. Las declaraciones deben tener el mismo nivel de sangría (generalmente 4 espacios).
- Una declaración de devolución (**return**) opcional para devolver un valor de la función.

Ejemplo de una función:

```
def saludo(nombre):
    """
    Esta función saluda a
    la persona pasada como parámetro
    """
    print("Hola, " + nombre + ". Buenos días")
```

Hay tres tipos de funciones en Python:

- Funciones nativas, como help() para buscar ayuda, **min()** para obtener el valor mínimo, **print()** para imprimir un objeto en el terminal, etc. Puede encontrar una descripción general con más de estas funciones en esta pagina https://docs.python.org/3/library/functions.html.
- Funciones definidas por el usuario, que son funciones que los usuarios crean para ayudarlos; y
- Funciones anónimas, que también se denominan funciones lambda porque no se declaran con la palabra clave **def** estándar.

2.6.1.5. Recepción de parámetros en una función/invocación de una función

Aquí hay una función simple llamada **greet_user()** que imprime un saludo:

```
def saludar_a_usuario():
        """Entrega saludo simple a usuario."""
        print("Hola!")
Saludar_a_usuario()
```

En Python, puede definir una función que tome un número variable de argumentos. Un argumento es una información que se pasa de una llamada a la función misma. Aquí se muestra una función simple:

```
def funcion_cualquiera(val1, val2, val3):
     return val1 + val2 + val3
```

Cuando se usa, obtenemos lo siguiente:

```
>>> print(funcion_cualquiera(1, 2, 3))
6
```

En este caso, los argumentos 1, 2 y 3 se pasaron a la función **cualquiera()**, y los valores se almacenaron en los parámetros **val1**, **val2** y **val3** respectivamente.

A veces se habla de argumentos y parámetros indistintamente. No se sorprenda si ve las variables en una definición de función denominada argumentos o variables en una llamada a la función referida como parámetros.

Orden de los parámetros

La función de la sección anterior tiene 3 parámetros posicionales (cada uno obtiene el siguiente valor pasado a la función cuando se llama: val1 obtiene el primer valor (1), val2 obtiene el segundo valor (2), etc.).

Debido a que una definición de función puede tener múltiples parámetros, una llamada de función puede necesitar múltiples argumentos. Puede pasar argumentos a sus funciones de varias maneras. Puede usar:

- argumentos posicionales, que deben estar en el mismo orden en que se escribieron los parámetros.
- argumentos de palabras clave, donde cada argumento consiste en un nombre de variable y un valor; y
- listas y diccionarios de valores (las que veremos en este documento en apartados específicos)

Argumentos posicionales

Cuando llama a una función, Python debe hacer coincidir cada argumento en la llamada de función con un parámetro en la definición de la función. La forma más

sencilla de hacerlo se basa en el orden de los argumentos proporcionados. Los valores emparejados de esta manera se denominan argumentos posicionales. Para ver cómo funciona esto, considere una función que muestre información sobre mascotas. La función nos dice qué tipo de animal es cada mascota y el nombre de la mascota, como se muestra aquí:

```
>>> def describir_mascota(tipo_animal, nombre_mascota):
...     """Muestra información sobre mascota."""
...     print("\nYo tengo un " + tipo_animal + ".")
...     print("Mi " + tipo_animal
...             + " se llama "
...             + nombre_mascota.title()
...             + ".")
...
>>> describir_mascota('hamster', 'harry')
```

La definición muestra que esta función necesita un **tipo_animal** y el **nombre_mascota**. Cuando llamamos a **describir_mascota()**, necesitamos proporcionar un tipo de animal y un nombre, en ese orden. Por ejemplo, en la llamada a la función, el argumento 'hamster' se almacena en el parámetro **tipo_animal** y el argumento 'harry' se almacena en el parámetro **nombre_mascota**. En el cuerpo de la función, estos dos parámetros se usan para mostrar información sobre la mascota descrita.
La salida describe un hámster llamado Harry:

```
Yo tengo un hamster.
Mi hamster se llama Harry.
```

Múltiples llamadas a funciones
Puede llamar a una función tantas veces como sea necesario. Describir una segunda mascota diferente requiere solo una llamada más a **describir_mascota()**:

```
>>> def describir_mascota(tipo_animal, nombre_mascota):
...     """Muestra información sobre mascota."""
...     print("\nYo tengo un " + tipo_animal + ".")
...     print("Mi " + tipo_animal
...             + " se llama "
...             + nombre_mascota.title()
...             + ".")
...
>>> describir_mascota('hamster', 'Harry')
>>> describir_mascota('perro', 'Willie')
```

En esta segunda llamada a la función, pasamos a **describir_mascota()** los argumentos **'perro'** y **'willie'**. Al igual que con el conjunto anterior de argumentos

que utilizamos, Python hace coincidir **'perro'** con el parámetro **tipo_animal** y **'willie'** con el parámetro **nombre_mascota**.

