😎 Funciones en Python

En Python, las funciones se pueden definir en cualquier punto de un programa. La primera línea de la definición de una función contiene:

- **La palabra reservada def**
- **El nombre de la función (la guía de estilo de Python recomienda escribir todos los caracteres en minúsculas separando las palabras por guiones bajos)**
- **Paréntesis (que pueden incluir los argumentos de la función) y dos puntos (:)**

- Las instrucciones que forman la función se escriben con sangría con respecto a la primera línea.
- Se puede indicar el final de la función con la palabra reservada **return**, aunque no es obligatorio.
- Para poder utilizar una función en un programa se tiene que haber definido antes.

El ejemplo siguiente muestra un programa que contiene una función y el resultado de la ejecución de ese programa.

Definición de una función:

In [1]:
```
def mensajes():
    print("Este es el primer ejemplo")

mensajes()
print("Programa terminado.")
```

```
Este es el primer ejemplo
Programa terminado.
```

El ejemplo siguiente muestra un programa incorrecto que intenta utilizar una función antes de haberla definido:

In [2]:
```
mens()
print("Programa terminado.")
def mens():
    print("Este es el primer ejemplo")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [2], in <cell line: 1>()
----> 1 mens()
      2 print("Programa terminado.")
      3 def mens():

NameError: name 'mens' is not defined
```

Ámbitos de las variables en Python

Python distingue tres tipos de ámbitos:

Las variables locales y dos tipos de variables "libres" (globales y no locales):

- **variables locales: las que pertenecen al ámbito de la función (y que pueden ser accesibles por niveles inferiores)**
- **variables globales: las que pertenecen al ámbito del programa principal.**
- **variables no locales: las que pertenecen a un ámbito superior al de la función, pero que no son globales.**

- Si el programa contiene solamente funciones que no contienen a su vez funciones, todas las variables libres son variables **globales**.
- Pero si el programa contiene una función que a su vez contiene una función, las variables libres de esas "sub-funciones" pueden ser **globales** (si pertenecen al programa principal) o **no locales** (si pertenecen a la función).
- Para identificar explícitamente las variables globales y no locales, se utilizan las palabras reservadas **global** y **nonlocal**.
- Las variables locales no necesitan identificación.

Variables locales

Si no se han declarado como globales o no locales, las variables a las que se asigna valor en una función se consideran variables locales, es decir, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre.

Ejemplo:

In [3]:
```
def funcion1():
    a = 2 #variable local en la función
    print(a)
a = 5
print(a)
funcion1()
print(a)
```

```
5
2
5
```

Las variables locales sólo existen en la propia función y no son accesibles desde niveles superiores.

Ejemplo:

In [4]:
```
def funcion2():
    b = 2
    print(b)

funcion()
print(b)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [4], in <cell line: 5>()
      2     b = 2
      3     print(b)
----> 5 funcion()
      6 print(b)

NameError: name 'funcion' is not defined
```

Si en el interior de una función se asigna valor a una variable que no se ha declarado como global o no local, esa variable es local a todos los efectos.

Por ello el siguiente programa da error:

In [5]:
```
def funcion3():
    print(c)
    c = 2
    print(c)

c = 5
funcion3()
print(c)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [5], in <cell line: 7>()
      4     print(c)
      6 c = 5
----> 7 funcion3()
      8 print(c)

Input In [5], in funcion3()
      1 def funcion3():
----> 2     print(c)
      3     c = 2
      4     print(c)

UnboundLocalError: local variable 'c' referenced before assignment
```

El motivo es que en la función se asigna valor a la variable "c" (en la segunda instrucción) por lo tanto Python la considera variable local. Como la primera instrucción quiere imprimir el valor de "c", pero a esa variable todavía no se le ha dado valor en la función (el valor de la variable "c" del programa principal no cuenta pues se trata de variables distintas, aunque se llamen igual), se produce el mensaje de error.

