Modulo 4: Funciones

¿Por qué necesitamos funciones?

Hasta ahorita has implementado varias veces el uso de funciones, pero solo se han visto algunas de sus ventajas. Solo se han invocado funciones para utilizarlas como herramientas, con el fin de hacer la vida más fácil, y para simplificar tareas tediosas y repetitivas.

Cuando se desea mostrar o imprimir algo en consola se utiliza print(). Cuando se desea leer el valor de una variable se emplea input(), combinados posiblemente con int() o float().

También se ha hecho uso de algunos métodos, los cuales también son funciones, pero declarados de una manera muy específica.

Ahora aprenderás a escribir tus propias funciones, y como utilizarlas. Escribiremos varias de ellas juntos, desde muy sencillas hasta algo complejas. Se requerirá de tu concentración y atención.

Muy a menudo ocurre que un cierto fragmento de código se repite muchas veces en un programa. Se repite de manera literal o, con algunas modificaciones menores, empleando algunas otras variables dentro del programa. También ocurre que un programador ha comenzado a copiar y pegar ciertas partes del código en más de una ocasión en el mismo programa.

Puede ser muy frustrante percatarse de repente que existe un error en el código copiado. El programador tendrá que escarbar bastante para encontrar todos los lugares en el código donde hay que corregir el error. Además, existe un gran riesgo de que las correcciones produzcan errores adicionales.

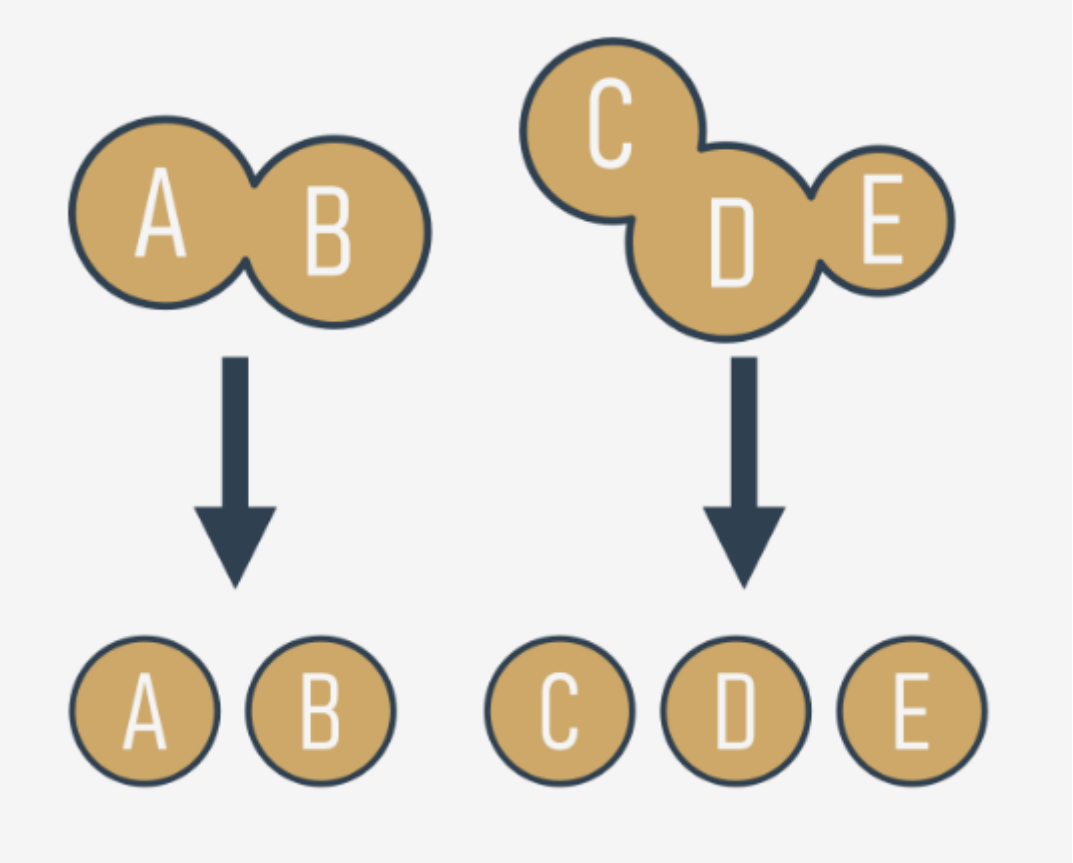
Definamos la primer condición por la cual es una buena idea comenzar a escribir funciones propias: si un fragmento de código comienza a aparecer en más de una ocasión, considera la posibilidad de aislarlo en la forma de una función invocando la función desde el lugar en el que originalmente se encontraba.

Puede suceder que el algoritmo que se desea implementar sea tan complejo que el código comience a crecer de manera incontrolada y, de repente, ya no se puede navegar por él tan fácilmente.

