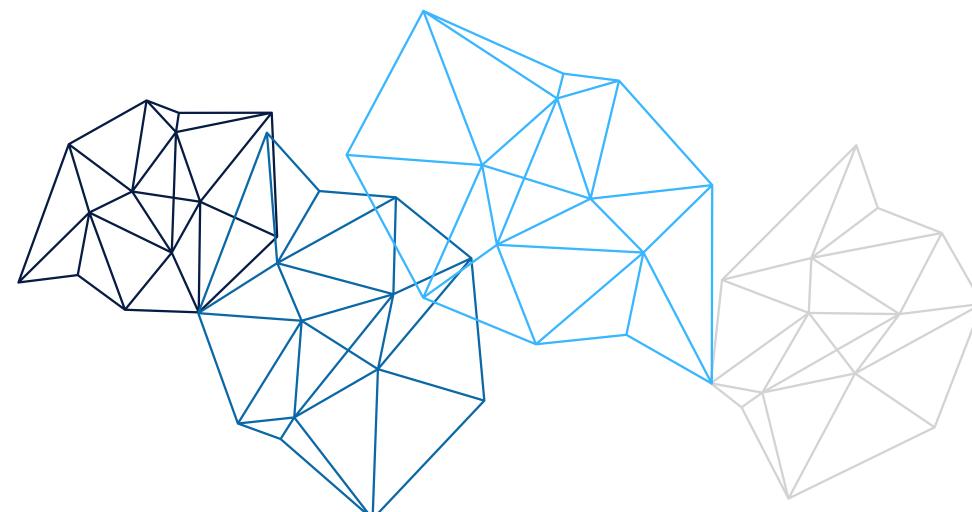


Funciones

Una función es *un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor*. A los fragmentos de código que tienen un nombre asociado y **no** devuelven valores se les suele llamar procedimientos. En Python **no** existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor **None** (nada).

El uso de funciones es un componente muy importante del paradigma de la programación llamada **estructurada**, y tiene varias ventajas:

- Modularización: permite segmentar un programa complejo en una serie de partes o módulos más simples, facilitando así la programación y el depurado.
- Reutilización: permite reutilizar una misma función en distintos programas



Sentencia def

La sentencia def es una definición de función usada para crear funciones definidas por el usuario. Una **definición de función** es una sentencia ejecutable. Su ejecución enlaza el nombre de la función en el *namespace* local actual a un objeto función (un envoltorio alrededor del código ejecutable para la función). Este objeto función contiene una referencia al *namespace* local como el *namespace* global para ser usado cuando la función es llamada.

La definición de la función no ejecuta el cuerpo de la misma; esto es ejecutado solamente cuando la función es llamada.

En Python las funciones se declaran de la siguiente forma:

```
>>def mi_funcion(param1, param2):  
    print (param1 print param2)
```

Es decir, la palabra clave **def** seguida del **nombre de la función** y entre paréntesis **cero o más argumentos** como entrada separados por comas, en otra línea, indentado y después de los dos puntos tendríamos las **líneas de código** que conforman el conjunto de instrucciones a ser ejecutadas por la función.



INFORMATARIO
Hacia una mejor industria informática

DocStrings

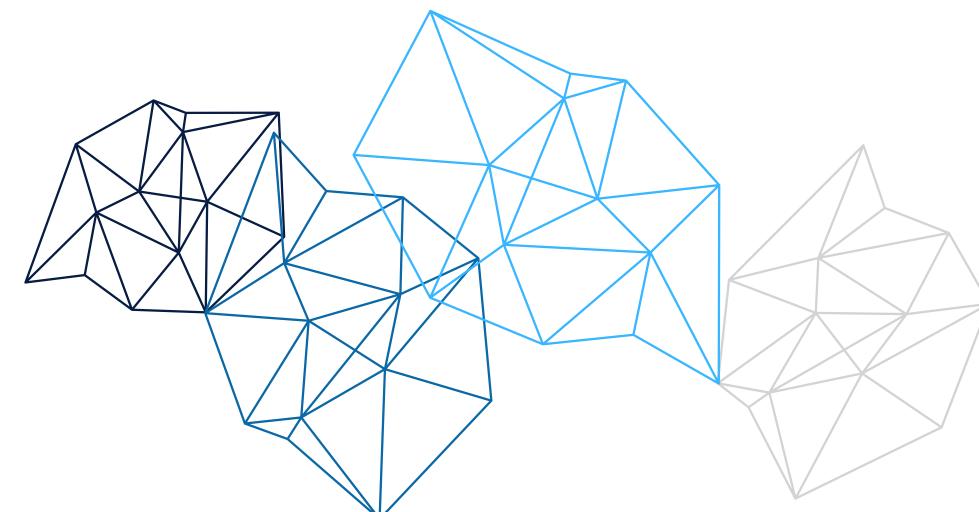
También podemos encontrarnos con una **cadena de texto** como primera línea del cuerpo de la función, estas cadenas se conocen con el nombre de **docstring** (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