Variables libres globales o no locales

- **Si a una variable no se le asigna valor en una función, Python la considera libre y busca su valor en los niveles superiores de esa función, empezando por el inmediatamente superior y continuando hasta el programa principal.**
- **Si a la variable se le asigna valor en algún nivel intermedio la variable se considera nonlocal.**
- **Si se le asigna en el programa principal la variable se considera global.**

En el ejemplo siguiente, la variable libre "d" de la función funcion4() se considera global porque obtiene su valor del programa principal:

In [6]:
```
def funcion4():
    print(d)

d = 5
funcion4()
print(d)
```

```
5
5
```

La última instrucción del programa escribe de nuevo el valor de "d", que sigue siendo 5 pues la función no ha modificado el valor de la variable. Al ser global es "visible" por cualquier función que puede utilizar su valor. Entonces

In [7]:
```
x = 0 # Es global por estar fuera de las funciones

def funcion5():
    print(x)

funcion5()
```

```
0
```

Pero existe un potencial conflicto. Si una función intenta cambiar el valor de x:

In [8]:
```
x = 0

def funcion6():
    x = 1
    print(x)

funcion6()
print(x)
```

```
1
0
```

Debido a que aparece una asignación dentro de la función la variable x se considerará local. Se crea por tanto una nueva variable x específica para la función que impedirá acceder a la variable global del mismo nombre. Aunque print(x) emitirá 1, es un valor local. Cuando la función termine, dejará de existir. La x global seguirá en 0.

Se nos puede plantear exactamente el mismo problema ya visto si la modificamos dentro de la función, pero la intentamos usar antes de haberla modificado, así:

In [9]:
```
x = 0

def funcion7():
    print(x)
    x = 1

funcion7()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [9], in <cell line: 7>()
      4     print(x)
      5     x = 1
----> 7 funcion7()

Input In [9], in funcion7()
      3 def funcion7():
----> 4     print(x)
      5     x = 1

UnboundLocalError: local variable 'x' referenced before assignment
```

Antes de intentar ejecutar esta función Python ya "ha visto" que en alguna línea se asigna un valor a x, y considerará x local. Por lo tanto el primer print(x) intentará imprimir la variable local y como aún no tiene un valor asignado y se producirá el error.
Para resolver este problema existe la palabra global que permite especificarle a Python que, aunque vea una asignación dentro de la función, no cree una nueva variable local, sino que haga uso de la global.

In [10]:
```
x = 0
print('x antes de test()',x)

def test():
    global x
    print('x dentro de test()',x)
    x = 1

test()
print('x luego de test()',x)
```

```
x antes de test() 0
x dentro de test() 0
x luego de test() 1
```

En este caso, las referencias que ocurren a x dentro de la función acceden a la x global, por lo que el código no sólo no da error, sino que cuando la función termine de ejecutarse el valor de la x global habrá cambiado.

nonlocal existe para un caso muy particular y es el de funciones anidadas que necesitan acceder a variables que, ni son locales de la propia función anidada, ni son globales, sino que son variables de la función dentro de la cual están anidadas.

In [11]:
```
x=0

def funcion8():
    x = 1

    def anidada():
        x = 2
        print('x dentro de anidada()',x)

    anidada()
    print('x dentro de funcion8()',x)

print('x antes de funcion8()',x)
funcion8()
print('x luego de funcion8()',x)
```

```
x antes de funcion8() 0
x dentro de anidada() 2
x dentro de funcion8() 1
x luego de funcion8() 0
```

1. Al ejecutar la función funcion8(), se crea una nueva variable local.
2. A esa x se le da el valor 1, la x global sigue en 0.
3. Luego se define una función anidada, y se invoca.
4. Dentro de ella hay una asignación x=2, pero como no se ha mencioado que la x sea global, nuevamente se creará una local y se le asigna a x el valor 2.
5. La global sigue siendo 0.
6. Así que se emitirá un 2 y la global seguirá en 0

In [12]:
```
x = 0

def funcion9():
    x = 1
    print('x dentro de funcion9()',x)
    def anidada():
        global x
        x = 2
        print('x dentro de anidada()',x)
    anidada()
    print('x dentro de funcion9()',x)

print('x antes de funcion9()',x)
funcion9()
print('x luego de funcion9()',x)
```

```
x antes de funcion9() 0
x dentro de funcion9() 1
x dentro de anidada() 2
x dentro de funcion9() 1
x luego de funcion9() 2
```

