

## Modularidad

La modularidad es la característica de un sistema que permite que sea estudiado, visto o entendido como la **unión de varias partes** que interactúan entre sí y que trabajan solidariamente para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de **módulo**.

Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de entradas y salidas bien definidas.

En relación a ello a continuación veremos, **Funciones** y **Módulos** en Python, temas sumamente importantes para comenzar a organizar nuestros programas.

### ➤ Funciones

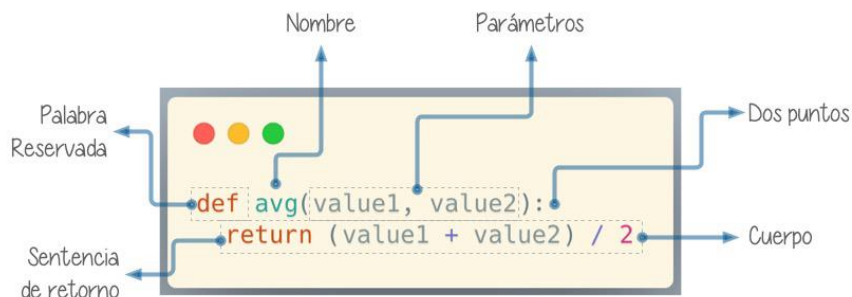
El concepto de función es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura que nos permite agrupar código. Persigue dos objetivos claros:

- No repetir trozos de código durante nuestro programa.
- Reutilizar el código para distintas situaciones.

Una función viene *definida* por su *nombre*, sus *parámetros* y su *valor de retorno*. Esta parametrización de las funciones las convierte en una poderosa herramienta ajustable a las circunstancias que tengamos. Al invocarla estaremos solicitando su ejecución y obtendremos unos resultados

### Definir una función

Para definir una función utilizamos la palabra reservada **def** seguida del nombre de la función. A continuación aparecerán 0 o más parámetros separados por comas (entre paréntesis), finalizando la línea con **dos puntos** : En la siguiente línea empezaría el cuerpo de la función que puede contener 1 o más sentencias, incluyendo (o no) una sentencia de retorno con el resultado mediante **return**.



Ejemplo:

```
def funcion_prueba():  
    print ("PRIMER MENSAJE IMPRESO EN LA FUNCION")  
    print ("SEGUNDO MENSAJE IMPRESO EN LA FUNCION")  
  
print ("MENSAJE IMPRESO EN EL PROGRAMA PRINCIPAL")
```

**Nota:** Note la indentación que hay en los dos print que hay a continuación de la definición de la función, eso indica que ambas sentencias pertenecen a dicha función, sin embargo el tercer print NO pertenece a la función sino al cuerpo principal del programa. Al ejecutar esta porción de código se obtiene como resultado la impresión del mensaje: *MENSAJE IMPRESO EN EL PROGRAMA PRINCIPAL*, debido a la función NO ha sido llamada desde ningún lugar dentro del programa principal.

### Invocar una función

Para invocar (o *llamar*) a una función sólo tendremos que escribir su nombre seguido de paréntesis.

```
def funcion_prueba():  
    print ("PRIMER MENSAJE IMPRESO EN LA FUNCION")  
    print ("SEGUNDO MENSAJE IMPRESO EN LA FUNCION")  
  
funcion_prueba()  
print ("MENSAJE IMPRESO EN EL PROGRAMA PRINCIPAL")
```

Como puedes ver, desde el programa principal se “llama” a la función, se transfiere la ejecución del programa a ella, se ejecuta y luego se retorna a la siguiente sentencia del programa principal. Con ello, la ejecución del programa anterior imprime lo siguiente:

```
PRIMER MENSAJE IMPRESO EN LA FUNCION  
SEGUNDO MENSAJE IMPRESO EN LA FUNCION  
MENSAJE IMPRESO EN EL PROGRAMA PRINCIPAL
```

**NOTA:** NO deben haber declaradas 2 funciones con el mismo nombre en un mismo archivo o módulo.-

### Retornar un valor

Como se puede apreciar, en los ejemplos anteriores la función desarrollada no le “aporta” al programa principal más que una serie de mensajes, y si quisiéramos también podríamos desarrollar en ella un par de sentencias que realicen cálculos propios del problema que estamos resolviendo, por ejemplo, talvez tengamos la necesidad de que la función realice la suma de los sueldos que tiene un *diccionario* definido por nosotros, pero solo podríamos mostrar esa suma por pantalla:

---

```
def funcion_suma_sueldo ():  
    diccionario_sueldo_años = {  
        2019: 100000,  
        2020: 110000,  
        2021: 125000,  
        2022: 140000  
    }  
    suma_sueldo = 0  
    for sueldo in diccionario_sueldo_años.values():  
        suma_sueldo += sueldo  
  
    print ("SUMATORIA: ", suma_sueldo)  
  
funcion_suma_sueldo()
```

Como se puede apreciar, la función realiza el cálculo de la suma de los valores definidos en el diccionario de sueldos, pero la suma obtenida solo la muestra por pantalla. Se dice que dicha función NO le aporta ninguna información al programa principal, si quisiéramos que le aportara información deberíamos hacer que la función retorne algún dato, en este ejemplo podría ser la sumatoria de los sueldos. Veamos:

```
def funcion_suma_sueldo():  
    diccionario_sueldo_años = {  
        2019: 100000,  
        2020: 110000,  
        2021: 125000,  
        2022: 140000  
    }  
    suma_sueldo = 0  
    for sueldo in diccionario_sueldo_años.values():  
        suma_sueldo += sueldo  
  
    return suma_sueldo  
  
calcula_sumatoria = funcion_suma_sueldo()  
print ("Sumatoria de Sueldos devuelta por la Funcion: ", calcula_sumatoria)
```

Ahora vemos que para devolver un dato, la función utiliza la sentencia “**return**” y en el programa principal ese valor retornado por la función se le asigna a la variable “calcula\_sumatoria” para luego imprimirlo en pantalla.