Hay herramientas que usan las **docstrings** para producir automáticamente documentación en línea o imprimible, o para permitirle al usuario que navegue el código en forma interactiva; es una buena práctica incluir docstrings en el código que uno escribe, **por lo que se debe hacer un hábito de esto.**

```
>>def mi_funcion(param1, param2):  
    """Esta función imprime los dos valores pasados como parámetros"""\n    print (param1)\n    print (param2)
```

INFORMATARIO

Hacia una mejor industria informática



Ejemplo

Como se mencionó anteriormente en la definición de la función lo único que hacemos es asociar un nombre al fragmento de código que conforma la función, de forma que podamos ejecutar dicho código más tarde referenciándolo por su nombre.

Para llamar a la función (ejecutar su código) se escribiría:

```
>>mi_funcion("Hola", 2)
```

Es decir, el nombre de la función a la que queremos llamar seguido de los valores que queramos pasar como parámetros entre paréntesis.

INFORMATORIO
Hacia una mejor industria informática



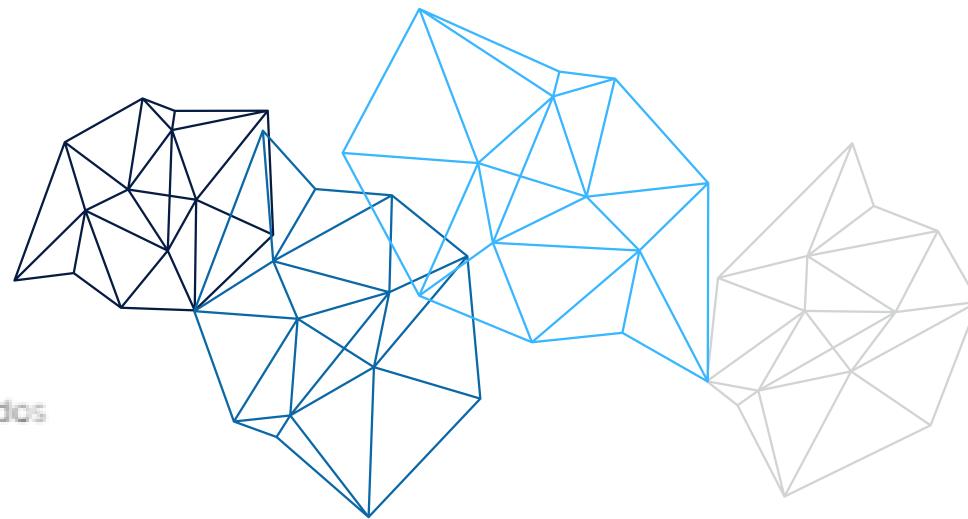
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos



Argumentos y parámetros

Al **definir** una función los valores que se reciben se denominan **parámetros**, pero durante la **llamada** los valores que se envían se denominan **argumentos**.

```
>>def sumar(a,b): <-- Acá a y b son PARÁMETROS  
    resultado = a+b  
    return resultado
```

x = 2

y = 3

```
resultado_suma = sumar(x,y) <-- Acá x e y son ARGUMENTOS  
print(resultado_suma)
```