En este ejemplo se crea x=2 actúa sobre la global. Y si quisiéramos desde anidada() modificar la x local de funcion10()?. Para esto es la palabra nonlocal:

In [13]:
```
x = 0

def funcion10():
    x = 1
    print('x dentro de funcion10()',x)

    def anidada():
        nonlocal x
        x = 2
        print('x dentro de anidada()',x)

    anidada()
    print('x dentro de funcion10()',x)

print('x antes de funcion10()',x)
funcion10()
print('x luego de funcion10()',x)
```

```
x antes de funcion10() 0
x dentro de funcion10() 1
x dentro de anidada() 2
x dentro de funcion10() 2
x luego de funcion10() 0
```

Cuando Python ve nonlocal x, sabe que aunque esa función intente asignar valores a una x, no debe crear una variable local para ello. Deberá usar la x que esté accesible dentro de su alcance (que será la definida en la asignación x=1)

Otro ejemplo, la variable libre "a" de la función sub_funcion() se considera no local porque obtiene su valor de una función intermedia:

In [14]:
```
def funcion11():

    def sub_funcion():
        print('a dentro de sub_funcion()',a)

    a = 3
    sub_funcion()
    print('a dentro de funcion11()',a)

a = 4
funcion11()
print('a luego funcion11()',a)
```

```
a dentro de sub_funcion() 3
a dentro de funcion11() 3
a luego funcion11() 4
```

1. En la función sub_funcion(), la variable "a" es libre pues no se le asigna valor.
2. Se busca su valor en los niveles superiores, por orden. En este caso, el nivel inmediatamente superior es la función funcion11().
3. Como en ella hay una variable local que también se llama "a", Python toma de ella el valor (en este caso, 3) y lo emite.
4. Para la función sub_funcion(), la variable "a" es una variable nonlocal, porque su valor proviene de una función intermedia.
5. Si una variable que Python considera libre (porque no se le asigna valor en la función), no se le asigna valor en niveles superiores, Python dará un mensaje de error.

Entonces...

Si queremos asignar valor a una variable en una subrutina, pero no queremos que Python la considere local, debemos declararla en la función como global o nonlocal. En el ejemplo siguiente la variable se declara como global, para que su valor sea el del programa principal:

In [15]:
```
def subrutina():
    global a
    print('a dentro de subrutina antes de la asignación()',a)
    a = 1
    print('a dentro de subrutina luego de la asignación()',a)

a = 5
print('a antes de subrutina()',a)
subrutina()
print('a luego de subrutina()',a)
```

```
a antes de subrutina() 5
a dentro de subrutina antes de la asignación() 5
a dentro de subrutina luego de la asignación() 1
a luego de subrutina() 1
```

En el ejemplo siguiente la variable se declara como nonlocal, para que su valor sea el de la función intermedia:

In [16]:
```
def funcion12():
    def sub_funcion():
        nonlocal a
        print('a dentro de sub_funcion()',a)
        a = 1
    a = 3
    sub_funcion()
    print('a dentro de funcion12()',a)

a = 4
print('a antes de funcion12()',a)
funcion12()
print('a luego de funcion12()',a)
```

```
a antes de funcion12() 4
a dentro de sub_funcion() 3
a dentro de funcion12() 1
a luego de funcion12() 4
```

Si a una variable declarada global o nonlocal en una función no se le asigna valor en el nivel superior correspondiente, Python dará un error de sintaxis.

Ejemplo de variable declarada nonlocal no definida

In [17]:
```
def funcion13():

    def sub_funcion():
        nonlocal a
        print('a dentro de sub_funcion()',a)
        a = 1

    sub_funcion()
    print('a dentro de funcion13()',a)

a = 4
print('a antes de funcion13()',a)
funcion13()
print('a luego de funcion13()',a)
```

```
  Input In [17]
    nonlocal a
    ^
SyntaxError: no binding for nonlocal 'a' found
```

Funciones nativas locals() y globals()

locals():

Devuelve un diccionario que contiene las variables locales del ámbito actual.