Se puede intentar solucionar este problema comentando el código, pero pronto te darás cuenta que esto empeorará la situación - demasiados comentarios hacen que el código sea más difícil de leer y entender. Algunos dicen que una función bien escrita debe ser comprensible con tan solo una mirada.

Un buen desarrollador divide el código (o mejor dicho: el problema) en piezas aisladas, y codifica cada una de ellas en la forma de una función.

Esto simplifica considerablemente el trabajo del programa, debido a que cada pieza se codifica por separado y consecuentemente se prueba por separado. A este proceso se le llama comúnmente descomposición.

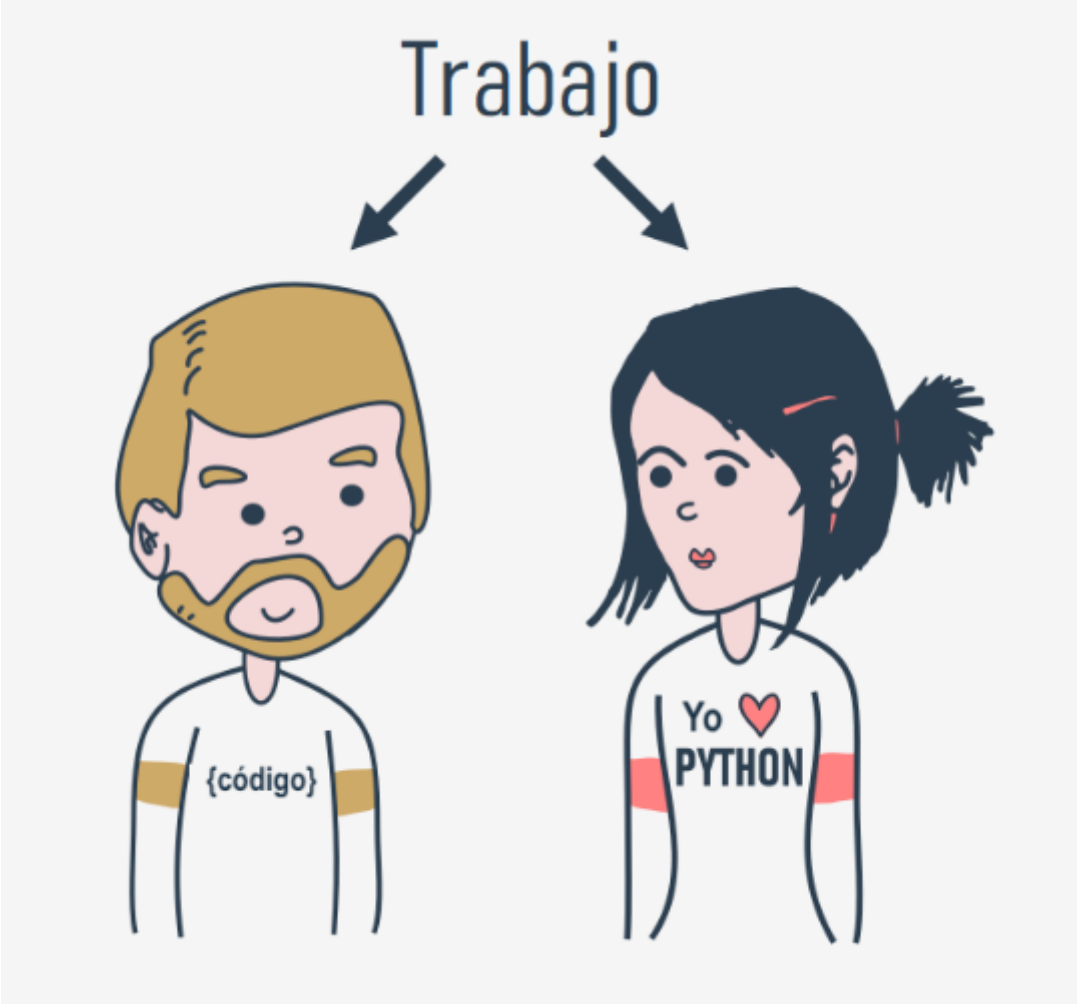


Existe una segunda condición: si un fragmento de código se hace tan extenso que leerlo o entenderlo se hace complicado, considera dividirlo pequeños problemas por separado e implementa cada uno de ellos como una función independiente.

Esta descomposición continúa hasta que se obtiene un conjunto de funciones cortas, fáciles de comprender y probar.

Descomposición

Es muy común que un programa sea tan largo y complejo que no puede ser asignado a un solo desarrollador, y en su lugar un equipo de desarrolladores trabajarán en el. El problema, debe ser dividido entre varios desarrolladores de una manera en que se pueda asegurar su eficiencia y cooperación.



Es inconcebible que más de un programador deba escribir el mismo código al mismo tiempo, por lo tanto, el trabajo debe de ser dividido entre todos los miembros del equipo.