**Nota:** En la sentencia return podemos incluir variables, expresiones y literales.

---

## Parámetros y argumentos

Generalmente, cada vez que desarrollamos una función no solo vamos a tener la necesidad de que nos devuelva información al programa principal, sino también de **enviarle** información para que cumpla con su objetivo.

Es por ello que los parámetros nos permiten variar los datos que consume una función para obtener distintos resultados.

Cuando llamamos a una función con **argumentos**, los valores de estos argumentos se copian en los correspondientes **parámetros** dentro de la función.

```
def funcion_suma(parametro_a, parametro_b): #estos son los parametros

    return (parametro_a+parametro_b)

calculo_suma = funcion_suma(10, 20) #estos son los argumentos
print ("Sumatoria devuelta por la Funcion: ", calculo_suma)
```

Imprime: Sumatoria devuelta por la Funcion: 30

- **Argumentos Posicionales**

Cuando llamamos a una función con argumentos, los valores de estos argumentos se copian en los correspondientes parámetros dentro de la función. En el ejemplo, significa que el valor 10 es representado dentro de la función por el parámetro que se llama “**parámetro\_a**” y el valor 20 es representado por “**parámetro\_b**”.

Una función no solo puede recibir parámetro “*atomicos*” (enteros, flotantes, string) sino también “*compuestos*” como diccionarios o tuplas:

```
def funcion_suma_sueldo(parametro_diccionario):

    suma_sueldo = 0
    for sueldo in parametro_diccionario.values():
        suma_sueldo += sueldo

    return suma_sueldo

diccionario_sueldo_años = {
    2019: 100000,
    2020: 110000,
    2021: 125000,
    2022: 140000
}

calculo_sumatoria = funcion_suma_sueldo(diccionario_sueldo_años)
print ("Sumatoria de Sueldos devuelta por la Funcion: ", calculo_sumatoria)
```

En ese ejemplo se ve que *diccionario\_sueldo\_años* es un diccionario que pertenece al programa principal y es pasado como parámetro a la función para que haga su trabajo.

Una de las ventajas de organizar nuestro programa en funciones es que podemos *reutilizar* código, llamando a la función cada vez que la necesite incluso con distintos parámetros, para obtener, lógicamente, resultados distintos:

```
def funcion_suma_sueldo(parametro_diccionario):

    suma_sueldo = 0
    for sueldo in parametro_diccionario.values():
        suma_sueldo += sueldo

    return suma_sueldo

diccionario_sueldo_años = {
    2019: 100000,
    2020: 110000,
    2021: 125000,
    2022: 140000
}
diccionario_sueldo_años_anteriores = {
    2015: 55000,
    2016: 60000,
    2017: 75000,
    2018: 80000
}

calculo_sumatoria = funcion_suma_sueldo(diccionario_sueldo_años)
print ("Sumatoria de los ultimos Sueldos: ", calculo_sumatoria)

calculo_sumatoria=funcion_suma_sueldo(diccionario_sueldo_años_anteriores)
print ("Sumatoria de Sueldos Antiguos: ", calculo_sumatoria)
```

Como podemos apreciar, a la función se la llama 2 veces desde el programa principal con parámetros distintos para realizar cálculos de acuerdo a la ocasión.

- **Argumentos Nominales**

En esta aproximación los argumentos no son copiados en un orden específico sino que se **asignan por nombre a cada parámetro**. Ello nos permite salvar el problema de conocer cuál es el orden de los parámetros en la definición de la función. Para utilizarlo, basta con realizar una asignación de cada argumento en la propia llamada a la función.

```
def funcion_suma(parametro_a, parametro_b):
    return (parametro_a-parametro_b)

calculo_suma = funcion_suma(parametro_b=10 , parametro_a=20)
```

---

```
print ("Sumatoria devuelta por la Funcion: ", calculo_suma)
```

Como se puede apreciar en el ejemplo, cuando se hace la llamada a la función se indica el nombre del parámetro al que se le asigna cada “argumento”, es decir que el valor 10 será representado en la función a través del parámetro llamado “parámetro\_b” y NO por el primer parámetro indicado de izquierda a derecha como lo fue con el primer ejemplo (argumentos posicionales).

De esta manera no tenemos que preocuparnos en qué orden pasarle los argumentos a la función cada vez que la llamemos.

Python permite mezclar el conceptos de parámetros nominales y posicionales en una llamada a función, solo que los posicionales deben ir en primer lugar antes (de izquierda a derecha) que los nominales.

- **Parámetros por defecto**

Es posible especificar valores por defecto en los parámetros de una función. En el caso de que no se proporcione un valor al argumento en la llamada a la función, el parámetro correspondiente tomará el valor definido por defecto.

