INFORMATORIO

Taller de Programación Web

Funciones

Subsecretaría de Empleo
Chaco Gobierno de todos

Ministerio de Producción, Industria y Empleo
Chaco Gobierno de todos

CHACO
Gobierno de todos

Paso de argumentos por referencia y por valor

Si conoces algún otro lenguaje de programación te estarás preguntando si ***en Python al pasar una variable como argumento de una función estas se pasan por referencia o por valor***.

En el **paso por referencia** lo que se pasa como argumento es una **referencia o puntero a la variable**, es decir, la *dirección de memoria en la que se encuentra el contenido de la variable*, y no el contenido en sí. En el **paso por valor**, por el contrario, lo que *se pasa como argumento es el valor que contenía la variable*.

Te dejamos un ejemplo en el aula para ver gráficamente esta diferencia

Pero en resumen, **la diferencia entre ambos es que en el paso por valor los cambios que se hagan sobre el parámetro no se ven fuera de la función, dado que los argumentos de la función son variables locales a la función que contienen los valores indicados por las variables que se pasaron como argumento.** Es decir, en realidad lo que se le pasa a la función son copias de los valores y no las variables en sí.

Si quisiéramos modificar el valor de uno de los argumentos y que estos cambios se reflejarán fuera de la función tendríamos que pasar el parámetro por referencia.

En C los argumentos de las funciones se pasan por valor, aunque se puede simular el paso por referencia usando punteros. En Java también se usa paso por valor, aunque para las variables que son objetos lo que se hace es pasar por valor la referencia al objeto, por lo que en realidad parece paso por referencia.

En Python también se utiliza el paso por valor de referencias a objetos, como en Java, aunque en el caso de Python, a diferencia de Java, **todo es un objeto** (para ser exactos lo que ocurre en realidad es que al objeto se le asigna otra etiqueta o nombre en el espacio de nombres local de la función).

Sin embargo ***no todos los cambios que hagamos a los parámetros dentro de una función Python se reflejarán fuera de esta***, ya que hay que tener en cuenta que en Python existen objetos inmutables, como las tuplas, por lo que si intentáramos modificar una tupla pasada como parámetro lo que ocurriría en realidad es que se crearía una nueva instancia, por lo que los cambios no se verían fuera de la función.

Veamos con este ejemplo que hace uso del método append de las listas. Un método no es más que una función que pertenece a un objeto, en este caso a una lista; y append, en concreto, sirve para añadir un elemento a una lista.

Código

```
Modificar el valor de los elementos pasados como parámetro.

def funcion(x, y):
     """Esta función actualiza los valores de los argumentos pasados como parámetro"""
    x = x + 3
    y.append(23)
    print(x, y)

x =  22
y =  [22]
funcion(x, y)
print(x, y)
```

Resultado

```
25 [22, 23]
22 [22, 23]
```

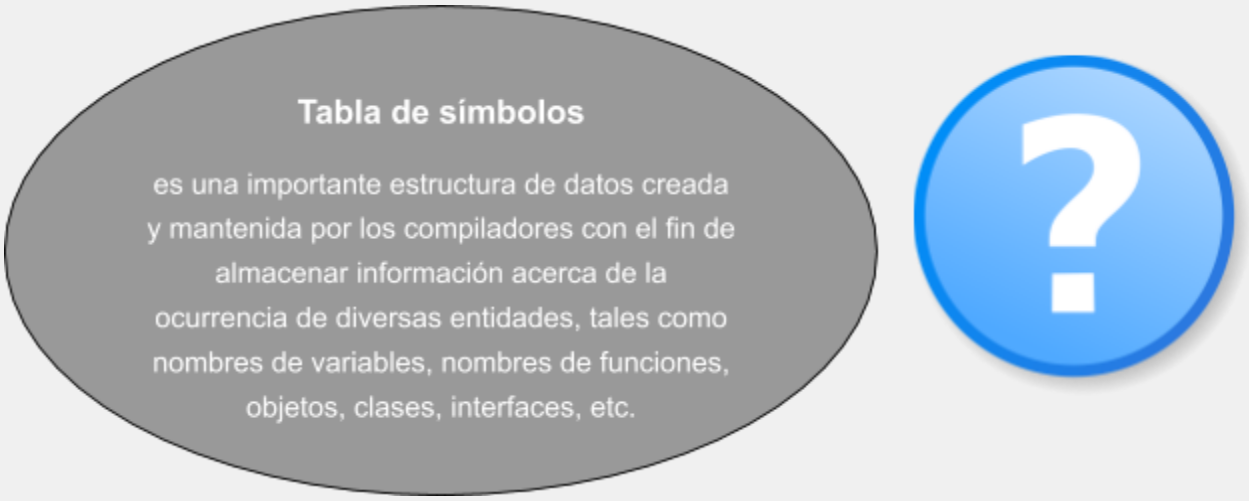
En el desarrollo y ejecución de funciones tendremos que tener en cuenta que los tipos de datos inmutables conservarán su valor original(estos son los tipos simples de datos: String, Integer, Float, Boolean, etc), sin embargo, los mutables que son estructuras de datos más "complejas"(como: dict, list, etc.) conservan los cambios hecho dentro de la función. Debido a esto podemos ver como en el ejemplo anterior la **variable x** no conserva los cambios una vez salimos de la función y la **lista y** si los conserva, porque las listas son mutables.