Este tipo de descomposición tiene diferentes propósitos, no solo se trata de compartir el trabajo, sino también de compartir la responsabilidad entre varios desarrolladores.

Cada uno debe escribir un conjunto bien definido y claro de funciones, las cuales al ser combinadas dentro de un módulo (esto se clarificara un poco más adelante) nos dará como resultado el producto final.

Esto nos lleva directamente a la tercera condición: si se va a dividir el trabajo entre varios programadores, **se debe descomponer el problema para permitir que el producto sea implementado como un conjunto de funciones escritas por separado empacadas juntas en diferentes módulos**.

¿De dónde provienen las funciones?

En general, las funciones provienen de al menos tres lugares:

- De Python mismo: varias funciones (como print()) son una **parte integral de Python**, y siempre están disponibles sin algún esfuerzo adicional del programador; se les llama a estas **funciones integradas**.
- De los **módulos preinstalados** de Python: muchas de las funciones, las cuales comúnmente son menos utilizadas que las integradas, están disponibles en módulos instalados juntamente con Python; para poder utilizar estas funciones el programador debe realizar algunos pasos adicionales (se explicará acerca de esto en un momento).
- **Directamente del código**: tu puedes escribir tus propias funciones, colocarlas dentro del código, y usarlas libremente.
- Existe una posibilidad más, pero se relaciona con clases, se omitirá por ahora.

Tu primera función

Se necesita definirla. Aquí, la palabra definir es significativa.

Así es como se ve la definición más simple de una función:

```
def function_name():
    function_body
```

- Siempre comienza con la palabra reservada def (que significa definir).
- Después de def va el nombre de la función (las reglas para darle nombre a las funciones son las mismas que para las variables).
- Después del nombre de la función, hay un espacio para un par de paréntesis (ahorita no contienen algo, pero eso cambiará pronto).
- La línea debe de terminar con dos puntos.
- La línea inmediatamente después de def marca el comienzo del cuerpo de la función - donde varias o (al menos una) instrucción anidada será ejecutada cada vez que la función sea invocada; nota: la función termina donde el anidamiento termina, se debe ser cauteloso.

A continuación se definirá la función. Se llamará **message**:

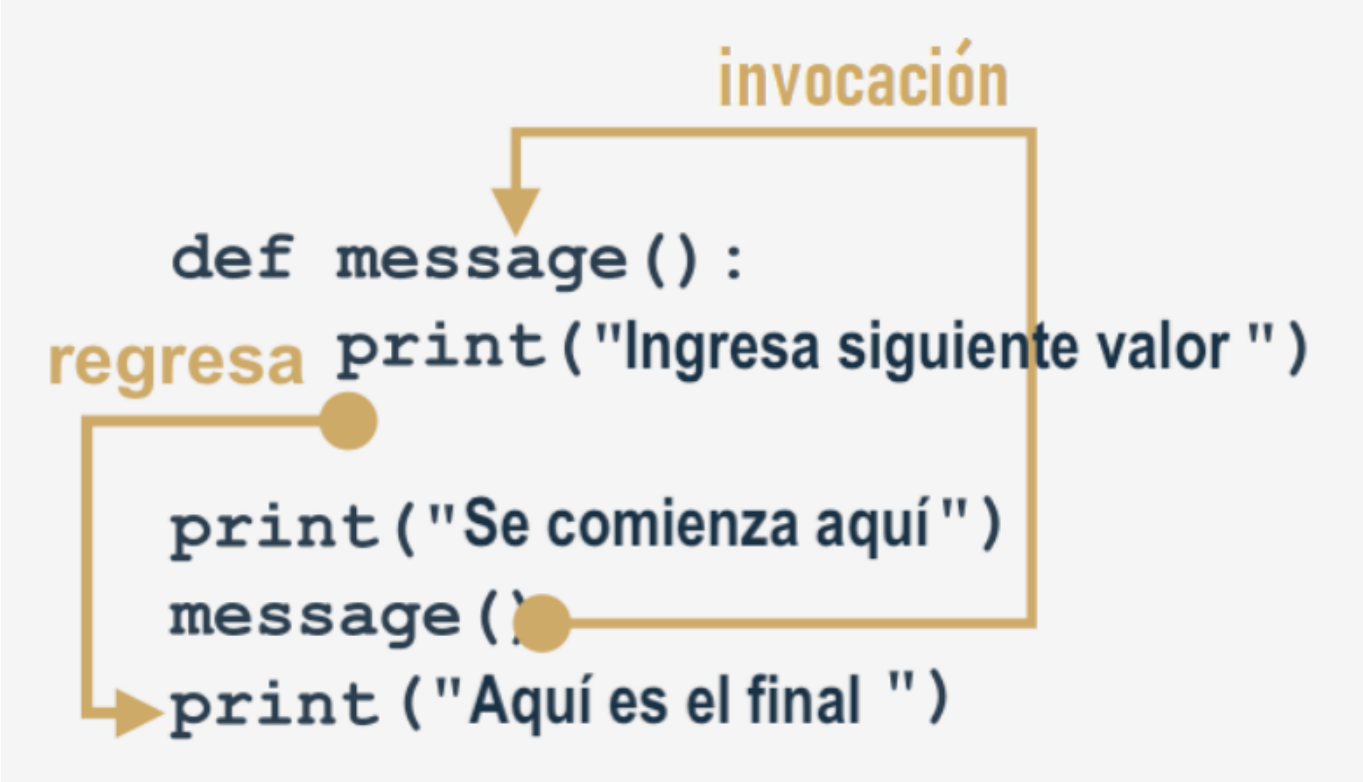
```
def message():
    print("Ingresa un valor: ")
```