Como antes, la función hace su trabajo, pero esta vez imprime valores para un perro llamado Willie. Ahora tenemos un hámster llamado Harry y un perro llamado Willie:

```
Yo tengo un hamster.
Mi hamster se llama Harry.
Yo tengo un perro.
Mi perro se llama Willie.
```

Llamar a una función varias veces es una forma muy eficiente de trabajar. El código que describe una mascota se escribe una vez en la función. Luego, cada vez que desee describir una nueva mascota, llamamos a la función con la información de la nueva mascota. Incluso si el código para describir una mascota se expandiera a diez líneas, aún podría describir una nueva mascota en una sola línea llamando nuevamente a la función. Puede usar tantos argumentos posicionales como necesite en sus funciones. Python trabaja a través de los argumentos que proporciona al llamar a la función y hace coincidir cada uno con el parámetro correspondiente en la definición de la función.

El orden importa en los argumentos posicionales

Puede obtener resultados inesperados si mezcla el orden de los argumentos en una llamada a la función cuando se utilizan argumentos posicionales:

```
>>> def describir_mascota(tipo_animal, nombre_mascota):
...     """Muestra información sobre mascota."""
...     print("\nYo tengo un " + tipo_animal + ".")
...     print("Mi " + tipo_animal
...             + " se llama "
...             + nombre_mascota.title()
...             + ".")
...
>>> describir_mascota('Harry', 'hamster')
```

En esta llamada a la función, ubicamos el nombre primero y el tipo de animal segundo. Debido a que el argumento **'Harry'** aparece primero esta vez, ese valor se almacena en el parámetro **tipo_animal**. Del mismo modo, **'hamster'** se almacena en **nombre_mascota**. Ahora nosotros tenemos un **'Harry'** llamado **'hamster'**:

```
Yo tengo un Harry.
Mi hamster se llama hamster.
```

Si obtiene resultados divertidos como este, verifique que el orden de los argumentos en su llamada a la función coincide con el orden de los parámetros en la definición de la ésta.

Argumentos de palabras clave

Un argumento de palabra clave es un par clave-valor que pasa a una función. Usted asocia directamente la clave y el valor dentro del argumento, de modo que cuando pase el argumento a la función, no hay confusión (no terminará con un Harry llamado hamster). Los argumentos de palabras clave liberan el hecho de tener que preocuparnos por ordenar correctamente los argumentos en la llamada a la función, y aclaran el papel de cada valor en la llamada a la ésta.

Reescribamos nuestra función usando argumentos de palabras clave:

```
>>> def describir_mascota(tipo_animal, nombre_mascota):
...     """Muestra información sobre mascota."""
...     print("\nYo tengo un " + tipo_animal + ".")
...     print("Mi " + tipo_animal
...             + " se llama "
...             + nombre_mascota.title()
...             + ".")
...
>>> describir_mascota(tipo_animal='hamster', nombre_mascota='Harry')
```

La función **describir_mascota()** no ha cambiado. Pero cuando llamamos a la función, le decimos explícitamente a Python con qué parámetro debe coincidir cada argumento. Cuando Python lee la llamada a la función, sabe almacenar el argumento **'hamster'** en el parámetro **tipo_animal** y el argumento **'Harry'** en **nombre_mascota**. La salida muestra correctamente que tenemos un hamster llamado Harry.

El orden de los argumentos de las palabras clave no importa porque Python sabe a qué corresponde cada valor. Las siguientes dos llamadas a funciones son equivalentes:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

Valores predeterminados

Al escribir una función, puede definir un valor predeterminado para cada parámetro. Si se proporciona un argumento para un parámetro en la llamada a la función, Python usa el valor del argumento. De lo contrario, utiliza el valor predeterminado del parámetro. Así cuando define un valor predeterminado para un parámetro, puede excluir el argumento correspondiente que normalmente escribiría en la llamada a la función. Usar valores predeterminados puede simplificar sus llamadas a funciones y aclarar las formas en que sus funciones se usan típicamente.

Por ejemplo, si observa que la mayoría de las llamadas a **describir_mascota()** son para describir perros, puede establecer el valor predeterminado de **tipo_animal** en **'perro'**. Ahora cualquiera que llame a **describir_mascota()** para un perro puede omitir esa información:

```
>>> def describir_mascota(nombre_mascota, tipo_animal='perro'):
...     """Muestra información sobre mascota."""
...     print("\nYo tengo un " + tipo_animal + ".")
...     print("Mi " + tipo_animal
...             + " se llama "
...             + nombre_mascota.title()
...             + ".")
...
>>> describir_mascota(nombre_mascota='Willie')
```

Cambiamos la definición de **describir_mascota()** para incluir un valor predeterminado, **'perro'**, para **tipo_animal**. Ahora cuando se llama a la función sin **tipo_animal** especificado, Python sabe usar el valor **'perro'** para este parámetro:

```
Yo tengo un perro.
Mi hamster se llama Willie.
```

Tenga en cuenta que el orden de los parámetros en la definición de la función tuvo que ser cambiado. Esto debido a que el valor predeterminado hace innecesario especificar un tipo de animal como argumento, el único argumento que queda en la llamada a la función es el nombre de la mascota. Python todavía interpreta esto como un argumento posicional, así es que si la función se llama sólo con el nombre de una mascota, ese argumento coincidirá con el primer parámetro listado en la definición de la función. Esta es la razón por la que el primer parámetro debe ser **nombre_mascota**.

La forma más sencilla de usar esta función ahora es proporcionar solo el nombre del perro en la llamada a la función:

```
describir_mascota('Cachupín')
```

Esta llamada a función tendría el mismo resultado que el ejemplo anterior. El único argumento proporcionado es 'Cachupín', por lo que coincide con el primer parámetro en la definición, **nombre_masciota**. Esto pues no se proporciona ningún argumento para **tipo_animal**, Python usa el valor predeterminado **'perro'**.