In [18]:
```
def demo1():
    print("Aquí no hay variable presente: ", locals())

def demo2():
    name = "Python"
    print("Aquí hay variable presente: ", locals())

demo1()
demo2()
```

```
Aquí no hay variable presente:  {}
Aquí hay variable presente:  {'name': 'Python'}
```

globals():

Devuelve el diccionario que contiene las variables globales del ámbito actual.

In [19]:
```
def demo1():
    print("Aquí no hay variable presente: \n ", locals())
demo1()
```

```
Aquí no hay variable presente:
  {}
```

In [20]:
```
def demo2():
    name = "Python"
    print("Aquí hay variable presente: \n", locals())
demo2()
```

```
Aquí hay variable presente:
 {'name': 'Python'}
```

In [ ]:
```
print("Esto está usando globals() : \n ", globals()) # ejecutar ésta instrucción
```

callable()

La función callable() indica si un objecto puede ser llamado.

In [22]:
```
def demo1():
    name = "Python"

demo1()
callable(demo1)
```

Out[22]: True

Argumentos y retorno de valores

Las funciones en Python admiten argumentos en su llamada y permiten retornar valores. Esta posibilidad permite crear funciones más útiles y reutilizables.

Ejemplo de función sin parámetros y con emisión de un mensaje:

In [23]:
```
def media():
    media = (a + b) / 2
    print(f"La media de {a} y {b} es: {media}")

a = 3
b = 5
media()
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Este tipo de función es muy difícil de reutilizar. Para evitar ese problema, las funciones admiten argumentos, es decir, permiten que se les envíen valores con los que trabajar. De esa manera, las funciones se pueden reutilizar más fácilmente, como muestra el ejemplo siguiente:

In [24]:
```
def media(x, y):
    media = (x + y) / 2
    print(f"La media de {x} y {y} es: {media}")

a = 3
b = 5
media(a, b)
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Esta función puede ser utilizada con más flexibilidad que la del ejemplo anterior. Pero aún tiene un inconveniente, como las variables locales de una función son inaccesibles desde los niveles superiores, el programa principal no puede utilizar la variable "media" calculada por la función y tiene que ser la función la que calcule y emita el resultado. Para evitar ese problema, las funciones pueden retornar valores, es decir, pueden enviar valores al programa o función de nivel superior:

In [25]:
```
def calcula_media(x, y):
    resultado = (x + y) / 2
    return resultado

a = 3
b = 5
media = calcula_media(a, b)
print(f"La media de {a} y {b} es: {media}")
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Esta función puede ser utilizada con más flexibilidad que la del ejemplo anterior. Pero esta función tiene todavía un inconveniente y es que sólo calcula la media de dos valores. Sería más interesante que la función calculara la media de cualquier cantidad de valores. Para evitar ese problema, las funciones pueden admitir una cantidad indeterminada de valores, como muestra el ejemplo siguiente:

In [26]:
```
def calcula_media(*args):
    total = 0
    for i in args:
        total += i
    resultado = total / len(args)
    return resultado

a, b, c, d = 3, 5, 10, 9
media = calcula_media(a, b, c, d)
print(f"La media de {a}, {b} y {c} es: {media} y le agregue {d}")
print("Programa terminado")
```

```
La media de 3, 5 y 10 es: 6.75 y le agregue 9
Programa terminado
```

Esta última función puede ser utilizada aún con más flexibilidad que las anteriores, puesto que se puede elegir de cuántos valores hacer la media y qué hacer con el valor calculado por la función. Las funciones pueden devolver varios valores simultáneamente, como muestra el siguiente ejemplo:

In [27]:
```
def calcula_media_desviacion(*args):
    total = 0
    for i in args:
        total += i

    media = total / len(args)

    total = 0
    for i in args:
        total += (i - media) ** 2

    desviacion = (total / len(args)) ** 0.5

    return media, desviacion

a, b, c, d = 3, 5, 10, 12
media, desviacion_tipica = calcula_media_desviacion(a, b, c, d)
print(f"Datos: {a} {b} {c} {d}")
print(f"Media: {media}")
print(f"Desviación típica: {desviacion_tipica}")
print("Programa terminado")
```