Se ha modificado el código, se ha insertado la invocación de la función entre los dos mensajes:

```
def message():
    print("Ingresa un valor: ")

print("Se comienza aquí.")
message()
print("Se termina aquí.")
```

el funcionamiento de las funciones



Intenta mostrarte el proceso completo:

- Cuando se invoca una función, Python recuerda el lugar donde esto ocurre y salta hacia dentro de la función invocada.
- El cuerpo de la función es entonces ejecutado.
- Al llegar al final de la función, Python regresa al lugar inmediato después de donde ocurrió la invocación.

Existen dos consideraciones muy importantes, la primera de ella es:

No se debe invocar una función antes de que se haya definido.

Recuerda: Python lee el código de arriba hacia abajo. No va a adelantarse en el código para determinar si la función invocada está definida más adelante, el lugar correcto para definirla es antes de ser invocada.

Una función y una variable no pueden compartir el mismo nombre.

El asignar un valor al nombre «message» causa que Python olvide su rol anterior. La función con el nombre de message ya no estará disponible.

Afortunadamente, es posible combinar o mezclar el código con las funciones - no es forzoso colocar todas las funciones al inicio del archivo fuente.

Funciones parametrizadas

El potencial completo de una función se revela cuando puede ser equipada con una interface que es capaz de aceptar datos provenientes de la invocación. Dichos datos pueden modificar el comportamiento de la función, haciéndola más flexible y adaptable a condiciones cambiantes.

Un parámetro es una variable, pero existen dos factores que hacen a un parámetro diferente:

- **Los parámetros solo existen dentro de las funciones en donde han sido definidos**, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra reservada def.

- **La asignación de un valor a un parámetro de una función se hace en el momento en que la función se manda llamar o se invoca**, especificando el argumento correspondiente.

```
def function(parameter):
    ###
```

Recuerda que:

- Los parámetros solo existen dentro de las funciones (este es su entorno natural).
- Los argumentos existen fuera de las funciones, y son los que pasan los valores a los parámetros correspondientes.
- Especificar uno o más parámetros en la definición de la función es un requerimiento, y se debe de cumplir durante la invocación de la misma. Se debe proveer el mismo número de argumentos como haya parámetros definidos.

Existe una circunstancia importante que se debe mencionar.

Es posible tener una variable con el mismo nombre del parámetro de la función.

El siguiente código muestra un ejemplo de esto:

```
def message(number):
    print("Ingresa un número:", number)

number = 1234
message(1)
print(number)
```

Una situación como la anterior, activa un mecanismo denominado **sombreado**:

- El parámetro x sombrea cualquier variable con el mismo nombre, pero...
- ... solo dentro de la función que define el parámetro.

El parámetro llamado number es una entidad completamente diferente de la variable llamada number.

Una función puede tener tantos parámetros como se desee, pero entre más parámetros, es más difícil memorizar su rol y propósito.

Paso de parámetros posicionales

La técnica que asigna cada argumento al parámetro correspondiente, es llamada **paso de parámetros posicionales**, los argumentos pasados de esta manera son llamados **argumentos posicionales**.

Paso de argumentos con palabra clave

Python ofrece otra manera de pasar argumentos, donde **el significado del argumento está definido por su nombre**, no su posición, a esto se le denomina **paso de argumentos con palabra clave**.

Observa el siguiente código:

```
def introduction(first_name, last_name):
    print("Hola, mi nombre es", first_name, last_name)

introduction(first_name = "James", last_name = "Bond")
```

```
introduction(last_name = "Skywalker", first_name = "Luke")
```

El concepto es claro: los valores pasados a los parámetros son precedidos por el nombre del parámetro al que se le va a pasar el valor, seguido por el signo de =.

La posición no es relevante aquí, cada argumento conoce su destino con base en el nombre utilizado.

Debes de poder predecir la salida. Ejecuta el código y verifica tu respuesta.

Por supuesto que **no se debe de utilizar el nombre de un parámetro que no existe**.

