Funciones en Python

Anteriormente hemos usado funciones nativas que vienen con Python como len() para calcular la longitud de una lista, pero al igual que en otros lenguajes de programación, también podemos definir nuestras propias funciones. Para ello hacemos uso de def.

def nombre\_funcion(argumentos):

código

return retorno

Cualquier función tendrá un nombre, unos argumentos de entrada, un código a ejecutar y unos parámetros de salida. Al igual que las funciones matemáticas, en programación nos permiten realizar diferentes operaciones con la entrada, para entregar una determinada salida que dependerá del código que escribamos dentro. Por lo tanto, es totalmente análogo al clásico y=f(x) de las matemáticas.

def f(x):

return 2\*x

y = f(3)

print(y) # 6

Algo que diferencia en cierto modo las funciones en el mundo de la programación, es que no sólo realizan una operación con sus entradas, sino que también parten de los siguientes principios:

El principio de reusabilidad, que nos dice que si por ejemplo tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función. Esto nos evitaría tener código repetido, y que modificarlo fuera más fácil, ya que bastaría con cambiar la función una vez.

Y el principio de modularidad, que defiende que en vez de escribir largos trozos de código, es mejor crear módulos o funciones que agrupen ciertos fragmentos de código en funcionalidades específicas, haciendo que el código resultante sea más fácil de leer.

Pasando argumentos de entrada

Empecemos por la función más sencilla de todas. Una función sin parámetros de entrada ni parámetros de salida.

def di\_hola():

print("Hola")

Hemos declarado o definido la función. El siguiente paso es llamarla con di\_hola(). Si lo realizamos veremos que se imprime Hola.

di\_hola() # Hola

Vamos a complicar un poco las cosas pasando un argumento de entrada. Ahora si pasamos como entrada un nombre, se imprimirá Hola y el nombre.

def di\_hola(nombre):

print("Hola", nombre)

di\_hola("Juan")

# Hola Juan

Python permite pasar argumentos también de otras formas. A continuación las explicamos todas.

Argumentos por posición

Los argumentos por posición o posicionales son la forma más básica e intuitiva de pasar parámetros. Si tenemos una función resta() que acepta dos parámetros, se puede llamar como se muestra a continuación.

def resta(a, b):

return a-b

resta(5, 3) # 2

Al tratarse de parámetros posicionales, se interpretará que el primer número es la a y el segundo la b. El número de parámetros es fijo, por lo que si intentamos llamar a la función con solo uno, dará error.

#resta(1) # Error! TypeError

Tampoco es posible usar mas argumentos de los tiene la función definidos, ya que no sabría que hacer con ellos. Por lo tanto si lo intentamos, Python nos dirá que toma 2 posicionales y estamos pasando 3, lo que no es posible.

#TypeError: resta() takes 2 positional arguments but 3 were given

#resta(5,4,3) # Error

Argumentos por nombre

Otra forma de llamar a una función, es usando el nombre del argumento con = y su valor. El siguiente código hace lo mismo que el código anterior, con la diferencia de que los argumentos no son posicionales.

resta(a=3, b=5) # -2

Al indicar en la llamada a la función el nombre de la variable y el valor, el orden ya no importa, y se podría llamar de la siguiente forma.

resta(b=5, a=3) # -2

Como es de esperar, si indicamos un argumento que no ha sido definido como parámetro de entrada, tendremos un error.

#resta() got an unexpected keyword argument 'c'

#resta(b=5, c=3) # Error!

Argumentos por defecto

Tal vez queramos tener una función con algún parámetro opcional, que pueda ser usado o no dependiendo de diferentes circunstancias. Para ello, lo que podemos hacer es asignar un valor por defecto a la función. En el siguiente caso c valdría cero salvo que se indique lo contrario.

def suma(a, b, c=0):

return a+b+c

suma(5,5,3) # 13

Dado que el parámetro c tiene un valor por defecto, la función puede ser llamada sin ese valor.

suma(4,3) # 7

Podemos incluso asignar un valor por defecto a todos los parámetros, por lo que se podría llamar a la función sin ningún argumento de entrada.

def suma(a=3, b=5, c=0):

return a+b+c

suma() # 8

Las siguientes llamadas a la función también son válidas

suma(1) # 6

suma(4,5) # 9

suma(5,3,2) # 10

O haciendo uso de lo que hemos visto antes y usando los nombres de los argumentos.

suma(a=5, b=3) #8

Argumentos de longitud variable

En el ejemplo con argumentos por defecto, hemos visto que la función puede ser llamada con diferente número de argumentos de entrada, pero esto no es realmente una función con argumentos de longitud variable, ya que existe un número máximo.

Imaginemos que queremos una función suma() como la de antes, pero en este caso necesitamos que sume todos los números de entrada que se le pasen, sin importar si son 3 o 100. Una primera forma de hacerlo sería con una lista.

def suma(numeros):

total = 0

for n in numeros:

total += n

return total

suma([1,3,5,4]) # 13

La forma es válida y cumple nuestro requisito, pero realmente no estamos trabajando con argumentos de longitud variable. En realidad tenemos un solo argumento que es una lista de números.