INFORMATARIO
Hacia una mejor industria informática



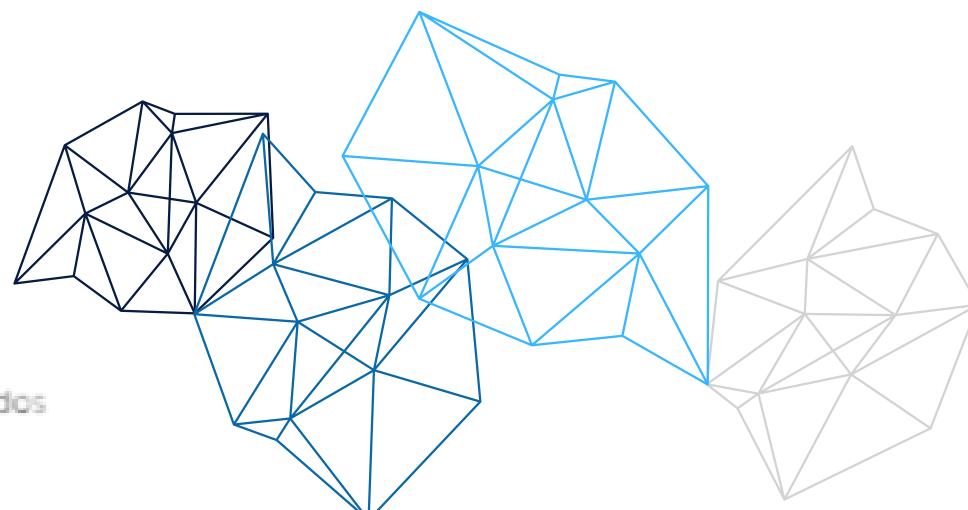
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos



Paso de parámetros

- Por posición:

Cuando se envían **argumentos** a una función, estos se reciben por **orden** en los parámetros definidos. La asociación de los parámetros y los valores pasados a la función se hace normalmente de izquierda a derecha.

- Por nombre:

Sin embargo también es posible modificar el orden de los parámetros si indicamos el nombre del parámetro al que asociar el valor a la hora de llamar a la función:

```
>>sumar(b = 3, a=2)
```

- Llamada sin argumentos:

El número de valores que se pasan como parámetro al llamar a la función tiene que coincidir con el número de parámetros que la función acepta según la declaración de la función. Si al momento de llamar una función la cual tiene definidos unos parámetros no pasa los argumentos correctamente provocará una excepción *TypeError*.

- Parámetros por defecto:

Para solucionar la excepción *TypeError* ejecutada al momento de la llamada a una función sin argumentos, entonces se puede asignar unos valores por defecto nulos a los parámetros, de esa forma puede hacer una comprobación antes de ejecutar el código de la función.

Argumentos indeterminados

En alguna ocasión no podremos determinar previamente cuantos elementos se necesita enviar a una función. En estos casos se puede utilizar los argumentos indeterminados por posición y por nombre.

- Por posición: (*args)

Se debe crear una lista dinámica de argumentos, es decir, un tipo tupla, definiendo el parámetro con un asterisco, para recibir los parámetros indeterminados por posición.

- Por nombre: (**kwargs)

Para recibir un número indeterminado de parámetros por nombre (clave-valor o en inglés keyword args), se debe crear un diccionario dinámico de argumentos definiendo el parámetro con dos asteriscos. Las claves de este diccionario serían los nombres de los parámetros indicados al llamar a la función y los valores del diccionario, los valores asociados a estos parámetros

- Por posición y nombre: (*args,**kwargs)

Si requiere aceptar ambos tipos de parámetros simultáneamente en una función, entonces debe crear ambas colecciones dinámicas. Primero los argumentos indeterminados por valor y luego los cuales son por clave y valor

Los nombres args y kwargs no son obligatorios, pero se suelen utilizar por convención. Muchos frameworks y librerías los utilizan por lo que es una buena práctica llamarlos así

Paso de argumentos por referencia y 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. 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 Python se utiliza el paso por valor de referencias a objetos, en Python, a diferencia de otros lenguajes, 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