```
Datos: 3 5 10 12
Media: 7.5
Desviación típica: 3.640054944640259
Programa terminado
```

***args y **kwargs**

No es obligatorio usar los nombres **args** o **kwargs**, pueden reemplazarse con nombres descriptivos.

args representa un conjunto arbitrario de argumentos posicionales. Lo importante es que use el operador de desempaquetado **(*)**.Tanto **list** como **tuple** admiten el corte y la iteración, pero se diferencian en que list es mutable y tuple inmutable.

In [28]:
```
def saludo(tipoSaludo, *amigos):
    listaDeAmigos = ""
    print(type(amigos))
    for amigo in amigos:
        listaDeAmigos = listaDeAmigos + ', ' + amigo
    print(tipoSaludo + listaDeAmigos)

print("Ejemplo con *args\n")
saludo("Hola", "Diego", "Juan", "...a todos")
```

```
Ejemplo con *args

<class 'tuple'>
Hola, Diego, Juan, ...a todos
```

kwargs es lo mismo que args pero con argumentos keyword, o nombre. Lo importante es el uso del operador de desempaquetado **(**)**. Hay que tener en cuenta que si se itera por un **dict** y se desea retornar el valor se debe usar **.values()**

In [29]:
```
def argsConClaveValor(**clave_valor_args):
    print(type(clave_valor_args))
    for nombre, valor in clave_valor_args.items():
        print(nombre + ': ' + valor)

print("Ejemplo con **kwargs\n")
argsConClaveValor(edad='32', profesion="Ingeniero", nacionalidad="Argentino")
```

```
Ejemplo con **kwargs

<class 'dict'>
edad: 32
profesion: Ingeniero
nacionalidad: Argentino
```

¿Qué sucede si desea crear una función que tome un número variable de argumentos posicionales y con nombre?. En este caso el orden cuenta. Del mismo modo que los argumentos no predeterminados tienen que preceder a los argumentos predeterminados, también ***args** debe aparecer antes ****kwargs**. El orden correcto es:

1. Argumentos estándar
2. *args argumentos
3. **kwargs argumentos

Por ejemplo, esta definición de función es correcta:

In [30]:
```
def funcion(a, b, *args, **kwargs):
    pass

def funcion():
    print("hola")

funcion()
```

```
hola
```

Variables globales y objetos mutables e inmutables

En Python dependiendo de si a la función se le envía como parámetro un objeto mutable o inmutable, la función podrá modificar o no al objeto. En los dos siguientes ejemplos, el parámetro de la función ("b") se llama igual que una de las dos variables del programa principal. En los dos ejemplos se llama dos veces a la función, enviando cada vez una de las dos variables ("a" y "b").

Ejemplo de conflicto entre nombre de parámetro y nombre de variable global. Objeto mutable.

Como en este caso las variables son **listas** (objetos mutables), la función modifica la lista que se envía como argumento:

> primero se modifica la lista "a" y a continuación la lista "b". La lista modificada no depende del nombre del parámetro en la función (que es "b"), sino de la variable enviada como argumento ("a" o "b")

In [31]:
```
def cambia(b):
    b += [5]

a, b = [3], [4]
print(f"Al principio : a = {a} b = {b}")
cambia(a)
print(f"Después de cambia(a): a = {a} b = {b}")
cambia(b)
print(f"Después de cambia(b): a = {a} b = {b}")
print("Programa terminado")
```