Por suerte, Python tiene una herramienta muy potente. Si declaramos un argumento con \*, esto hará que el argumento que se pase sea empaquetado en una tupla de manera automática. No confundir \* con los punteros en otros lenguajes de programación, no tiene nada que ver.

def suma(\*numeros):

print(type(numeros))

# <class 'tuple'>

total = 0

for n in numeros:

total += n

return total

suma(1, 3, 5, 4) # 13

El resultado es igual que el anterior, y podemos ver como efectivamente numeros es de la clase tuple. También podemos hacer otras llamadas con diferente número de argumentos

suma(6) # 6

suma(6, 4, 10) # 20

suma(6, 4, 10, 20, 4, 6, 7) # 57

Usando doble \*\* es posible también tener como parámetro de entrada una lista de elementos almacenados en forma de clave y valor. En este caso podemos iterar los valores haciendo uso de items().

def suma(\*\*kwargs):

suma = 0;

for key, value in kwargs.items():

print(key, value)

suma += value

return suma

suma(a=5, b=20, c=23) # 48

De igual manera, podemos pasar un diccionario como parámetro de entrada.

def suma(\*\*kwargs):

suma = 0

for key, value in kwargs.items():

print(key, value)

suma += value

return suma

di = {'a': 10, 'b':20}

suma(\*\*di) # 30

Sentencia return

El uso de la sentencia return permite realizar dos cosas:

Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada.

Devolver uno o varios parámetros, fruto de la ejecución de la función.

En lo relativo a lo primero, una vez se llama a return se para la ejecución de la función y se vuelve o retorna al punto donde fue llamada. Es por ello por lo que el código que va después del return no es ejecutado en el siguiente ejemplo.

def mi\_funcion():

print("Entra en mi\_funcion")

return

print("No llega")

mi\_funcion() # Entra en mi\_funcion

Por ello, sólo llamamos a return una vez hemos acabado de hacer lo que teníamos que hacer en la función.

Por otro lado, se pueden devolver parámetros. Normalmente las funciones son llamadas para realizar unos cálculos en base a una entrada, por lo que es interesante poder devolver ese resultado a quien llamó a la función.

def di\_hola():

return "Hola"

di\_hola()

# 'Hola'

También es posible devolver mas de una variable, separadas por ,. En el siguiente ejemplo tenemos una función que calcula la suma y media de tres números, y devuelve su resultado.

def suma\_y\_media(a, b, c):

suma = a+b+c

media = suma/3

return suma, media

suma, media = suma\_y\_media(9, 6, 3)

print(suma) # 18

print(media) # 6.0

Documentación

Ahora que ya tenemos nuestras propias funciones creadas, tal vez alguien se interese en ellas y podamos compartírselas. Las funciones pueden ser muy complejas, y leer código ajeno no es tarea fácil. Es por ello por lo que es importante documentar las funciones. Es decir, añadir comentarios para indicar como deben ser usadas.

def mi\_funcion\_suma(a, b):

"""

Descripción de la función. Como debe ser usada,

que parámetros acepta y que devuelve

"""

return a+b

Para ello debemos usar la triple comilla """ al principio de la función. Se trata de una especie de comentario que podemos usar para indicar como la función debe ser usada. No se trata de código, es un simple comentario un tanto especial, conocido como docstring.

Ahora cualquier persona que tenga nuestra función, podrá llamar a la función help() y obtener la ayuda de como debe ser usada.

help(mi\_funcion\_suma)

Otra forma de acceder a la documentación es la siguiente.

print(mi\_funcion\_suma.\_\_doc\_\_)

Para saber más: Las descripciones de las funciones suelen ser un poco mas detalladas de lo que hemos mostrado. En la PEP257 se define en detalle como debería ser.

Anotaciones en funciones

Existe una funcionalidad relativamente reciente en Python llamada function annotation o anotaciones en funciones. Dicha funcionalidad nos permite añadir metadatos a las funciones, indicando los tipos esperados tanto de entrada como de salida.

def multiplica\_por\_3(numero: int) -> int:

return numero\*3

multiplica\_por\_3(6) # 18

Las anotaciones son muy útiles de cara a la documentación del código, pero no imponen ninguna norma sobre los tipos. Esto significa que se puede llamar a la función con un parámetro que no sea int, y no obtendremos ningún error.

multiplica\_por\_3("Cadena")

# 'CadenaCadenaCadena'

Paso por valor y referencia

En muchos lenguajes de programación existen los conceptos de paso por valor y por referencia que aplican a la hora de como trata una función a los parámetros que se le pasan como entrada. Su comportamiento es el siguiente:

Si usamos un parámetro pasado por valor, se creará una copia local de la variable, lo que implica que cualquier modificación sobre la misma no tendrá efecto sobre la original.