Paso de argumentos

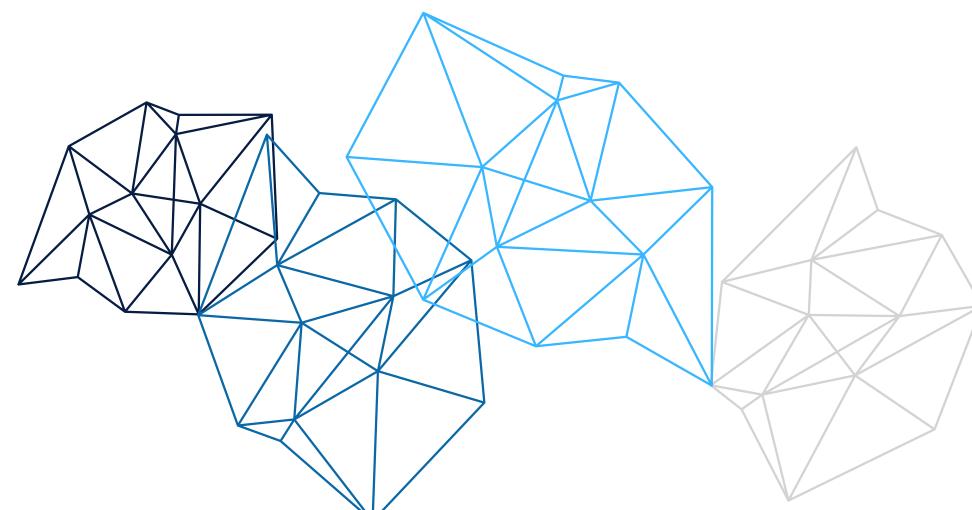
En el desarrollo y ejecución de funciones tendremos que tener en cuenta que los tipos de datos inmutables conservarán su valor original (estos son los tipos simples de datos: String, Integer, Float, Boolean, etc), sin embargo, los mutables que son estructuras de datos más “complejas” (como: dict, list, etc.) conservan los cambios hecho dentro de la función.

En resumen:

LOS VALORES **MUTABLES** SE COMPORTAN COMO PASO POR **REFERENCIA**

LOS VALORES **INMUTABLES** COMO PASO POR **VALOR**

INFORMATARIO
Hacia una mejor industria informática



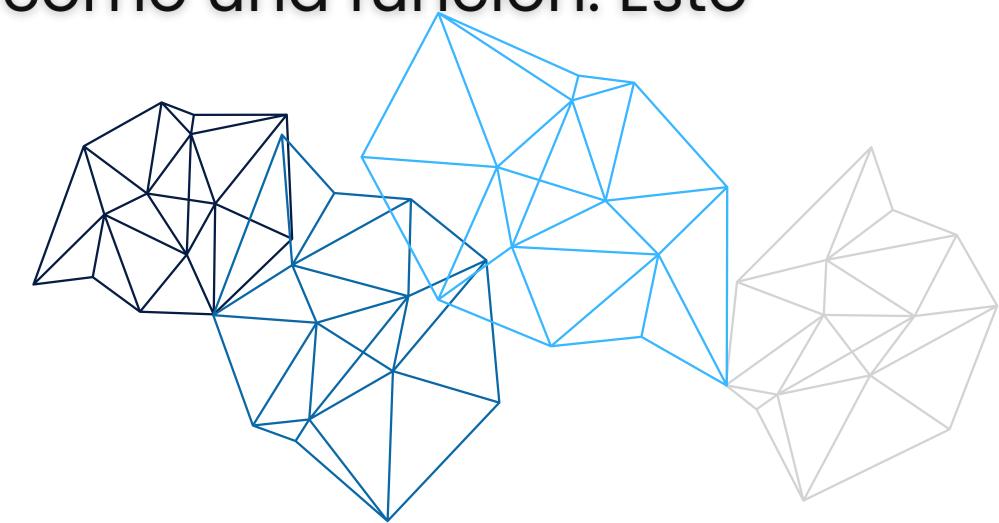
Asignación de variables locales y globales

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.



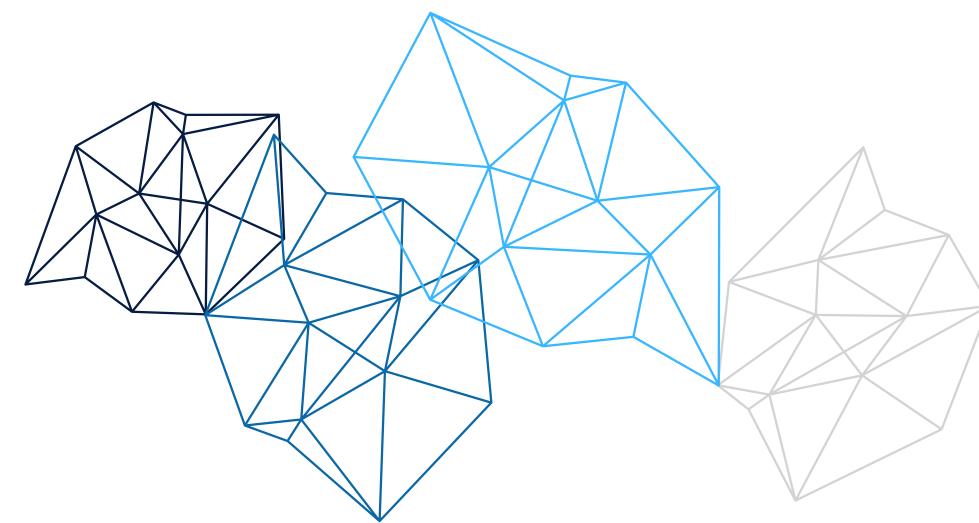
Sentencias

- 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.