Para describir un animal que no sea un perro, puede usar una llamada de función como esta:

```
describir_mascota(nombre_mascota='Harry', tipo_animal='hamster')
```

Debido a que se proporciona un argumento explícito para **tipo_animal**, Python ignorará el valor predeterminado del parámetro.

Cuando utiliza valores predeterminados, debe enumerarse cualquier parámetro con un valor predeterminado después de todos los parámetros que no tienen valores predeterminados. Esto permite que Python continúe interpretando argumentos posicionales correctamente.

Listas como parámetros

A menudo le resultará útil pasar una lista a una función, ya sea una lista de nombres, números u objetos más complejos, como diccionarios. Cuando usted pasa una lista a una función, la función obtiene acceso directo a los contenidos de la lista.

Digamos que tenemos una lista de usuarios y queremos imprimir un saludo para cada uno. El siguiente ejemplo envía una lista de nombres a una función llamada **saludar_usuarios()**, que saluda a cada persona en la lista individualmente:

```
def saludar_usuarios(nombres):
    """Imprime saludo simple a cada usuario de lista nombres."""
    for nombre in nombres:
        msj = "¡Hola, " + nombre.title() + "!"
        print(msj)

usernames = ['kenita', 'cecilia', 'marisela']
saludar_usuarios(usernames)
```

Definimos **saludar_usuarios()** para que espere una lista de nombres, que almacena en los nombres de los parámetros. La función recorre la lista que recibe y imprime un saludo a cada usuario. En la línea que declara **usernames** definimos una lista de usuarios y luego pasamos la lista de nombres de usuario a sa**ludar_usuarios()** en nuestra llamada de función, obteniendo como salida:

```
¡Hola, Kenita!
¡Hola, Cecilia!
¡Hola, Marisela!
```

Este es el resultado que queríamos. Cada usuario ve un saludo personalizado, y podemos llamar a la función en cualquier momento que desee saludar a un conjunto específico de usuarios.

Modificar una lista en una función

Cuando pasa una lista a una función, la función puede modificar la lista. Los cambios realizados en la lista dentro del cuerpo de la función son permanentes, lo que permite trabajar eficientemente incluso cuando se trata de grandes cantidades de datos.

Considere una empresa que crea modelos impresos en 3D de diseños que los usuarios envían. Los diseños que deben imprimirse se almacenan en una lista y luego cuando se imprimen se mueven a una lista separada. El siguiente código hace esto sin usar funciones:

```
# Comenzar con algunos diseños que deben imprimirse
sin_imprimir = ['iphone', 'robot', 'dodecaedro']
impresos= []

# Simular impresión de cada diseño, hasta terminarlos
# Moder cada diseño a impresos luego de imprimir
while sin_imprimir:
    actual = sin_imprimir.pop()
    # Simular creacion de impresión 3D desde diseño
    print("Imprimiendo modelo: " + actual)
    impresos.append(actual)

# Mostrar todos los modelos ya impresos
print("\nLos siguientes modelos han sido impresos:")
for impreso in impresos:
    print(impreso)
```

Este programa comienza con una lista de diseños que deben imprimirse y una lista vacía llamada **impresos** a la que se moverá cada diseño después de que ha sido impreso. Mientras los diseños permanezcan en diseños no impresos, el loop **while** simula la impresión de cada diseño eliminando un diseño del final de la lista, almacenándola en **actual** y mostrando un mensaje de que se está imprimiendo el diseño actual. Luego agrega el diseño a la lista de modelos completados. Cuando el

ciclo termina de ejecutarse, una lista de los diseños que han sido impresos se muestra:

```
Imprimiendo modelo: dodecaedro
Imprimiendo modelo: robot
Imprimiendo modelo: iphone
Los siguientes modelos han sido impresos:
dodecaedro
robot
iphone
```

Podemos reorganizar este código escribiendo dos funciones, cada una de las cuales hace un trabajo específico. La mayor parte del código no cambiará; solo lo estamos haciendo más eficiente. La primera función se encargará de imprimir los diseños, y el segundo resumirá las impresiones que se han realizado:

```
def imprimir_modelos(sin_imprimir, impresos):
    """
    Simular la impresión de cada diseño, hasta que no queden.
    Mover cada diseño a impresos luego de imprimir.
    """
    while sin_imprimir:
        actual = sin_imprimir.pop()
        # Simular la creación de impresión 3D desde diseño
        print("Imprimiendo modelo: " + actual)
        impresos.append(actual)

def mostrar_modelos_impresos(impresos):
    """Mostrar todos los modelos que se imprimieron"""
    print("\nLos siguientes modelos han sido impresos:")
    for impreso in impresos:
        print(impreso)

sin_imprimir = ['iphone', 'robot', 'dodecaedro']
impresos= []
imprimir_modelos(sin_imprimir, impresos)
mostrar_modelos_impresos(impresos)
```

En la primera parte definimos la función **imprimir_modelos()** con dos parámetros: una lista de diseños que deben imprimirse y una lista de modelos completos. Dadas estas dos listas, la función simula la impresión de cada diseño al vaciar la lista de diseños sin imprimir y llenando la lista de modelos completos.