Ejemplo 1:

```
def funcion_suma(parametro_a=100, parametro_b=200, parametro_c=300):  
    return (parametro_a+parametro_b+parametro_c)  
  
calculo_suma = funcion_suma(10, 20, 30)  
print ("Sumatoria devuelta por la Funcion: ", calculo_suma)
```

**Imprime:** 60

Ejemplo 2:

```
def funcion_suma(parametro_a=100, parametro_b=200, parametro_c=300):  
    return (parametro_a+parametro_b+parametro_c)  
  
calculo_suma = funcion_suma(10, 20)  
print ("Sumatoria devuelta por la Funcion: ", calculo_suma)
```

**Imprime:** 330 (aquí el parametro\_c toma el valor por defecto 300)

Ejemplo 3:

```
def funcion_suma(parametro_a=100, parametro_b=200, parametro_c=300):  
    return (parametro_a+parametro_b+parametro_c)  
  
calculo_suma = funcion_suma(10, parametro_c=30)  
print ("Sumatoria devuelta por la Funcion: ", calculo_suma)
```

**Imprime:** 240 (aquí el parametro\_b toma el valor por defecto 200)

## **Empaquetar/Desempaquetar argumentos**

Python nos ofrece la posibilidad de empaquetar y desempaquetar argumentos cuando estamos invocando a una función, tanto para **argumentos posicionales** como para **argumentos nominales**.

Y de este hecho se deriva que podamos utilizar un *número variable de argumentos* en una función, algo que puede ser muy interesante según el caso de uso que tengamos.

- **Empaquetar/Desempaquetar argumentos posicionales**

Si utilizamos el operador \* delante del nombre de un parámetro posicional, estaremos indicando que los argumentos pasados a la función se empaqueten en una **tupla**:

```
def funcion_prueba(*parametro):  
    print ("Imprimo todos los elementos: ",parametro)  
    print ("Imprimo solo el primer elemento: ",parametro[0])  
  
funcion_prueba(10, 20, 30)
```

Se imprime los valores 10,20 y 30 pero como una **TUPLA**. Y además se imprime el valor del primer elemento.

También podemos utilizar esta estrategia para establecer en una función una serie de parámetros como *requeridos* y recibir el resto de *argumentos como opcionales* y *empaquetados*:

```
def funcion_prueba(parametro_a, parametro_b, *parametros):  
    total = 0  
    for value in (parametro_a, parametro_b) + parametros:  
        total += value  
    return total  
  
valor_retornado = funcion_prueba(10, 20, 30, 40, 50)  
print ("Valor devuelto por la Funcion: ", valor_retornado)  
valor_retornado = funcion_prueba(10, 20)  
print ("Valor devuelto por la Funcion (2): ", valor_retornado)
```

Se imprime:

Valor devuelto por la Funcion (1): 150

Valor devuelto por la Funcion (2): 30

Como se puede apreciar, el tercer parámetro está definido como opcional y además empaquetado. Los parámetros *parametro\_a* y *parametro\_b* son obligatorios.

- **Empaquetar/Desempaquetar argumentos nominales**

Si utilizamos el operador **\*\*** delante del nombre de un parámetro nominal, estaremos indicando que los argumentos pasados a la función se empaqueten en un **diccionario**:

```
def funcion_prueba(**parametros):  
    print("Diccionario completo: ",parametros)  
    for clave in parametros.keys():  
        print (" Clave: ", clave, end=",")  
    total = 0  
    for value in parametros.values():  
        total += value  
    return total  
  
valor_retornado = funcion_prueba(a=10, b=20, c=30, d=40, e=50)  
print ("\nValor devuelto por la Funcion (1): ", valor_retornado)  
valor_retornado = funcion_prueba(a=10, b=20)  
print ("\nValor devuelto por la Funcion (2): ", valor_retornado)
```

Imprime:

Diccionario completo: {'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

Clave: a, Clave: b, Clave: c, Clave: d, Clave: e,

Valor devuelto por la Funcion (1): 150

Diccionario completo: {'a': 10, 'b': 20}

Clave: a, Clave: b,

Valor devuelto por la Funcion (2): 30

**NOTA IMPORTANTE:** El hecho de que los parámetros sean “empaquetados” en un diccionario, eso no quiere decir que las modificaciones que se realicen en los elementos de ese diccionario dentro de la función tengan efecto directo sobre las variables relacionadas en el programa principal (variables parámetros con las que se llamó a la función, en el ejemplo a,b,c,d,e). El “empaquetado” de los parámetros en diccionario es solo para poder identificarlos dentro de la función a través de una sola variable estructurada.

## Modificando parámetros mutables

Hay que tener cuidado a la hora de manejar los parámetros que pasamos a una función ya que podemos obtener resultados indeseados, especialmente cuando trabajamos con *tipos de datos mutables*.