El siguiente código provocará un error de ejecución:

```
def introduction(first_name, last_name):
    print("Hola, mi nombre es", first_name, last_name)

introduction(surname="Skywalker", first_name="Luke")
```

Esto es lo que Python arrojará:

```
TypeError: introduction() got an unexpected keyword argument 'surname'
```

Combinar argumentos posicionales y de palabra clave

Es posible combinar ambos tipos si se desea, solo hay una regla inquebrantable: se deben colocar primero los argumentos posicionales y después los de palabra clave.

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)

adding(1, 2, 3) # 1 + 2 + 3 = 6
adding(c = 1, a = 2, b = 3) # 2 + 3 + 1 = 6
adding(3, c = 1, b = 2) # 3 + 2 + 1 = 6
```

- El argumento (3) para el parámetro a es pasado utilizando la forma posicional.
- Los argumentos para c y b son especificados con palabras clave.

```
adding(3, a = 1, b = 2) # ERROR
```

Funciones parametrizadas: más detalles

En ocasiones ocurre que algunos valores de ciertos argumentos son más utilizados que otros. Dichos argumentos tienen valores predefinidos los cuales pueden ser considerados cuando los argumentos correspondientes han sido omitidos.

Uno de los apellidos más comunes en Latinoamérica es González. Tomémoslo para el ejemplo.

El valor por default para el parámetro se asigna de la siguiente manera:

```
def introduction(first_name, last_name="González"):
    print("Hola, mi nombre es", first_name, last_name)
```

Solo se tiene que colocar el nombre del parámetro seguido del signo de = y el valor por default.

Invoquemos la función de manera normal:

```
introduction("Jorge", "Pérez") # Hola, mi nombre es Jorge Pérez.
```

No parece haber cambiado algo, pero cuando se invoca la función de una manera inusual, como esta:

```
introduction("Enrique") # Hola, mi nombre es Enrique González
```

o así:

```
introduction(first_name="Guillermo") # Hola, mi nombre es Guillermo González
```

Es importante recordar que **primero se especifican los argumentos posicionales y después los de palabras clave**. Es por esa razón que si se intenta ejecutar el siguiente código:

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # salida: 3
subtra(a=5, 2)    # Syntax Error
```

```
def add_numbers(a, b=2, c):
    print(a + b + c)

add_numbers(a=1, c=3) # SyntaxError - a non-default argument (c) follows a default
argument (b=2)
```

Efectos y resultados: la instrucción return

Para lograr que las funciones devuelvan un valor (pero no solo para ese propósito) se utiliza la instrucción return (regresar o retornar).

Esta palabra nos da una idea completa de sus capacidades. Nota: es una **palabra clave reservada** de Python.

La instrucción return tiene dos variantes diferentes: considerémoslas por separado.

return sin una expresión

La primera consiste en la palabra reservada en sí, sin nada que la siga.

Cuando se emplea dentro de una función, provoca la terminación inmediata de la ejecución de la función, y un retorno instantáneo (de ahí el nombre) al punto de invocación.

Nota: si una función no está destinada a producir un resultado, emplear la instrucción returnno es obligatorio, se ejecutará implícitamente al final de la función.

De cualquier manera, se puede emplear para terminar las actividades de una función, antes de que el control llegue a la última línea de la función.

Consideremos la siguiente función:

```
def happy_new_year(wishes = True):
    print("Tres...")
```

```
    print("Dos...")
    print("Uno...")
    if not wishes:
        return

    print("¡Feliz año nuevo!")
```

Cuando se invoca sin ningún argumento:

```
happy_new_year()
```

La función produce un poco de ruido; la salida se verá así:

```
Tres...
Dos...
Uno...
¡Feliz año nuevo!
```

Al proporcionar False como argumento:

```
happy_new_year(False)
```

Se modificará el comportamiento de la función; la instrucción return provocará su terminación justo antes de los deseos. Esta es la salida actualizada:

```
Tres...
Dos...
Uno...
```

return con una expresión

La segunda variante de return está extendida con una expresión:

```
def function():
    return expression
```

Existen dos consecuencias de usarla:

- Provoca la terminación inmediata de la ejecución de la función (nada nuevo en comparación con la primer variante).
- Además, la función evaluará el valor de la expresión y lo devolverá (de ahí el nombre una vez más) como el resultado de la función.