En la segunda parte definimos la función **mostrar_modelos_impresos()** con un parámetro: la lista de modelos terminados. Dada esta lista, **mostrar_modelos_impresos()** muestra el nombre de cada modelo que fue impreso.

Este programa tiene el mismo resultado que la versión sin funciones, pero el código está mucho más organizado. El código que hace la mayor parte del trabajo ha sido trasladado a dos funciones separadas, lo que hace que la parte principal del

programa sea más fácil de entender. Mire el cuerpo del programa para ver cómo es mucho más fácil entender qué está haciendo este programa:

```
sin_imprimir = ['iphone', 'robot', 'dodecaedro']
impresos= []
imprimir_modelos(sin_imprimir, impresos)
mostrar_modelos_impresos(impresos)
```

Configuramos una lista de diseños no impresos y una lista vacía que contendrá modelos terminados. Entonces, como ya hemos definido nuestras dos funciones, todo lo que tenemos que hacer es llamarlos y pasarles los argumentos correctos. Nosotros llamamos a **imprimir_modelos()** y le pasamos las dos listas que necesita; como se esperaba, **imprimir_modelos()** simula la impresión de los diseños. Luego llamamos **mostrar_modelos_impresos()** y pasamos la lista de modelos completos para que pueda informar los modelos que han sido impresos. Los nombres descriptivos de las funciones permiten a otros leer este código y comprenderlo, incluso sin comentarios.

Este programa es más fácil de extender y mantener que la versión sin funciones. Si necesitamos imprimir más diseños más adelante, simplemente podemos llamar **imprimir_modelos()** nuevamente. Si nos damos cuenta de que el código de impresión necesita ser modificado, podemos cambiar el código una vez, y nuestros cambios tendrán lugar en todas partes donde se llama a la función. Esta técnica es más eficiente que tener que actualizar código por separado en varios lugares del programa.

Este ejemplo también demuestra la idea de que cada función debería tener un trabajo específico. La primera función imprime cada diseño, y la segunda muestra los modelos completos. Esto es más beneficioso que usar una función para hacer ambos trabajos. Si está escribiendo una función y nota que la función está haciendo demasiadas tareas diferentes, intente dividir el código en dos funciones. Recuerde que siempre puede llamar a una función desde otra función, que puede ser útil al dividir una tarea compleja en una serie de pasos.

Evitar que una función modifique una lista

A veces usted querrá evitar que una función modifique una lista. Por ejemplo, digamos que comienza con una lista de diseños sin imprimir y escribe una función para moverlos a una lista de modelos completados, como en el anterior ejemplo. Puede decidir que, aunque haya impreso todos los diseños, desea mantener la lista original de diseños sin imprimir para sus registros. Pero porque movió todos los nombres de diseño de **sin_imprimir**, la lista ahora está vacía, y la lista vacía es la

única versión que tiene; el original se ha ido. En este caso, puede solucionar este problema pasando una copia a la función de la lista, no el original. Cualquier cambio que realice la función en la lista afecta solo a la copia, dejando la lista original intacta. Puede enviar una copia de una lista a una función como esta:

```
function_name(list_name[:])
```

La notación **[:]** hace una copia de la lista para enviar a la función. Si no quisiéramos vaciar la lista de diseños sin imprimir podríamos llamar a **imprimir_modelos()** así:

```
imprimir_modelos(sin_imprimir[:], impresos)
```

La función **imprimir_modelos()** puede hacer su trabajo porque aún recibe los nombres de todos los diseños sin imprimir. Pero esta vez usa una copia de la lista de diseños originales sin imprimir, no la lista real de diseños sin imprimir. La lista **impresos** se llenará con los nombres de los modelos impresos como lo hizo antes, pero la lista original de diseños no impresos no se verá afectada por la función.

Aunque puede preservar el contenido de una lista pasando una copia a sus funciones, debe pasar la lista original a funciones a menos que tenga una razón específica para pasar una copia. Es más eficiente para una función trabajar con una lista existente para evitar usar el tiempo y la memoria necesarios para que haga una copia por separado, especialmente cuando trabaje con listas grandes.

Diccionarios como parámetros

Como hemos visto un diccionario en Python es una colección de datos en formato key value que no tiene orden y es mutable. A diferencia de los índices numéricos utilizados por las listas, un diccionario usa la clave como índice para un valor específico. Las claves mismas se emplean para usar un valor específico.

En Python, todo es un objeto, por lo que el diccionario se puede pasar como argumento a una función como se pasan otras variables.

```
def func(d):
    for clave in d:
        print("Clave:", clave, "Valor:", d[clave])

# Código de conductor
D = {'a':1, 'b':2, 'c':3}
func(D)
```

La salida de este código es

```
Clave: a Valor: 1
Clave: b Valor: 2
Clave: c Valor: 3
```

Pasando diccionario como kwargs

kwargs significa argumentos de palabras clave. Se usa para pasar objetos de datos avanzados como diccionarios a una función porque en tales funciones no se tiene idea de la cantidad de argumentos, por lo tanto, los datos pasados se tratan correctamente agregando "**" al argumento.