En resumen:

Los **valores mutables** se comportan como **paso por referencia**, y los **valores inmutables** como **paso por valor**.

Asignación de variables locales y globales

Todas las **asignaciones de variables** en la función almacenan el valor en la **tabla de símbolos local**; así mismo la referencia a variables primero mira la **tabla de símbolos local**, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, no se les puede asignar directamente un valor a las variables globales dentro de una función (a menos se las nombre en la sentencia global), aunque sí pueden ser referenciadas.



Los **parámetros reales** (argumentos) de una función se introducen en la tabla de símbolos local de la función llamada cuando esta es ejecutada; así, los argumentos son pasados por valor (donde el valor es siempre una referencia a un objeto, no el valor del objeto).

Cuando una función llama a otra función, una nueva tabla de símbolos local es creada para esa llamada.

La definición de una función introduce el nombre de la función en la tabla de símbolos actual. El valor del nombre de la función tiene un tipo que es reconocido por el interprete como una función definida por el usuario. Este valor puede ser asignado a otro nombre que luego puede ser usado como una función. Esto sirve como un mecanismo general para renombrar.

Código

```
Crear una función que actualice el valor de una variable inmutable.

mi_variable_global_1 = 20
mi_variable_global_2 = 20

def funcion():
     """Esta función actualiza el valor de una variable global de tipo
inmutable"""
    global mi_variable_global_1
    mi_variable_global_1  = 10
    mi_variable_global_2 = 10

funcion()
print(mi_variable_global_1)
print(mi_variable_global_2)
```

Resultado

```
10
20
```

Sentencia pass

Es una **operación nula**, lo que quiere decir que cuando es ejecutada nada sucede. Eso es útil como un contenedor cuando una sentencia es requerida sintácticamente, pero no necesita código que ser ejecutado, por ejemplo:

Código

```
Ejemplos de utilización de la sentencia pass.

def consultar_nombre_genero(genero):
    """Es una función que no hace nada aun"""
    pass

class Persona():
    """Es una clase sin métodos o atributos aun"""
    pass


print(type(consultar_nombre_genero))
consultar_nombre_genero("M")
persona1 = Persona
print(type(persona1))
```

Resultado

```
<class 'function'>
<class 'type'>
```

Sentencia return

Las funciones pueden comunicarse con el exterior de las mismas, al proceso principal del programa usando la sentencia return. El proceso de comunicación con el exterior se hace devolviendo valores. A continuación, un ejemplo de función usando return:

Código

```
Dado dos números calcular la suma de ambos números.

def suma(x, y):
      """Esta función imprime la suma de los dos valores pasados como parámetros"""
    return x + y

print(suma(27, 89))
```

Resultado

```
116
```

Una característica interesante, es la posibilidad de devolver valores múltiples separados por comas:

Código

```
Crear una función que retorne varios valores.

def prueba():
    """Esta función retorna varios valores"""
    return "Hola", 20, [1, 2, 3]

print(prueba())
```

Resultado

```
('Hola', 20, [1, 2, 3])
```

Sin embargo no se trata de que que las funciones Python puedan devolver varios valores, lo que ocurre en realidad es que Python crea una tupla inmutable cuyos elementos son los valores a retornar, y esta única variable es la que se devuelve.

Código

```
Crear una función que retorne varios valores y asignarlo a distintas variables.

def prueba():
    """Esta función retorna varios valores"""
    return "Hola", 20, [1, 2, 3]

print(prueba())
x, y, z = prueba()
print(x)
```

```
print(y)
print(z)
```

Resultado

```
('Hola', 20, [1, 2, 3])
Hola
20
[1, 2, 3]
```

En el código anterior se muestra como asignar valores a distintas variables en base a la tupla inmutable que retorna la función.

Llamadas de retorno

En Python, es posible (al igual que en la gran mayoría de los lenguajes de programación),llamar a una función dentro de otra de forma fija y de la misma manera que se la llamaría desde fuera de dicha función.

Código

```
Crear una función que luego sea llamada desde otra función.

def mensaje():
    """Esta función retorna el mensaje 'Hola Mundo'"""
    return "Hola Mundo"

def saludar(nombre, saludo="Hola"):
       """Esta función imprime el mensaje 'Hola Mundo' y un mensaje
personalizado"""
```


```
    print(mensaje())
    print(saludo, nombre)

saludar("Juan")
```

Resultado

```
Hola Mundo
Hola Juan
```

Sin embargo, es posible que se desee realizar dicha llamada, de manera dinámica, es decir, desconociendo el nombre de la función a la que se desea llamar. A este tipo de acciones, se las denomina llamadas de retorno.