Si, este ejemplo es sencillo:

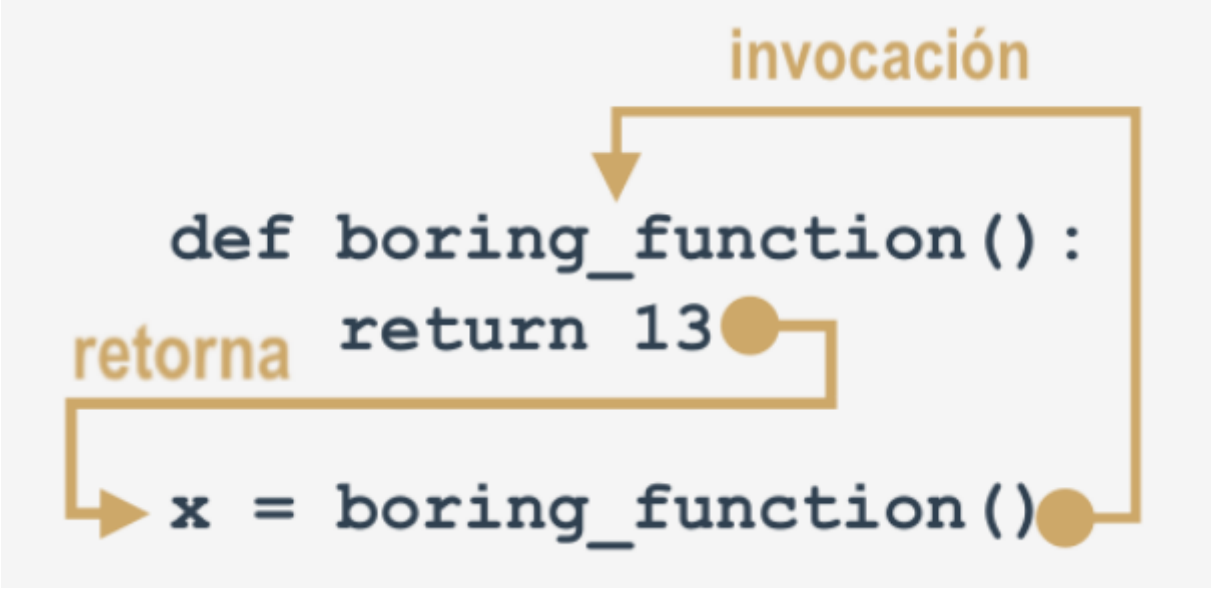
```
def boring_function():
    return 123

x = boring_function()

print("La función boring_function ha devuelto su resultado. Es:", x)
```

El fragmento de código escribe el siguiente texto en la consola:

```
La función boring_function ha devuelto su resultado. Es: 123
```



La instrucción **return**, enriquecida con la expresión (la expresión es muy simple aquí), «transporta» el valor de la expresión al lugar donde se ha invocado la función.

El resultado se puede usar libremente aquí, por ejemplo, para ser asignado a una variable.

También puede ignorarse por completo y perderse sin dejar rastro.

Ten en cuenta que no estamos siendo muy educados aquí: la función devuelve un valor y lo ignoramos (no lo usamos de ninguna manera):

```
def boring_function():
    print("'Modo aburrimiento' ON.")
    return 123

print("¡Esta lección es interesante!")
boring_function()
print("Esta lección es aburrida...")
```

El programa produce el siguiente resultado:

```
¡Esta lección es interesante!
'Modo aburrimiento' ON.
Esta lección es aburrida...
```

No olvides:

- Siempre se te permite ignorar el resultado de la función y estar satisfecho con el efecto de la función (si la función tiene alguno).
- Si una función intenta devolver un resultado útil, debe contener la segunda variante de la instrucción return.

Unas pocas palabras acerca de None

Permítenos presentarte un valor muy curioso (para ser honestos, un valor que es ninguno) llamado **None**.

Sus datos no representan valor razonable alguno; en realidad, no es un valor en lo absoluto; por lo tanto, no debe participar en ninguna expresión.

Por ejemplo, un fragmento de código como el siguiente:

```
print(None + 2)
```

Causará un error de tiempo de ejecución, descrito por el siguiente mensaje de diagnóstico:

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Nota: None es una **palabra clave reservada**.

Solo existen dos tipos de circunstancias en las que None se puede usar de manera segura:

- Cuando se le asigna a una variable (o se devuelve como el resultado de una función).
- Cuando se compara con una variable para diagnosticar su estado interno.

Al igual que aquí:

```
value = None
if value is None:
    print("Lo siento, no contienes ningún valor")
```

No olvides esto: si una función no devuelve un cierto valor utilizando una cláusula de expresión return, se asume que devuelve implícitamente None.

Efectos y resultados: listas y funciones

¿Se puede enviar una lista a una función como un argumento?

¡Por supuesto que se puede! Cualquier entidad reconocible por Python puede desempeñar el papel de un argumento de función, aunque debes asegurarte de que la función sea capaz de hacer uso de él.

Entonces, si pasas una lista a una función, la función tiene que manejarla como una lista.

¿Puede una lista ser el resultado de una función?