Observemos el siguiente ejemplo:

```
def mostrar(**nombre):
    print(nombre["nombre1"]
    +" "+nombre["nombre2"]
    +" "+nombre["apellido"]
    )

def main():
    # pasando un diccionario de pares clave valor
    # como argumento
    mostrar(nombre1 ="John",
            nombre2 ="F.",
            apellido ="Kennedy")
# Código de conductor
main()
```

Cuyo resultado se imprime como:

```
John F. Kennedy
```

Veámoslo con otro ejemplo, para dejarlo más claro:

```
def mostrar(x = 0, y = 0, **nombre):
    print(nombre["nombre1"]
            +" "+nombre["nombre2"]
            +" "+nombre["apellido"])
    print("x =", x)
    print("y =", y)

def main():
    mostrar(2, nombre1 ="John", nombre2 ="F.",
            apellido ="Kennedy")

main()
```

Esta vez se imprime como resultado:

```
John F. Kennedy
x = 2
```

```
y = 0
```

2.6.1.6. Retorno de una función

Una función no siempre tiene que mostrar su salida directamente. En cambio, puede procesar algunos datos y luego devolver un valor o conjunto de valores. El valor que la función retorna se llama valor de retorno. La declaración de retorno toma un valor desde el interior de una función y la envía de vuelta a la línea que llamó a la función. Los valores de retorno le permiten mover gran parte del trabajo duro de su programa a funciones, que pueden simplificar el cuerpo principal de éste.

Retornando un valor simple

Veamos una función que toma un nombre y apellido, y devuelve un nombre completo formateado:

```
def formatear_nombre(nombre, apellido):
    """Retornar un nombre completo formateado"""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = formatear_nombre('jimi', 'hendrix')
print(musico)
```

La definición de **formatear_nombre()** toma como parámetros el nombre y apellidos de una persona. La función combina estos dos nombres, agrega un espacio entre ellos, y almacena el resultado en **nombre_completo**. El valor de **nombre_completo** se convierte a mayúscula y luego se retorna a la línea en que se llamó a la función. Cuando llama a una función que devuelve un valor, debe proporcionar una variable donde se puede almacenar el valor de retorno. En este caso, el valor devuelto se almacena en la variable **musico**. El resultado muestra un nombre bien formateado compuesto por las partes del nombre de una persona:

```
Jimi Hendrix
```

Esto puede parecer mucho trabajo para obtener un nombre bien formateado cuando podríamos haber escrito:

```
print("Jimi Hendrix")
```

Pero cuando considera trabajar con un programa grande que necesita almacenar muchos nombres y apellidos por separado, funciones como **formatear_nombre()** se vuelven muy útiles. Usted puede almacenar los nombres y apellidos por separado y luego llama esta función cada vez que desee mostrar un nombre completo.

Hacer un argumento opcional

A veces tiene sentido hacer un argumento opcional para que al utilizar la función se pueda elegir el proporcionar información sólo si es deseado. Puede usar valores predeterminados para hacer que un argumento sea opcional.

Por ejemplo, supongamos que queremos expandir **formatear_nombre()** para manejar un segundo nombre también. Un primer intento de incluir los segundos nombres podría verse como esto:

```
def formatear_nombre(nombre, nombre2, apellido):
    """Retornar un nombre completo formateado"""
    nombre_completo = nombre + ' ' + nombre2 + ' ' + apellido
    return nombre_completo.title()

musico = formatear_nombre('jimi', 'felix', 'hendrix')
print(musico)
```

Esta función trabaja cuando se le da un nombre, segundo nombre y apellido. La función toma las tres partes de un nombre y luego construye un string con ellos. La función agrega espacios donde es apropiado y convierte el total del nombre a primera letra mayúscula:

```
Jimi Felix Hendrix
```

Pero los segundos nombres no siempre son necesarios, y esta función como está escrita no funcionaría si se intenta llamar solo con un nombre y un apellido. Para hacer que el segundo nombre sea opcional, podemos dar al argumento **nombre2** un valor predeterminado vacío e ignorar el argumento a menos que el usuario proporcione un valor. Para hacer que **formatear_nombre()** funcione sin un segundo nombre, establecemos el valor predeterminado de **nombre2** como una cadena vacía y lo movemos al final de la lista de parámetros:

```
def formatear_nombre(nombre, apellido, nombre2=''):
    """Retornar un nombre completo formateado"""
    if nombre2:
        nombre_completo = nombre + ' ' + nombre2 + ' ' + apellido
else:
```

```
        nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

musico = formatear_nombre('jimi', 'hendrix')
print(musico)

musico = formatear_nombre('jimi', 'felix', 'hendrix')
print(musico)
```

En este ejemplo, el nombre se construye a partir de tres partes posibles. Porque siempre hay un nombre y apellido, estos parámetros se enumeran primero en la definición de la función. El segundo nombre es opcional, por lo que aparece en último lugar en la definición, y su valor predeterminado es una cadena vacía.

En el cuerpo de la función, verificamos si un segundo nombre ha sido dado. Python interpreta cadenas no vacías como **True**, por lo que si **nombre2** se evalúa como **True** sí hay un argumento de segundo nombre en la llamada a la función. Si se proporciona un segundo nombre, el primer nombre, el segundo nombre y el apellido se combinan para formar un nombre completo. Este nombre se cambia a primera letra con mayúscula y se devuelve a la línea de llamada de la función donde se almacena en la variable **musico** y se imprime. Si no se proporciona un segundo nombre, el string vacío provoca la ejecución del bloque **else**. El nombre completo se construye solo con un nombre y apellido, y el nombre con formato se devuelve a la línea de llamada donde se almacena en **musico** y se imprime.