Supongamos una función que modifica elementos de una lista que pasamos por parámetro. Hagamos una serie de pruebas pasando alguna lista como argumento y veamos las consecuencias en el programa principal:

```
def funcion_prueba(parametro_lista, parametro_atómico):  
    print("Lista dentro de la FC: ", parametro_lista)  
    print("Variable comun dentro de la FC: ", parametro_atómico)  
    parametro_lista[1] = 200  
    parametro_atómico = 2000  
    print("Variable comun dentro de la FC (modificado): ",  
          parametro_atómico)  
  
lista = [10,20,30,40,50]  
variable_comun = 1000;  
print("Lista antes de llamar a la FC: ", lista)  
print("Variable comun antes de llamar a la FC: ", variable_comun)  
funcion_prueba(lista,variable_comun )  
print("Lista despues de llamar a la FC: ", lista)  
print("Variable comun despues de llamar a la FC: ", variable_comun)
```

Como se puede apreciar en el ejemplo anterior, los elementos de la lista pueden ser modificados dentro de la función y también se modificara la lista argumento en el programa principal (variable identificada con el nombre *lista*), sin embargo, algo que NO sucede con el parámetro “atómico” ya que la modificación solo tiene implicancia dentro del ambiente de la función, pero NO afecta al argumento en el programa principal (variable identificada con el nombre *variable\_comun*)

La explicación conceptual de lo que sucedió anteriormente es debido a que hay 2 maneras de pasar argumentos a una función, una de ella se llama **por valor** y la otra es **por referencia**.

- ✚ **Por valor** significa que el parámetro vinculado en la función (en el ejemplo anterior *parametro\_atómico*) es una **copia** del argumento (*variable\_comun*), con lo cual cualquier modificación dentro de la función solo tendrá implicancia dentro de la función pero NO modifica a *variable\_comun* en el programa principal.
- ✚ **Por referencia** significa que el parámetro vinculado en la función (en el ejemplo anterior *parametro\_lista*) es una **referencia** del argumento (*lista*), es decir, que lo que se pasa como parámetro es la dirección de memoria de la variable *lista*, con lo cual cualquier modificación dentro de la función tendrá implicancia directa no solo en el parámetro dentro de la función sino también en el argumento vinculando en el programa principal, porque al hacer la modificación se modifica el contenido de un espacio de memoria.

En Python siempre que se pasen como argumentos tipos de datos *mutables* (listas, diccionarios, conjuntos...*NO tuplas*) se pasan bajo la modalidad *por referencia*, sin embargo los tipos de datos simples se pasan *por valor*.

- **Alternativa NO Recomendable**

Python nos permite poder modificar una variable del ámbito del programa principal, dentro de una función, algo que NO es recomendable hacer, ya que se puede perder el control absoluto de la ejecución del programa y de la lógica implementada.

```
def funcion_prueba():  
    global variable_local_al_programa  
    print("Valor de la variable al ingresar a la FC: ",  
          variable_local_al_programa)  
    variable_local_al_programa = 2000  
    print("Valor de la variable al salir de la FC: ",  
          variable_local_al_programa)  
  
    variable_local_al_programa = 1000;  
    print("Valor de la variable ANTES de llamar a la FC: ",  
          variable_local_al_programa)  
    funcion_prueba()  
    print("Valor de la variable DESPUES de llamar a la FC: ",  
          variable_local_al_programa)
```

Como se puede ver...la sentencia `global variable_local_al_programa` significa que estoy “accediendo” a una variable que se llama como tal que pertenece al ámbito “general” del programa, esto no es recomendable hacerlo ya que cualquier llamada a dicha función va a modificar la variable y talvez esa no sea la intención que motiva el llamado a dicha función.

**Ejercicio para Practicar**

**Realiza una función llamada *area\_rectangulo* (*base*, *altura*) que devuelva el área del rectangulo a partir de una base y una altura.**

**Ejercicio para Practicar**

**Realiza una función *separar* (*lista*) que tome una lista de números enteros y devuelva dos listas ordenadas. La primera con los números pares y la segunda con los números impares.**

```
pares, impares = separar([6,5,2,1,7])  
print(pares)  
print(impares)
```

---

## Funciones recursivas

La recursividad es el mecanismo por el cual una función se llama a sí misma.

En definitiva, una función recursiva es aquella que se define en términos de sí misma, es decir, que el resultado de la función depende de resultados obtenidos de evaluar la misma función con otros valores.

La principal ventaja de una función recursiva es que se pueden usar para crear versión de algoritmos más claras y sencillas pero se debe tener mucho cuidado en la definición de este tipo de funciones, pues si no se hace bien, la función podría requerir de un cálculo infinito o no ser calculable.

```
def mi_funcion_recursiva():  
    mi_funcion_recursiva()  
  
mi_funcion_recursiva() #aquí llamo a la fc por primera vez
```

Como se puede ver la función se la llama desde el programa principal (línea 4) por primera vez, pero dentro de la función también se la llama, cuando se la vuelva a llamar, en ese llamado se la vuelve a llamar y así sucesivamente. Python, tiene un mecanismo de seguridad que controla esta situación por nosotros, ya que, de no ser así, podríamos llegar a consumir los recursos del sistema.

Un clásico ejemplo es el cálculo del **factorial** de un número (*El factorial de un número entero positivo se define como el producto de todos los números naturales anteriores o iguales a él*). Veamos un desarrollo en relación a esto:

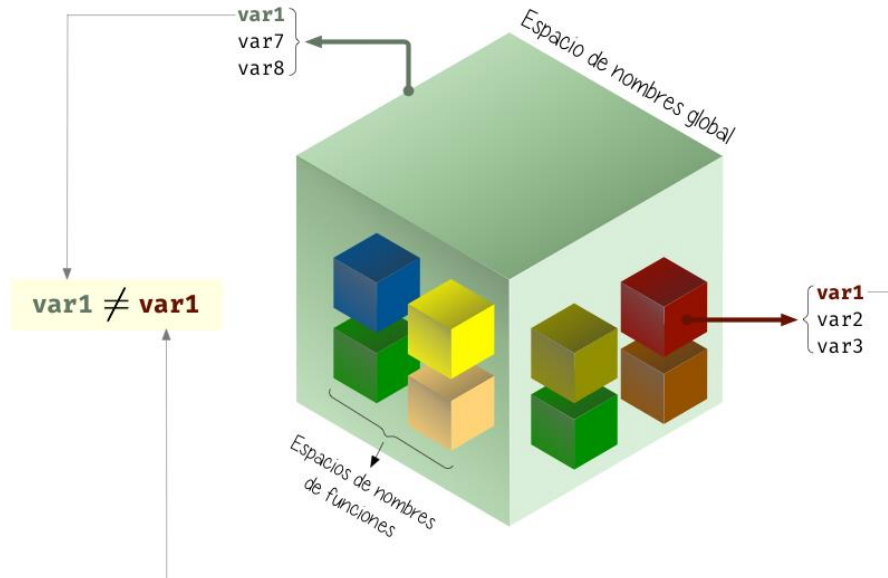
```
def factorial(n):  
    if (n==1):  
        return 1  
    return factorial(n-1)*n  
  
numero = int(input())  
calculo = factorial(numero)  
print("El Factorial de ",numero," es ", calculo)
```

### ➤ Espacios de nombres (Módulos, paquetes y namespaces)

Los **espacios de nombres** permiten definir **ámbitos** o **contextos** en los que agrupar nombres de objetos.

Los espacios de nombres proporcionan un mecanismo de empaquetamiento, de tal forma que podamos tener incluso nombres iguales que no hacen referencia al mismo objeto (siempre y cuando estén en ámbitos distintos).

Cada *función* define su propio espacio de nombres y es diferente del espacio de nombres global aplicable a todo nuestro programa.



### Acceso a variables globales

Cuando una variable se define en el **espacio de nombres global** podemos hacer uso de ella con total transparencia dentro del ámbito de las funciones del programa, por ejemplo:

```
def funcion_prueba(parametro_a):  
    print ("Valor del parametro recibido en la FC: ", parametro_a)  
    print ("Valor de una variable GLOBAL en la FC: ", variable_global_2)  
  
variable_global_1=100  
variable_global_2=200  
funcion_prueba(variable_global_1)
```

Como se puede apreciar, dentro de la función *función\_prueba*, se imprime (y se puede usar libremente) el valor que tiene una variable GLOBAL, en este caso **variable\_global\_2**. Si se intenta “modificar” dicha variable dentro de la función, NO es un error pero hay que tener cuidado ya que esa acción lo que hace es “crear” una variable con el mismo nombre pero de manera LOCAL a la función.

### Creando variables locales

En el caso de que asignemos un valor a una variable global dentro de una función, no estaremos modificando ese valor. Por el contrario, estaremos creando una *variable* en el *espacio de nombres local*:

```
def funcion_prueba(parametro_a):
    variable_global_2 = 500 #esta variable es LOCAL a la funcion, es decir
                           #que no tiene NADA que ver con la variable
                           #del mismo nombre fuera de la fc
    print ("Valor del parametro recibido en la FC: ", parametro_a)
    print ("Valor de una variable GLOBAL en la FC: ", variable_global_2)

variable_global_1=100
variable_global_2=200
funcion_prueba(variable_global_1)

print ("Variables LOCALES al programa: ", variable_global_1," ///
",variable_global_2)
```

- **Salida del Programa:**

Valor del parámetro recibido en la FC: 100  
Valor de una variable GLOBAL en la FC: 500  
Variables LOCALES al programa: 100 /// 200

Como se puede apreciar, dentro de la función se le asigna un valor a una variable (*variable\_global\_2*) que tiene el mismo nombre que otra variable del ámbito global (programa principal), pero aquí NO se está modificando el valor de la global, al contrario el intérprete detecta esta situación y lo que hace es “crear” una variable pero de manera local a la función.

### **Forzando modificación global**

Python nos permite modificar una variable definida en un espacio de nombres global dentro de una función. Para ello debemos usar el modificador **global**:

```
def funcion_prueba(parametro_a):
    global variable_global_2
    variable_global_2 = 500 #modifico el valor de la variable del programa
                           #principal
    print ("Valor del parametro recibido en la FC: ", parametro_a)
    print ("Valor de una variable GLOBAL en la FC: ", variable_global_2)

variable_global_1=100
variable_global_2=200
funcion_prueba(variable_global_1)
```

```
print ("Variables LOCALES al programa: " , variable_global_1," ///  
",variable_global_2)
```

### Salida del Programa:

Valor del parametro recibido en la FC: 100  
Valor de una variable GLOBAL en la FC: 500  
Variables LOCALES al programa: 100 /// 500

A diferencia del caso anterior, si la intención es modificar el valor de la variable global, le tenemos que indicar al intérprete que esa variable es del ámbito “global” usando la sentencia “**global**”.

**NOTA CONSEJO:** *El uso de global no se considera una buena práctica ya que puede inducir a confusión y tener efectos colaterales indeseados.*

### ➤ Módulos

Escribir pequeños trozos de código puede resultar interesante para realizar determinadas pruebas. Pero a la larga, nuestros programas tenderán a crecer y será necesario agrupar el código en unidades manejables.