¡Si, por supuesto! Cualquier entidad reconocible por Python puede ser un resultado de función.

ejercicio

```
Tu tarea es escribir y probar una función que toma un argumento (un año) y devuelve
True si el año es un año bisiesto, o False si no lo es.

Parte del esqueleto de la función ya está en el editor.

Nota: también hemos preparado un breve código de prueba, que puedes utilizar para
probar tu función.

El código utiliza dos listas: una con los datos de prueba y la otra con los
resultados esperados. El código te dirá si alguno de tus resultados no es válido.
```

```
def is_year_leap(year):
  if year >= 1582:
    if year % 4 != 0: retorno = False
    elif year % 100 != 0: retorno = True
    elif year % 400 != 0: retorno = False
    else: retorno = True
  else:
    retorno = False

  return retorno


test_data = [1900, 2000, 2016, 1987]
test_results = [False, True, True, False]
for i in range(len(test_data)):
    yr = test_data[i]
    print(yr,"->",end="")
    result = is_year_leap(yr)
    if result == test_results[i]:
        print("OK")
    else:
        print("Fallido")
```

ejercicio

```
Tu tarea es escribir y probar una función que toma dos argumentos (un año y un mes)
y devuelve el número de días del mes/año dado (mientras que solo febrero es
sensible al valor year, tu función debería ser universal).

La parte inicial de la función está lista. Ahora, haz que la función devuelva None
si los argumentos no tienen sentido.

Por supuesto, puedes (y debes) utilizar la función previamente escrita y probada
(LABORATORIO 4.1.3.6). Puede ser muy útil. Te recomendamos que utilices una lista
con los meses. Puedes crearla dentro de la función; este truco acortará
significativamente el código.

Hemos preparado un código de prueba. Amplíalo para incluir más casos de prueba.
```

```
def is_year_leap(year):
  if year >= 1582:
    if year % 4 != 0: retorno = False
    elif year % 100 != 0: retorno = True
    elif year % 400 != 0: retorno = False
    else: retorno = True
  else:
    retorno = False

  return retorno

def days_in_month(year, month):
  dias = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  if is_year_leap(year):
      dias[1] = 29
```

```
  return dias[month-1]

test_years = [1900, 2000, 2016, 1987]
test_months = [2, 2, 1, 11]
test_results = [28, 29, 31, 30]
for i in range(len(test_years)):
    yr = test_years[i]
    mo = test_months[i]
    print(yr, mo, "->", end="")
    result = days_in_month(yr, mo)
    if result == test_results[i]:
        print("OK")
    else:
        print("Fallido")
```

ejercicio

```
Tu tarea es escribir y probar una función que toma tres argumentos (un año, un mes
y un día del mes) y devuelve el día correspondiente del año, o devuelve None si
cualquiera de los argumentos no es válido.

Debes utilizar las funciones previamente escritas y probadas. Agrega algunos casos
de prueba al código. Esta prueba es solo el comienzo.
```

Las funciones y sus alcances

Comencemos con una definición:

El alcance de un nombre (por ejemplo, el nombre de una variable) es la parte del código donde el nombre es reconocido correctamente.

Por ejemplo, el alcance del parámetro de una función es la función en sí. El parámetro es inaccesible fuera de la función.

```
def scope_test():
    x = 123


scope_test()
print(x)
```

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

Comencemos revisando si una variable creada fuera de una función es visible dentro de una función. En otras palabras, ¿El nombre de la variable se propaga dentro del cuerpo de la función?

```
def my_function():
    print("¿Conozco a la variable?", var)


var = 1
my_function()
```

```
print(var)
```

El resultado de la prueba es positivo, el código da como salida:

```
¿Conozco a la variable? 1
1
```

La respuesta es: una variable que existe fuera de una función tiene alcance dentro del cuerpo de la función.

Esta regla tiene una excepción muy importante. Intentemos encontrarla.

Hagamos un pequeño cambio al código:

```
def my_function():
    var = 2
    print("¿Conozco a la variable?", var)


var = 1
my_function()
print(var)
```

El resultado ha cambiado también, el código arroja una salida con una ligera diferencia:

```
¿Conozco a la variable? 2
1
```

¿Qué es lo que ocurrió?

- La variable var creada dentro de la función no es la misma que la que se definió fuera de ella, parece ser que hay dos variables diferentes con el mismo nombre.
- La variable de la función es una sombra de la variable fuera de la función.

La regla anterior se puede definir de una manera más precisa y adecuada:

Una variable que existe fuera de una función tiene un alcance dentro del cuerpo de la función, excluyendo a aquellas que tienen el mismo nombre.

También significa que **el alcance de una variable existente fuera de una función solo se puede implementar dentro de una función cuando su valor es leído**. El asignar un valor hace que la función cree su propia variable.

Las funciones y sus alcances: la palabra clave reservada global

Al llegar a este punto, debemos hacernos la siguiente pregunta: ¿Una función es capaz de modificar una variable que fue definida fuera de ella? Esto sería muy incomodo.

Afortunadamente, la respuesta es no.

Existe un método especial en Python el cual puede **extender el alcance de una variable incluyendo el cuerpo de las funciones** para poder no solo leer los valores de las variables sino también modificarlos.