Llamar a esta función con un nombre y apellido es sencillo. Sin embargo, si estamos usando un segundo nombre, debemos asegurarnos de que el segundo nombre sea el último argumento aprobado para que Python haga coincidir los argumentos posicionales correctamente.

Esta versión modificada de nuestra función sirve para personas con solo un nombre y apellido, y también funciona para personas que tienen un segundo nombre:

```
Jimi Hendrix
John Lee Hooker
```

Los valores opcionales permiten que las funciones manejen una amplia gama de casos al tiempo que permiten que las llamadas a funciones sigan siendo lo más simples posible.

Retornar un diccionario

Una función puede devolver cualquier tipo de valor que necesite, incluidas estructuras de datos más complicadas como listas y diccionarios. Por ejemplo, la

siguiente función toma partes de un nombre y devuelve un diccionario que representa a una persona:

```
def construir_persona(nombre, apellido):
        """Retorna un diccionario con informacion de una persona."""
        persona = {'nombre': nombre, 'apellido': apellido}
        return persona

musico = construir_persona('jimi', 'hendrix')
print(musico)
```

La función **construir_persona()** toma un nombre y un apellido, y agrupa estos valores en un diccionario. El valor de **nombre** se almacena con la clave **'nombre'**, y el valor de **apellido** se almacena con la clave **'apellido'**. El diccionario completo representa a la persona que se regresa. El valor de retorno se imprime con las dos piezas originales de información textual ahora almacenadas en un diccionario:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Esta función toma información textual simple y la coloca en una estructura de datos más significativa que le permite trabajar con la información más allá de simplemente imprimirla. Las cadenas 'jimi' y 'hendrix' ahora están etiquetadas como nombre y apellido. Puede ampliar fácilmente esta función para aceptar valores opcionales como un segundo nombre, una edad, una ocupación o cualquier otra información que desee almacenar sobre una persona. Por ejemplo, el siguiente cambio también le permite almacenar la edad de una persona:

```
def construir_persona(nombre, apellido, edad=''):
    """Retorna un diccionario con informacion de una persona."""
    persona = {'nombre': nombre, 'apellido': apellido}
    if edad:
        persona['edad'] = edad
    return persona

musico = construir_persona('jimi', 'hendrix', edad=27)
print(musico)
```

Agregamos edad como parámetro opcional a la definición de la función y asignamos al parámetro un valor predeterminado vacío. Si la llamada a la función incluye un valor para este parámetro, el valor se almacena en el diccionario. Esta función siempre almacena el nombre de una persona, pero también se puede modificar para almacenar cualquier otra información que desee sobre ella.

Usar una función con un loop while

Podemos usar funciones con todas las estructuras de Python que hemos aprendido hasta ahora. Por ejemplo, usemos la función **formatear_nombre()** con un loop while para saludar a los usuarios de manera más formal. Aquí hay un primer intento de saludar a las personas con sus nombres y apellidos:

```
def formatear_nombre(nombre, apellido):
    """Retorna nombre completo bien formateado."""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

while True:
    print("\nPor favor dime tu nombre:")
    nombre = input("Nombre: ")
    apellido = input("Apellido: ")
    nombre_formateado = formatear_nombre(nombre, apellido)
    print("\nHola, " + nombre_formateado + "!")
```

Para este ejemplo, usamos una versión simple de **formatear_nombre()** que no involucra segundo nombre. El loop while le pide al usuario que ingrese su nombre, y solicitamos su nombre y apellido por separado. Pero hay un problema con este ciclo **while**: no hemos definido una condición de salida. ¿Dónde emplazamos una condición para salir cuando solicita una serie de entradas? Queremos que el usuario pueda salir tan fácilmente como sea posible, por lo que cada prompt debería ofrecer una forma de salir. Al igual que en el estudio de loops, usaremos la declaración **break** que ofrece la forma de salir del loop en cualquier solicitud de información por parte del usuario:

```
def formatear_nombre(nombre, apellido):
    """Retorna nombre completo bien formateado."""
    nombre_completo = nombre + ' ' + apellido
    return nombre_completo.title()

while True:
    print("\nPor favor dime tu nombre:")
    print("(Ingrese 'q' para salir)")
    nombre = input("Nombre: ")
    if nombre == 'q':
        break
    apellido = input("Apellido: ")
    if apellido == 'q':
        break
    nombre_formateado = formatear_nombre(nombre, apellido)
    print("\nHola, " + nombre_formateado + "!")
```

Agregamos un mensaje que informa al usuario cómo salir, y luego salimos del loop si el usuario ingresa el valor para dejar de ejecutar el programa en cualquiera de las solicitudes. Ahora el programa continuará saludando a las personas hasta que alguien ingrese **'q':**

```
Por favor dime tu nombre:
(Ingrese 'q' para salir)
Nombre: Alan
Apellido: Turing

Hola, Alan Turing!

Por favor dime tu nombre:
(Ingrese 'q' para salir)
Nombre: q
$
```

2.6.1.7. Variables locales y variables globales

La forma en que Python usa variables globales y locales es especial. Mientras que en muchos o en la mayoría de los otros lenguajes de programación, las variables se tratan como globales si no se declara lo contrario, Python trata con las variables al revés. Son locales, si no se declaran de otra manera. La razón principal detrás de este enfoque es que las variables globales son generalmente una mala práctica y deben evitarse. En la mayoría de los casos en los que se sienta tentado a utilizar una variable global, es mejor utilizar un parámetro para obtener un valor en una función o devolver un valor para sacarlo. Como en muchas otras estructuras de programa, Python también impone un buen hábito de programación por diseño.

