# Contenido


Función ..... 2
Definiendo funciones ..... 3
Retornos de funciones ..... 4
Parámetros de Funciones ..... 5
- Ámbito de Parámetros ..... 6
- Parámetros por omisión ..... 7
- Parámetros por omisión con claves-valor ..... 8
- Parámetros arbitrarios ..... 9
- Parámetros arbitrarios con clave-valor ..... 10
- Desempaquetado de parámetros ..... 11
- Desempaquetado de parámetros con clave-valor ..... 12

Llamadas de retorno ..... 13
- Llamadas recursivas ..... 14
- Llamadas recursivas – Ejemplo ..... 15

Sobre la finalidad de las funciones ..... 16




# Función

Una función, es la forma de agrupar expresiones y sentencias (algoritmos) que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas.

Es decir que, al colocar un algoritmo dentro de una función, al correr el archivo, el algoritmo no será ejecutado si no se ha hecho una referencia a la función que lo contiene.



# Definiendo funciones

Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone, irá indentado con 4 espacios:

```
def mi_funcion():
    # aquí el algoritmo
```

Una función, no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
    print ("Hola Mundo")

mi_funcion()        # imprime Hola Mundo
```



# Retornos de funciones

Los retornos son datos que la funciones nos devuelve a ser invocadas para poder guardarlas o manipularlas en nuestro programa.

Se utiliza la palabra reservada **return** para indicar el dato que vamos a retornar con nuestra función.

Cuando una función, haga un retorno de datos, éstos, pueden ser asignados a una variable:

```
def funcion():
    return "Hola Mundo"

frase = funcion()
print(frase) #imprime Hola Mundo
```



# Parámetros de Funciones

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo.

Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno:

```
def mi_funcion(nombre, apellido):
    # algoritmo
```

Los parámetros, se indican entre los paréntesis, a modo de variables, a fin de poder utilizarlos como tales, dentro de la misma función.

```
def mi_funcion(nombre, apellido):
    print(nombre)      # esto imprime lo que contenga la variable nombre
    print(apellido)    # esto imprime lo que contenga la variable apellido
```



# Ámbito de Parámetros

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de variables de ámbito local.
Es decir, que los parámetros serán variables locales, a las cuáles sólo la función podrá acceder:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre, apellido
    print(nombre_completo)
```

Si quisiéramos acceder a esas variables locales, fuera de la función, obtendremos un error:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre, apellido
    print(nombre_completo)

print nombre # Retornará el error: NameError: name 'nombre' is not defined
```



## Parámetros por omisión

En Python, también es posible asignar valores por defecto a los parámetros de las funciones.

Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def saludar(nombre, mensaje='Hola'):
    print(mensaje, nombre)

saludar('Pepe Grillo') # Imprime: Hola Pepe Grillo
```

Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo “**=**”



## Parámetros por omisión con claves-valor

En Python, también es posible llamar a una función, pasándole los argumentos esperados, como pares de claves=valor:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre

saludar(mensaje="Buen día", nombre="Juancho")
```

Aquí lo que sucede es que al valor del parámetro mensaje lo sustituimos con un valor que le pasamos nosotros al llamar a la función.



## Parámetros arbitrarios

Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir una cantidad arbitraria (desconocido) de argumentos.

Estos argumentos que se envía a la función, llegarán a la misma como un parámetro en forma de tupla: **arbitrarios será una tupla**.

Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco **(*)**:

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios):
    print(parametro_fijo)
    # Los parámetros arbitrarios se corren como tuplas
    for argumento in arbitrarios:
        print (argumento)

recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1', 'arbitrario 2', 'arbitrario 3')
```



# Parámetros arbitrarios con clave-valor

Es posible también, obtener parámetros arbitrarios como pares de clave=valor.

En estos casos, al nombre del parámetro deben precederlo dos asteriscos **(**):**

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios, **kwords):
    print (parametro_fijo)
    for argumento in arbitrarios:
        print (argumento)
    # Los argumentos arbitrarios tipo clave, se recorren como los diccionarios
    for clave in kwords:
        print ("El valor de", clave, "es", kwords[clave])

recorrer_parametros_arbitrarios("Fixed", "arbitrario 1", "arbitrario 2", "arbitrario 3", clave1="valor uno",
clave2="valor dos")
```



# Desempaquetado de parámetros

Puede ocurrir, además, una situación inversa a la anterior.
Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla.

En este caso, el signo asterisco **(*)** deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función:

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = [1500, 10]
print(calcular(*datos))
```



## Desempaquetado de parámetros con clave-valor

El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se encuentren disponibles en un diccionario.

Aquí, deberán pasarse a la función, precedidos de dos asteriscos **(**)**:

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = {"descuento": 10, "importe": 1500}
print(calcular(**datos))
```



# 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:

```
def mi_funcion():
    return "Hola Mundo"

def saludar(nombre, mensaje='Hola'):
    print(mensaje, nombre9
    print(mi_funcion())
```



# Llamadas recursivas

Se denomina llamada recursiva (o recursividad), a aquellas funciones que, en su algoritmo, hacen referencia a 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.

Hay que tener bien en claro cuáles son los casos **base** (Son aquellos que para su solución no requieren utilizar la **función** que se está definiendo) y los casos **recursivos** (Son aquellos que sí que requieren utilizar la **función** que se está definiendo).



# Llamadas recursivas – Ejemplo

Observemos el siguiente ejemplo:

```
def jugar(intento=1):
    respuesta = input("¿De qué color es una naranja? ")
    if respuesta != "naranja":
        if intento < 3:
            print "\nFallaste! Inténtalo de nuevo"
            intento += 1
            jugar(intento) #<---- Llamada recursiva
        else:
            print "\nPerdiste!"
    else:
        print "\nGanaste!"
jugar()
```



# Sobre la finalidad de 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.