Este efecto es causado por la palabra clave reservada llamada global:

```
global name
```

```
global name1, name2, ...
```

El utilizar la palabra reservada dentro de una función con el nombre o nombres de las variables separados por comas, obliga a Python a abstenerse de crear una nueva variable dentro de la función; se empleará la que se puede acceder desde el exterior.

En otras palabras, este nombre se convierte en global (tiene un **alcance global**, y no importa si se esta leyendo o asignando un valor).

```
def my_function():
    global var
    var = 2
    print("¿Conozco a aquella variable?", var)


var = 1
my_function()
print(var)
```

Se ha agregado la palabra global a la función.

El código ahora da como salida:

```
¿Conozco a aquella variable? 2
2
```

Como interactúa la función con sus argumentos

Ahora descubramos como la función interactúa con sus argumentos.

El código en el editor nos enseña algo. Como puedes observar, la función cambia el valor de su parámetro. ¿Este cambio afecta el argumento?

```
def my_function(n):
    print("Yo recibí", n)
    n += 1
    print("Ahora tengo", n)


var = 1
my_function(var)
print(var)
```

La salida del código es:

```
Yo recibí 1
Ahora tengo 2
1
```

La conclusión es obvia - **al cambiar el valor del parámetro este no se propaga fuera de la función** (más específicamente, no cuando la variable es un valor escalar, como en el ejemplo).

Esto también significa que una función recibe el **valor del argumento**, no el argumento en sí. Esto es cierto para los valores escalares.

Vale la pena revisar cómo funciona esto con las listas (¿Recuerdas las peculiaridades de asignar rebanadas de listas en lugar de asignar la lista entera?)

El siguiente ejemplo arrojará luz sobre el asunto:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    my_list_1 = [0, 1]
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)


my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

La salida del código es:

```
Print #1: [2, 3]
Print #2: [2, 3]
Print #3: [0, 1]
Print #4: [2, 3]
Print #5: [2, 3]
```

Parece ser que se sigue aplicando la misma regla.

Finalmente, la diferencia se puede observar en el siguiente ejemplo:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    del my_list_1[0]  # Presta atención a esta línea.
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)

my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

No se modifica el valor del parámetro

```
my_list_1
```

(ya se sabe que no afectará el argumento), en lugar de ello se modificará la lista identificada por el.

El resultado puede ser sorprendente. Ejecuta el código y verifícalo:

```
Print #1: [2, 3]
Print #2: [2, 3]
Print #3: [3]
Print #4: [3]
Print #5: [3]
```

¿Lo puedes explicar?

Intentémoslo:

- Si el argumento es una lista, el cambiar el valor del parámetro correspondiente no afecta la lista (Recuerda: las variables que contienen listas son almacenadas de manera diferente que las escalares).
- Pero si se modifica la lista identificada por el parámetro (Nota: ¡La lista, no el parámetro!), la lista reflejará el cambio.

Algunas funcione simples: recursividad

Este termino puede describir muchos conceptos distintos, pero uno de ellos, hace referencia a la programación computacional.

Aquí, la recursividad es una **técnica donde una función se invoca a si misma**.

Tanto el factorial como la serie Fibonacci, son las mejores opciones para ilustrar este fenómeno.

La serie de Fibonacci es un claro ejemplo de recursividad. Ya te dijimos que:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem_1 = elem_2 = 1
    the_sum = 0
    for i in range(3, n + 1):
        the_sum = elem_1 + elem_2
        elem_1, elem_2 = elem_2, the_sum
    return the_sum


for n in range(1, 10):  # probando
    print(n, "->", fib(n))
```

¿Puede ser empleado en el código? Por supuesto que puede. Puede hacer el código más corto y claro.

La segunda versión de la función fib() hace uso directo de la recursividad:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1
    return fib(n - 1) + fib(n - 2)
```

El código es mucho más claro ahora.

¿Pero es realmente seguro?, ¿Implica algún riesgo?

Si, existe algo de riesgo. Si no se considera una condición que detenga las invocaciones recursivas, el programa puede entrar en un bucle infinito. Se debe ser cuidadoso.

El factorial también tiene un lado recursivo. Observa:

```
n! = 1 × 2 × 3 × ... × n-1 × n
```

Es obvio que:

```
1 × 2 × 3 × ... × n-1 = (n-1)!
```

Entonces, finalmente, el resultado es:

```
n! = (n-1)! × n
```

Esto se empleará en nuestra nueva solución.

```
def factorial_function(n):
    if n < 0:
        return None
    if n < 2:
        return 1
    return n * factorial_function(n - 1)
```

From:
https://matebcn.eu/ - **miguel angel torres egea**

Permanent link:
https://matebcn.eu/info:cursos:netacad:python:pe1m4:funciones

Last update: **21/06/2022 19:22**