```
Al principio : a = [3] b = [4]
Después de cambia(a): a = [3, 5] b = [4]
Después de cambia(b): a = [3, 5] b = [4, 5]
Programa terminado
```

Ejemplo de conflicto entre nombre de parámetro y nombre de variable global. Objeto inmutable

Como en este caso las variables son números enteros (objetos inmutables), la función no puede modificar los números que se envían como argumentos, ni la variable "a" ni la variable "b".

In [32]:
```
def cambia(b):
    b += 1

a, b = 3, 4
print(f"Al principio : a = {a} b = {b}")
cambia(a)
print(f"Después de cambia(a): a = {a} b = {b}")
cambia(b)
print(f"Después de cambia(b): a = {a} b = {b}")
print("Programa terminado")
```

```
Al principio : a = 3 b = 4
Después de cambia(a): a = 3 b = 4
Después de cambia(b): a = 3 b = 4
Programa terminado
```

Paso por valor o paso por referencia En los lenguajes en los que las variables son "cajas" en las que se guardan valores, cuando se envía una variable como argumento en una llamada a una función suelen existir dos posibilidades:

- **paso por valor**: se envía simplemente el valor de la variable, en cuyo caso la función no puede modificar la variable, pues la función sólo conoce su valor.
- **paso por dirección**: se envía la dirección de memoria de la variable, en cuyo caso la función sí que puede modificar la variable.

En Python no se hace ni una cosa ni otra. En Python cuando se envía una variable como argumento en una llamada a una función lo que se envía es la referencia al objeto al que hace referencia la variable. Dependiendo de si el objeto es mutable o inmutable, la función podrá modificar o no el objeto.

Ejemplo de paso de variable (objeto inmutable)

In [33]:
```
def aumenta(x):
    print(id(x))
    x += 1
    print(id(x))
    return x

a = 3
print(id(3), id(4))
print(id(a))
print(aumenta(a))
print(a)
print(id(a))
```

```
2355688073584 2355688073616
2355688073584
2355688073584
2355688073616
4
3
2355688073584
```

Ejemplo de paso de variable (objeto mutable)

In [34]:
```
def aumenta(x):
    print(id(x))
    x += [1]
    print(id(x))
    return x

a = [3]
print(id(a))
print(aumenta(a))
print(a)
print(id(a))
```

```
2355787551872
2355787551872
2355787551872
[3, 1]
[3, 1]
2355787551872
```

Funciones anónimas lambda

Una función anónima, como su nombre indica es una función sin nombre. En Python podemos ejecutar una función sin definirla con def. De hecho son similares pero con una diferencia fundamental: El contenido de una función **lambda** debe ser una única expresión en lugar de un bloque de acciones. Por lo tanto se puede decir que, mientras las funciones anónimas lambda sirven para realizar funciones simples, las funciones definidas con def sirven para manejar tareas más extensas. Ejemplo:

Función que calcule el doble de un valor:

In [35]:
```
def doble(num):
    resultado = num*2
    return resultado

doble(2)
```

Out[35]: 4

Simplificamos un poco:

In [36]:
```
def doble (num):
    return num*2
doble(2)
```

Out[36]: 4

Simplificamos más escribiéndola en una sola línea:

In [37]:
```
def doble (num):
    return num*2
doble(2)
```

Out[37]: 4

La convertimos en una función anónima guardando en una variable el resultado y utilizarla tal como haríamos con una función normal:

In [38]:
```
doble= lambda num: num*2
doble(2)
```

Out[38]: 4

O sea que:

Función anónima

funcion = lambda argumentos: resultado

Es equivalente a:

def funcion(argumentos): return resultado

Diferencias entre las funciones lambda y las definidas con def

Aunque aparentemente se ha obtenido el mismo resultado existen ciertas diferencias entre ambos métodos que es necesario tener en cuenta.

- Al utilizar la palabra clave lambda se crea un objeto función sin crearse al mismo tiempo un nombre en el espacio de nombres. Nombre que sí se crea al definir la función con def.
- Las funciones lambda se crean en una única línea de código, por lo que son adecuadas cuando se desea minimizar el número de estas.
- Las funciones lambda son generalmente menos legibles que las tradicionales.
- En el caso de que se desee una función lambda es necesario asignarla a una variable, porque, si no es así, al carecer de identificador, solamente se podrá utilizar en la línea donde se defina.