- 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. Una característica interesante, es la posibilidad de devolver valores múltiples separados por comas, sin embargo, no se trata de 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.



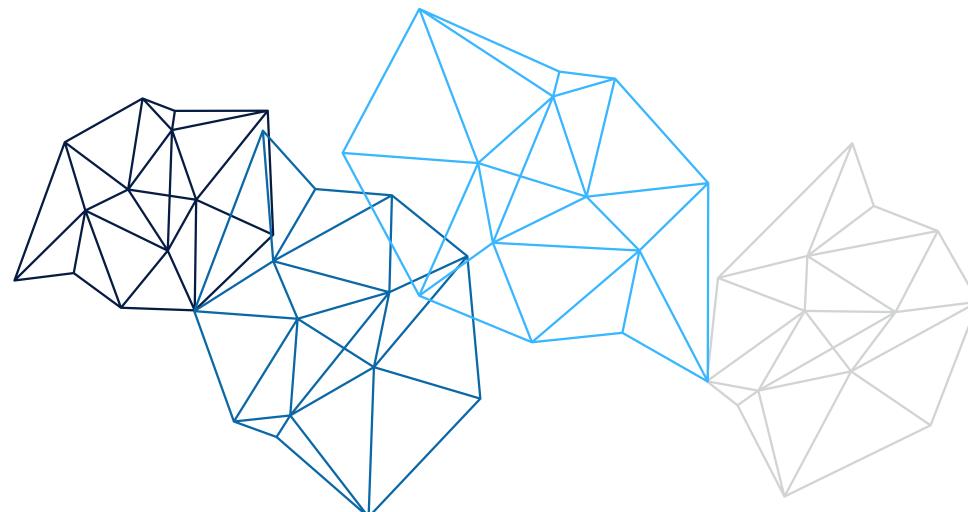
Llamadas de retorno

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.

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.

INFORMATARIO

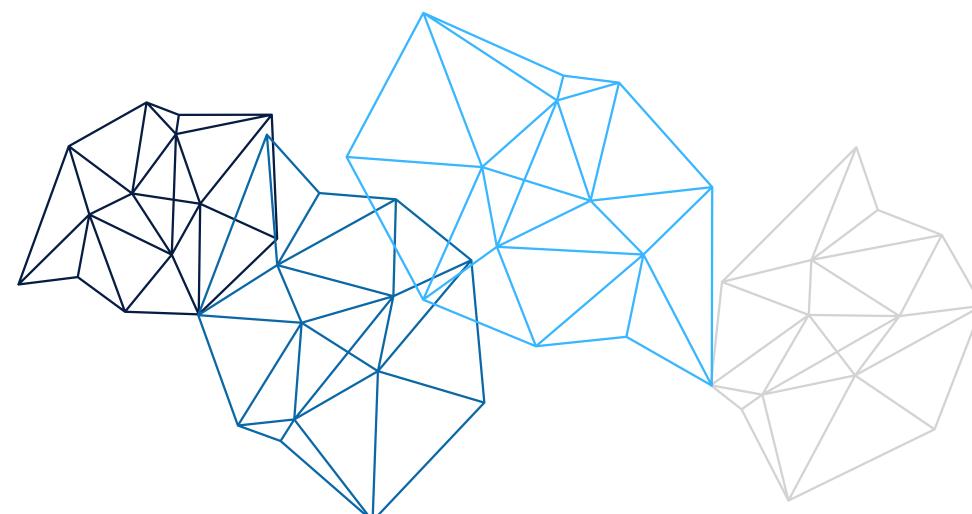
Hacia una mejor industria informática



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.

INFORMATORIO
Hacia una mejor industria informática



¿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.**

INFORMATARIO

Hacia una mejor industria informática