2.6.1.8. Alcance de las variables locales

Cuando define variables dentro de una definición de función, son locales a esta función por defecto. Es decir, cualquier cosa que haga a dicha variable en el cuerpo de la función no tendrá efecto en otras variables fuera de la función, incluso si tienen el mismo nombre. En otras palabras, el cuerpo de la función es el alcance de dicha variable, es decir, el contexto de cierre donde este nombre está asociado con sus valores.

Todas las variables tienen el alcance del bloque, donde se declaran y definen. Solo se pueden utilizar después del momento de su declaración.

Solo a modo de recapitulación de apartados anteriores, las variables no tienen que ser y no pueden declararse de la forma en que se declaran en lenguajes de programación como Java o C. Las variables en Python se declaran implícitamente definiéndolas, es decir, la primera vez que asigna un valor a una variable, esta

variable se declara y tiene automáticamente el tipo de datos del objeto que se le debe asignar.

Variables globales y locales en funciones

En el siguiente ejemplo, queremos demostrar cómo se pueden usar los variables globales dentro del cuerpo de una función:

```
def f():
    print(s)

s = "¡Amo París en el verano!"
f()
```

Este código imprime:

```
¡Amo París en el verano!
```

La variable **s** se define como el string "¡Amo París en el verano!", Antes de llamar a la función **f()**. El cuerpo de **f()** consta únicamente de la declaración **print(s)**. Como no hay una variable local **s**, es decir, ninguna asignación a **s**, se utiliza el valor de la variable global **s**. Entonces la salida será el string "¡Amo París en el verano!". La pregunta es, ¿qué pasará si cambiamos el valor de **s** dentro de la función **f()**? ¿Afectará también a la variable global? Probamos esto en el siguiente código:

```
def f():
    s = "¡Amo Londres!"
    print(s)

s = "¡Amo París!"
f()
print(s)
```

El resultado al ejecutar este código es:

```
¡Amo Londres!
¡Amo París!
```

¿Qué pasa si combinamos el primer ejemplo con el segundo, es decir, primero accedemos a **s** con una función **print()**, con la esperanza de obtener el valor global y luego le asignamos un nuevo valor? Asignarle un valor significa, como hemos dicho anteriormente, crear una variable local **s**. Entonces, tendríamos **s** como una variable global y local en el mismo ámbito, es decir, el cuerpo de la función. Python afortunadamente no permite esta ambigüedad. Entonces, generará un error, como podemos ver en el siguiente ejemplo:

```
def f():
```

```
    print(s)
    s = "¡Amo Londres!"
    print(s)

s = "¡Amo París!"
f()
```

El resultado al ejecutar este fragmento de código resulta en:

```
Traceback (most recent call last):
  File "funcion_local_global.py", line 7, in <module>
    f()
  File "funcion_local_global.py", line 2, in f
    print(s)
UnboundLocalError: local variable 's' referenced before assignment
```

Una variable no puede ser tanto local como global dentro de una función. Entonces, Python decide que queremos una variable local debido a la asignación de **s** dentro de **f()**, por lo que la primera instrucción de impresión antes de la definición de **s** arroja el mensaje de error anterior. Cualquier variable que se cambie o se cree dentro de una función es local, si no se ha declarado como una variable global. Para decirle a Python que queremos usar la variable global, tenemos que declararlo explícitamente usando la palabra clave "global", como se puede ver en el siguiente ejemplo:

```
def f():
    global s
    print(s)
    s = "¡Pero Londres es genial también!"
    print(s)

s = "¡Busco un curso en París!"
f()
print(s)
```

El resultado de ejecutar este fragmento de código resulta en:

```
¡Busco un curso en París!
¡Pero Londres es genial también!
¡Pero Londres es genial también!
```

Note la tercera línea, y busque una explicación a tal sentencia.

Normalmente no se puede acceder a las variables locales de funciones desde el exterior, cuando la llamada a la función ha finalizado.

La segunda línea ¡Pero Londres es genial también! aparecerá solo si ha continuado trabajando en el mismo entorno, ¿puede explicar por qué aparece/o no aparece esta línea en su experiencia?

El siguiente ejemplo muestra una combinación bastante salvaje de variables locales y globales y parámetros de función. Por favor no corra este código, es importante para su comprensión de los contenidos. Intente predecir cuál es la salida entregada por este código.