Los *módulos* son simplemente ficheros de texto que contienen código Python y representan unidades con las que evitar la repetición y favorecer la reutilización.

### Importar un módulo

Para hacer uso del código de otros módulos usaremos la sentencia **import**. Esto permite importar el código y las variables de dicho módulo para que estén disponibles en nuestro programa.

La forma más sencilla de importar un módulo es `import <module>` donde **module** es el nombre de otro fichero Python, sin la extensión .py.

Supongamos que partimos del siguiente archivo (módulo): ***funciones.py***, cuyo contenido es el siguiente:

```
def suma(a,b):  
  
    return (a + b)  
  
def multiplicacion(a,b):  
  
    return (a * b)
```

Y desde otro archivo, suponiendo que sea el programa “principal”, podemos hacer uso de las funciones declaradas en el archivo anterior:

***principal.py***

```
import funciones
```

```
respuesta_suma = funciones.suma(10,20)  
print("La suma obtenida es: ", respuesta_suma)
```

```
respuesta_mult = funciones.multiplicacion(10,20)  
print("La multiplicacion obtenida es: ", respuesta_mult)
```

Al ejecutar este archivo, la salida seria:

La suma obtenida es: 30

La multiplicacion obtenida es: 200

Como podemos ver, lo que sucedió es que en la línea 1 del programa principal “importamos” todo el contenido del archivo “funciones.py” y en las 2 y 4 del código invocamos las funciones definidas en el archivo anexo para cumplir con el objetivo deseado.

Una versión más abreviada es la siguiente:

```
import funciones as f
```

```
respuesta_suma = f.suma(10,20)  
print("La suma obtenida es: ", respuesta_suma)
```

```
respuesta_mult = f.multiplicacion(10,20)  
print("La multiplicacion obtenida es: ", respuesta_mult)
```

Aquí la funcionalidad es exactamente la misma solamente que utilizamos un “alias” (es este caso f) del archivo para facilitar la escritura.

### **Importar partes de un módulo**

Es posible que NO necesitemos todo aquello que está definido en *funciones.py*. Supongamos que sólo vamos a realizar *sumas*. Para ello haremos lo siguiente:

```
from funciones import suma
```

```
respuesta_suma = suma(10,20)  
print("La suma obtenida es: ", respuesta_suma)
```

Aquí lo que estamos indicando es que vamos a “importar” desde el archivo “funciones”, solo la función “suma”.

NOTAR que en la segunda línea de código, donde invocamos la función “suma”, NO antepone la palabra “funciones.”, como en el ejemplo anterior, esto es porque al

importar solo la función suma, la misma ya forma parte del ambiente del programa principal.

Este “estilo”, tiene el inconveniente de la posible colisión de nombres, en aquellos casos en los que tuviéramos algún objeto con el mismo nombre que el objeto que estamos importando. Es decir, si el programa principal también tuviese una función suma pero “propia”, estaríamos en problema. Para solucionar esto se utiliza un “alias” de importación:

```
from funciones import suma as suma_libreria

respuesta_suma = suma_libreria(10,20)
print("La suma obtenida es: ", respuesta_suma)
```

Un ejemplo más completo con funciones importadas y funciones locales:

```
from funciones import suma as suma_libreria
def suma(a,b,c):
    return (a+b+c)

respuesta_suma = suma_libreria(10,20)
print("La suma obtenida es de la fc libreria: ", respuesta_suma)
respuesta_suma_local = suma(10,20,30)
print("La suma obtenida es de la fc libreria: ", respuesta_suma_local)
```





Los módulos son una excelente manera de ir organizando nuestro desarrollo, de manera tal de ir colocando funcionalidades en archivos “lógicamente” creados para tal fin. Por ejemplo, funciones básicas, conexiones a bases de datos locales, conexiones bases de datos remotas, procedimientos de controles de acceso, etc,etc,etc.

## Paquetes





En Python, cada uno de nuestros archivos **.py** se denominan módulos. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un **paquete**, es una carpeta que contiene archivos **.py**. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado **\_\_init\_\_.py**. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

Además permite tener una jerarquía con más de un nivel de subcarpetas anidadas.

Para ejemplificar este modelo vamos a crear un paquete llamado *librerias* que contendrá 2 módulos: *matemáticas* y *lógicas*

 <b>__pycache__</b>	29/08/2022 10:50	Carpeta de archivos	
 <b>librerias</b>	29/08/2022 11:28	Carpeta de archivos	
 <b>varios</b>	25/08/2022 22:36	Carpeta de archivos	
 <b>principal.py</b>	29/08/2022 11:26	Archivo PY	1 KB



 <code>__pycache__</code>	29/08/2022 11:25	Carpeta de archivos	
 <code>__init__.py</code>	19/07/2022 11:45	Archivo PY	0 KB
 <code>logicas.py</code>	19/07/2022 14:38	Archivo PY	1 KB
 <code>matematicas.py</code>	19/07/2022 14:38	Archivo PY	1 KB

El detalle de las funciones *matemáticas* es el mismo que en los ejemplos anteriores llamábamos *funciones*. A continuación se muestra el contenido del módulo *lógicas*:

```
def primero_mayor(a,b):
    return (a > b)
```

```
def son_iguales(a,b):
    return (a == b)
```