Para conseguir llamar a una función de manera dinámica, Python dispone de dos funciones nativas: locals() y globals().

Ambas funciones, retornan un diccionario. En el caso de locals(), este diccionario se compone de todos los elementos de ámbito local, mientras que el globals(), retorna lo propio pero a nivel global.

Código

```
Crear una función que luego sea llamada desde otra función.

def mensaje():
    """Esta función retorna el mensaje 'Hola Mundo'"""
    return "Hola Mundo"

def llamada_de_retorno(funcion="" ):
    """Llamada de retorno a nivel global"""
```


```
    return globals()[funcion]()

print(llamada_de_retorno("mensaje"))
nombre_de_la_funcion = "mensaje"
print(locals()[nombre_de_la_funcion])
```

Resultado

```
Hola Mundo
<function mensaje at 0x0000018E6FAFA2F0>
```

Si se tienen que pasar argumentos en una llamada retorno, se lo puede hacer normalmente:

Código

```
Crear una función que luego sea llamada desde otra función pasando parámetros
entre estas.

def mensaje(nombre):
    """Esta función retorna un mensaje de saludo"""
    return "Hola " + nombre

def llamada_de_retorno(nombre, funcion=""):
    """Llamada de retorno a nivel global"""
    return globals()[funcion](nombre)

print(llamada_de_retorno("Maria", "mensaje"))
nombre_de_la_funcion = "mensaje"
print(locals()[nombre_de_la_funcion]("Pedro"))
```

Resultado

```
Hola Maria
Hola Pedro
```

Saber si una función existe y puede ser llamada

Durante una llamada de retorno, el nombre de la función, puede no ser el indicado.

Entonces, siempre que se deba realizar una llamada de retorno, es necesario comprobar que ésta exista y pueda ser llamada.

Código

```
Crear una función que luego sea llamada desde otra función validando la existencia de la función.

def mensaje(nombre):
    """Esta función retorna un mensaje de saludo"""
    return "Hola " + nombre

def llamada_de_retorno(nombre, funcion=""):
    """Llamada de retorno a nivel global"""
    if funcion in globals():
        return globals()[funcion](nombre)
    else:
        return "La función no existe"

print(llamada_de_retorno("Maria", "mensaje"))
print(llamada_de_retorno("Maria", "saludo"))
```

Resultado

```
Hola Maria
La función no existe
```

El operador **in**, nos permitirá conocer si un elemento se encuentra dentro de una colección, mientras que la función callable() nos dejará saber si esa función puede ser llamada.

Código

```
Crear una función que luego sea llamada desde otra función validando la existencia de la función.

def mensaje(nombre):
    """Esta función retorna un mensaje de saludo"""
    return "Hola " + nombre

def llamada_de_retorno(nombre, funcion=""):
    """"Llamada de retorno a nivel global"""
    if funcion in globals():
      if callable(globals()[funcion]):
           return globals()[funcion](nombre)
    else:
      return "La función no existe"

print(llamada_de_retorno("Maria", "mensaje"))
print(llamada_de_retorno("Maria", "saludo"))
```

Resultado

```
Hola Maria
La función no existe
```

Llamadas recursivas

Se denomina **llamada recursiva** (o recursividad), a aquellas funciones que en su algoritmo, hacen referencia sí misma. Las llamadas recursivas suelen ser muy útiles en casos muy puntuales, pero debido a su gran factibilidad de caer en iteraciones infinitas, deben extremarse las medidas preventivas adecuadas y, solo utilizarse cuando sea estrictamente necesario y no exista una forma alternativa viable, que resuelva el problema evitando la recursividad. Python admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma,de igual forma que lo hace cuando llama a otra función.

Código

```
Crear una función que realice una llamada recursiva.

def jugar(intento=1):
    """Llamada de retorno a nivel global"""
    respuesta = input("¿De qué color es una naranja?")
    if respuesta != "naranja":
       if intento < 3:
         print("Fallaste. Intenta de nuevo")
         intento += 1
         jugar(intento)
      else:
        print("Perdiste")
    else:
      print("Ganaste")

jugar()
```

Resultado

```
¿De qué color es una naranja?>? Azul
```


```
Fallaste. Intenta de nuevo
¿De qué color es una naranja?>? Verde
Fallaste. Intenta de nuevo
¿De qué color es una naranja?>? naranja
Ganaste
```

Para qué me sirven las funciones?

Una función, puede tener cualquier tipo de algoritmo y cualquier cantidad de ellos y,utilizar cualquiera de las características vistas hasta ahora. No obstante, una buena práctica, indica que la finalidad de una función, debe ser realizar una única acción,reutilizable y por lo tanto, tan genérica como sea posible.