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3, 4
foo(17, 4)
print(a, b, x, y)
```

Variables Globales en Funciones Anidadas

Examinaremos ahora qué sucederá si usamos la palabra clave global dentro de funciones anidadas. El siguiente ejemplo muestra una situación en la que se usa una variable **ciudad** en varios ámbitos:

```
def f():
    ciudad = "Hamburgo"
    def g():
        global ciudad
        ciudad = "Ginebra"
    print("Antes de llamar a g: " + ciudad)
    print("Llamando a g ahora:")
    g()
    print("Luego de llamar a g: " + ciudad)

f()
print("Valor de ciudad en programa principal: " + ciudad)
```

El resultado de la ejecución de este código entrega:

```
Antes de llamar a g: Hamburgo
Llamando a g ahora:
Luego de llamar a g: Hamburgo
Valor de ciudad en programa principal: Ginebra
```

Podemos ver que la declaración global dentro de la función anidada **g** no afecta la variable **ciudad** de la función **f,** es decir, mantiene su valor **'Hamburgo'**. También podemos deducir de este ejemplo que después de llamar a **f()** existe una variable

ciudad en el espacio de nombres del módulo y tiene el valor **'Ginebra'**. ¡Esto significa que la palabra clave global en funciones anidadas no afecta el espacio de nombres (**namespace**) de la función que anida! Esto es consistente con lo que hemos descubierto en el subcapítulo anterior: una variable definida dentro de una función es local a menos que esté marcada explícitamente como global. En otras palabras, podemos referirnos a un nombre de variable en cualquier ámbito adjunto, pero solo podemos volver a vincular nombres de variables en el ámbito local asignándolo a él o en el ámbito global del módulo mediante el uso de una declaración global. También necesitamos una forma de acceder a variables de otros ámbitos. La forma de hacerlo es con definiciones no locales, que explicaremos a continuación.

Variables no locales

Python3 introdujo variables no locales como un nuevo tipo de variables. las variables no locales tienen mucho en común con las variables globales. Una diferencia con las variables globales radica en el hecho de que no es posible cambiar las variables desde el alcance del módulo, es decir, las variables que no están definidas dentro de una función, mediante el uso de la declaración no local. Mostramos esto en los dos ejemplos siguientes:

```
def f():
    global ciudad
    print(ciudad)

ciudad = "Frankfort"
f()
```

Resultado impreso:

```
Frankfurt
```

Este programa es correcto y devuelve 'Frankfurt' como salida. Cambiaremos "global" a "nonlocal" en el siguiente programa:

```
def f():
    nonlocal ciudad
    print(ciudad)

ciudad = "Frankfort"
f()
```

Resultado:

```
  File "<stdin>", line 2
SyntaxError: no binding for nonlocal 'ciudad' found
```

Esto muestra que los enlaces **nonlocal** solo se pueden usar dentro de funciones anidadas. Se debe definir una variable no local en el ámbito de la función adjunta. Si la variable no está definida en el ámbito de la función adjunta, la variable no se puede definir en el ámbito anidado. Ésta es otra diferencia con la semántica "global".

```
def f():
    ciudad = "Munich"
    def g():
        nonlocal ciudad
        ciudad = "Zurich"
    print("Antes de llamar a g: " + ciudad)
    print("Llamando a g ahora:")
    g()
    print("Luego de llamar a g: " + ciudad)

ciudad = "Stuttgart"
f()
print("'ciudad' en principal: " + ciudad)
```

El resultado obtenido es:

```
Antes de llamar a g: Munich
Llamando a g ahora:
Luego de llamar a g: Zurich
'ciudad' en principal: Stuttgart
```

En el ejemplo anterior, la variable **ciudad** se definió antes de la llamada a `g`. Obtenemos un error si no está definida:

```
def f():
    #ciudad = "Munich"
    def g():
        nonlocal ciudad
        ciudad = "Zurich"
    print("Antes de llamar a g: " + ciudad)
    print("Llamando a g ahora:")
    g()
    print("Luego de llamar a g: " + ciudad)

ciudad = "Stuttgart"
f()
print("'ciudad' en principal: " + ciudad)
```

Como resultado tenemos:

```
  File "<stdin>", line 4
SyntaxError: no binding for nonlocal 'ciudad' found
```

El programa funciona bien, con o sin la línea 'city = "Munich"' dentro de `f`, si cambiamos "nonlocal" a "global":

```
def f():
    #ciudad = "Munich"
    def g():
        global ciudad
        ciudad = "Zurich"
    print("Antes de llamar a g: " + ciudad)
    print("Llamando a g ahora:")
    g()
    print("Luego de llamar a g: " + ciudad)

ciudad = "Stuttgart"
f()
print("'ciudad' en principal: " + ciudad)
```

La ejecución resulta en:

```
Antes de llamar a g: Stuttgart
Llamando a g ahora:
Luego de llamar a g: Zurich
'ciudad' en principal: Zurich
```

2.6.1.10. El problema de las variables globales

Según lo visto hasta ahora podemos inferir que:

- Una variable global es una variable definida en el programa 'principal'. Se dice que tales variables tienen un alcance "global".
- Una variable local es una variable definida dentro de una función. Dichas variables se dice que tienen 'alcance' local.
- Las funciones pueden acceder a variables globales y modificarlas.
- La modificación de variables globales en una función se considera una mala práctica de programación. Es mejor enviar una variable como parámetro (o hacer que se devuelva en la declaración 'return').

Fundamentalmente La razón por la cual las variables globales son malas es que permiten que las funciones tengan efectos secundarios ocultos (no obvios, sorprendentes, difíciles de detectar, difíciles de diagnosticar), lo que lleva a un

aumento en la complejidad, lo que puede conducir al código Spaghetti https://es.wikipedia.org/wiki/C%C3%B3digo_espagueti

2.6.2.- Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[3] Eric Matthes, Python Crash Course, 2016.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python Tutorial.
https://www.w3schools.com/python/