A continuación se muestra un ejemplo donde usamos el “paquete” *librería* que hemos creado:






```
from librerias import matematicas, logicas

resultado_suma = matematicas.suma(10,20)
resultado_multiplicacion = matematicas.multiplicacion(10,20)
print ("La suma obtenida es: ",resultado_suma)
print ("La multiplicacion obtenida es: ",resultado_multiplicacion)

print ("El primer parametro es Mayor que el Segundo? :
",logicas.primero_mayor(10,20))
print ("Ambos parametros son iguales? : ",logicas.son_iguales(10,10))
```





Otro ejemplo un poco más complejo que contiene 2 paquetes de librerías distintos con funciones que comparten su nombre. Veremos cómo se utilizan:

- Estructura de carpetas de la raíz del proyecto:





 <code>__pycache__</code>	29/08/2022 10:50	Carpeta de archivos	
 <code>librerias</code>	29/08/2022 11:28	Carpeta de archivos	
 <code>librerias_nuevas</code>	29/08/2022 14:26	Carpeta de archivos	
 <code>varios</code>	25/08/2022 22:36	Carpeta de archivos	
 <code>principal.py</code>	29/08/2022 14:37	Archivo PY	1 KB

- Estructura de la carpeta *librerias*:

---

 _pycache_	29/08/2022 11:36	Carpeta de archivos	
 _init_.py	19/07/2022 11:45	Archivo PY	0 KB
 logicas.py	29/08/2022 11:33	Archivo PY	1 KB
 matematicas.py	29/08/2022 11:36	Archivo PY	1 KB

- Estructura de la carpeta **librerías\_nuevas**:

 _pycache_	29/08/2022 14:26	Carpeta de archivos	
 _init_.py	19/07/2022 11:45	Archivo PY	0 KB
 logicas.py	29/08/2022 14:23	Archivo PY	1 KB
 matematicas.py	29/08/2022 14:25	Archivo PY	1 KB

Veamos el contenido de los archivos:

**librerías:**

- matemáticas.py

```
def suma(a,b):  
    return (a + b)
```

```
def multiplicacion(a,b):  
    return (a * b)
```

- logicas.py

```
def primero_mayor(a,b):  
    return (a > b)
```

```
def son_iguales(a,b):  
    return (a == b)
```

**librerías\_nuevas:**

- matemáticas.py

```
def suma(a,b,c):  
    return (a + b + c)
```

```
def division(a,b):  
    if b>0:  
        return (a / b)  
    else:  
        return "ERROR DIVISION POR CERO"
```

- logicas.py

---

```
def primero_menor_o_igual(a,b):  
    return (a < b)
```

```
def son_distintos(a,b):  
    return (a != b)
```

**PROGRAMA PRINCIPAL:**

```
from librerias import matematicas as mate_ori, logicas as log_ori  
from librerias_nuevas import matematicas as mate_nueva, logicas as log_nueva  
  
resultado_suma_ori = mate_ori.suma(10,20)  
resultado_primer_mayor_ori = log_ori.primer_mayor(10,20)  
print ("*****RESULTADOS DE LIBRERIAS ORIGINALES*****")  
print ("La suma obtenida es: ",resultado_suma_ori)  
print ("El primer parametro es mayor que el segundo??: ",resultado_primer_mayor_ori)  
  
resultado_suma_nueva = mate_nueva.suma(10,20,30)  
resultado_primer_menor_nueva = log_nueva.primer_menor_o_igual(10,20)  
print ("*****RESULTADOS DE LIBRERIAS NUEVAS*****")  
print ("La suma obtenida es: ",resultado_suma_nueva)  
print ("El primer parametro es menor o igual que el segundo??:  
",resultado_primer_menor_nueva)
```

**Salida:**

```
*****RESULTADOS DE LIBRERIAS ORIGINALES*****  
La suma obtenida es: 30  
El primer parametro es mayor que el segundo??: False  
*****RESULTADOS DE LIBRERIAS NUEVAS*****  
La suma obtenida es: 60  
El primer parametro es menor o igual que el segundo??: True
```

**NOTA:** tal como lo había anticipado, en este ejemplo importamos 2 paquetes, *librerias* y *librerías\_nuevas*, ambas tienen 2 módulos (podrían tener más tranquilamente si fuese necesario) con los mismos nombres (*logicas* y *matematicas*) y cada uno de ellos tienen funciones que son utilizadas dentro del programa principal de manera arbitraria y libre según se necesita, accediendo a ellas a través del *espacio de nombre correcto* (*namespaces*).

**ACLARACION GENERAL:** La importación de módulos y paquetes debe realizarse al comienzo del documento. Y como buena práctica de desarrollo en PYTHON, primero deben importarse los *módulos propios de Python*. Luego, los *módulos de terceros* y finalmente, los *módulos propios de la aplicación*.

Entre cada bloque de *imports*, debe dejarse una línea en blanco, para poder interpretar rápidamente el lugar desde donde esta importando.

## ➤ Manejo de Errores - Excepciones

A medida que vayamos avanzando en nuestro desarrollo nos daremos cuenta que seguramente vamos a cometer errores (y muchos más de los que nos imaginamos), lo más importante es “saber cómo resolverlos o tratarlos”.

Entre los errores comunes que podemos cometer se encuentran los:

- **ERRORES SINTÁCTICOS:** Se producen cuando existe un problema de sintaxis en nuestros comandos, como escribir mal un comando, y Python nos alerta de esto con el mensaje de error *SyntaxError: invalid syntax*.
- **ERRORES EN TIEMPO DE EJECUCIÓN:** Ocurren mientras el programa se está ejecutando y algo inesperadamente ocurre mal. Generalmente Python nos informa ese tipo de error como por ejemplo una recursión infinita causando el error *maximum recursion Depth exceded*.
- **ERRORES SEMÁNTICOS:** Se dan cuando el programa compila y se ejecuta normalmente, pero no hace lo que se pretendía que hiciera y Python en este caso no nos va informar donde está el error. Y para eso debemos valernos de la depuración.

Los errores de sintaxis siempre son detectados por el intérprete de Python antes de ejecutar el programa, por lo que son errores relativamente simples de resolver.

Estos errores de tipografía nos pueden indicar si el nombre de algún comando es incorrecto, nombres de variables incorrectas, si falta algún paréntesis, palabras claves mal escritas, etc.

Nosotros nos encargaremos de controlar los errores “en tiempo de ejecución”, los cuales producen *excepciones*. Una *excepción* es el bloque de código que se lanza cuando se produce un error en la ejecución de un programa. Para evitar que Python nos indique de manera “básica” estos errores, por ejemplo:

```
resultado = 10/0  
print (resultado)
```

*Se produjo una excepción: ZeroDivisionError  
division by zero*

Aquí el intérprete nos indica que se ha producido una división por cero y por ende no nos “ejecuta” el programa, forzando su detención. Si queremos evitar esta “detención”, tenemos que capturar esos mensajes y tratarlos como nosotros queramos:

Para manejar (capturar) las excepciones podemos usar un bloque de código con las palabras reservadas *try* and *except*:

**TRY:** En el *try* se coloca código que esperamos que pueda lanzar algún error.

**EXCEPT:** En el except se maneja el error, es decir, si ocurre un error dentro del bloque de código del try, se deja de ejecutar el código del try y se ejecuta lo que se haya definido en el Except.

```
try:

    resultado = 10/0
    print (resultado)

except:
    print("Se produjo una division por cero. Verifique!!")
```

Aquel código que se encuentre dentro del bloque *try* se ejecutará normalmente siempre y cuando no haya un error. Si se produce una excepción, ésta será capturada por el bloque *except*, ejecutándose el código que contiene. De esta manera nuestro programa NO se detiene de manera abrupta, de hecho se ejecuta sin problemas y nos indica el mensaje que definimos en el bloque de excepción (except).

### Cláusula ELSE

En Python, también puede usar la cláusula *else* en el bloque *try-except* que debe estar presente después de todas las cláusulas *except*. El código ingresa al bloque *else* solo si la cláusula *try* no genera una excepción.

```
try:
    #Nuestro codigo normal
except:
    #Codigo que se ejecuta si hay error
else:
    #Codigo que se ejecuta si NO hay error
```

### Cláusula FINALLY

Python proporciona una palabra clave **finally**, que siempre se ejecuta después de **try** y **except**.

Este código **siempre** se ejecuta después de los otros bloques, incluso si hubo una excepción no detectada o una declaración de retorno en uno de los otros bloques.

```
try:
    #Nuestro codigo normal
except:
    #Codigo que se ejecuta si hay error
else:
    #Codigo que se ejecuta si NO hay error
finally:
    #Codigo que se va a ejecutar SIEMPRE, independientemente si hubo o no
```

---

## #errores

### Cláusula RAISE

Podemos usar **raise** para lanzar una excepción si ocurre una condición. La declaración se puede complementar con una excepción personalizada.

```
raise Exception('Mensaje')
raise ValueError('Mensaje')
raise TypeError('Mensaje')
raise NameError('Mensaje')
```

- Veamos un ejemplo un poco genérico:

```
try:
    x = input("Ingrese el dividendo: ")
    y = input("Ingrese el divisor: ")
    z = int(x) / int(y)
except:
    print("ERROR DE ALGO (vaya a saber que !!)")

else:
    print("El resultado de la division es: ", z)
finally:
    print("***PROGRAMA FINALIZADO**")
```

- Veamos un ejemplo completo con todas estas sentencias, pero identificando algunos tipos de excepciones:

```
try:
    x = int(input("Ingrese el dividendo: "))
    y = int(input("Ingrese el divisor: "))
    z = x / y
    if z < 0:
        raise ValueError("No deben haber numero negativos!!")
except (ZeroDivisionError):
    print("Division por CERO!!")
    try:
        v = 1 / 'a'
    except TypeError:
        print("Error de tipo de datos")
except ValueError as error:
    print("ERROR: ",error)
else:
    print("El resultado de la division es: ", z)
```

---

```
finally:  
    print("***PROGRAMA FINALIZADO***")
```

## Tipos de excepciones

Los principales excepciones definidas en Python son:

- ***TypeError***: Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.
- ***ZeroDivisionError***: Ocurre cuando se intenta dividir por cero.
- ***OverflowError***: Ocurre cuando un cálculo excede el límite para un tipo de dato numérico.
- ***IndexError***: Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.
- ***KeyError***: Ocurre cuando se intenta acceder a un diccionario con una clave que no existe.
- ***FileNotFoundError***: Ocurre cuando se intenta acceder a un fichero que no existe en la ruta indicada.
- ***ImportError***: Ocurre cuando falla la importación de un módulo.

Para ver el listado completo de todas las excepciones de Python, se recomienda visitar:  
<https://docs.python.org/es/3/library/exceptions.html>