Con una variable pasada como referencia, se actuará directamente sobre la variable pasada, por lo que las modificaciones afectarán a la variable original.

En Python las cosas son un poco distintas, y el comportamiento estará definido por el tipo de variable con la que estamos tratando. Veamos un ejemplo de paso por valor.

x = 10

def funcion(entrada):

entrada = 0

funcion(x)

print(x) # 10

Iniciamos la x a 10 y se la pasamos a funcion(). Dentro de la función hacemos que la variable valga 0. Dado que Python trata a los int como pasados por valor, dentro de la función se crea una copia local de x, por lo que la variable original no es modificada.

No pasa lo mismo si por ejemplo x es una lista como en el siguiente ejemplo. En este caso Python lo trata como si estuviese pasada por referencia, lo que hace que se modifique la variable original. La variable original x ha sido modificada.

x = [10, 20, 30]

def funcion(entrada):

entrada.append(40)

funcion(x)

print(x) # [10, 20, 30, 40]

El ejemplo anterior nos podría llevar a pensar que si en vez de añadir un elemento a x, hacemos x=[], estaríamos destruyendo la lista original. Sin embargo esto no es cierto.

x = [10, 20, 30]

def funcion(entrada):

entrada = []

funcion(x)

print(x)

# [10, 20, 30]

Una forma muy útil de saber lo que pasa por debajo de Python, es haciendo uso de la función id(). Esta función nos devuelve un identificador único para cada objeto. Volviendo al primer ejemplo podemos ver como los objetos a los que “apuntan” x y entrada son distintos.

x = 10

print(id(x)) # 4349704528

def funcion(entrada):

entrada = 0

print(id(entrada)) # 4349704208

funcion(x)

Sin embargo si hacemos lo mismo cuando la variable de entrada es una lista, podemos ver que en este caso el objeto con el que se trabaja dentro de la función es el mismo que tenemos fuera.

x = [10, 20, 30]

print(id(x)) # 4422423560

def funcion(entrada):

entrada.append(40)

print(id(entrada)) # 4422423560

funcion(x)

Args y Kwargs en Python

Si alguna vez has tenido que definir una función con un número variable de argumentos y no has sabido como hacerlo, a continuación te explicamos cómo gracias a los args y kwargs en Python.

Vamos a suponer que queremos una función que sume un conjunto de números, pero no sabemos a priori la cantidad de números que se quieren sumar. Si por ejemplo tuviéramos tres, la función sería tan sencilla como la siguiente.

def suma(a, b, c):

return a+b+c

suma(2, 4, 6)

#Salida: 12

El problema surge si por ejemplo queremos sumar cuatro números. Como es evidente, la siguiente llamada a la función anterior daría un error ya que estamos usando cuatro argumentos mientras que la función sólo soporta tres.

suma(2, 4, 6, 1)

#TypeError: suma() takes 3 positional arguments but 4 were given

Introducida ya la problemática, veamos como podemos resolver este problema con \*args y \*\*kwargs en Python.

Uso de \*args

Gracias a los \*args en Python, podemos definir funciones cuyo número de argumentos es variable. Es decir, podemos definir funciones genéricas que no aceptan un número determinado de parámetros, sino que se “adaptan” al número de argumentos con los que son llamados.

De hecho, el args viene de arguments en Inglés, o argumentos. Haciendo uso de \*args en la declaración de la función podemos hacer que el número de parámetros que acepte sea variable.

def suma(\*args):

s = 0

for arg in args:

s += arg

return s

suma(1, 3, 4, 2)

#Salida 10

suma(1, 1)

#Salida 2

Antes de nada, el uso del nombre args es totalmente arbitrario, por lo que podrías haberlo llamado como quisieras. Es una mera convención entre los usuarios de Python y resulta frecuente darle ese nombre. Lo que si es un requisito, es usar \* junto al nombre.

En el ejemplo anterior hemos visto como \*args puede ser iterado, ya que en realidad es una tupla. Por lo tanto iterando la tupla podemos acceder a todos los argumentos de entrada, y en nuestro caso sumarlos y devolverlos.

Nótese que es un mero ejemplo didáctico. En realidad podríamos hacer algo como lo siguiente, lo que sería mucho más sencillo.

def suma(\*args):

return sum(args)

suma(5, 5, 3)

#Salida 13

Con esto resolvemos nuestro problema inicial, en el que necesitábamos un número variable de argumentos. Sin embargo, hay otra forma que nos proporciona además un nombre asociado al argumento, con el uso de \*\*kwargs. La explicamos a continuación.

Uso de \*\*kwargs

Al igual que en \*args, en \*\*kwargs el nombre es una mera convención entre los usuarios de Python. Puedes usar cualquier otro nombre siempre y cuando respetes el \*\*.

En este caso, en vez de tener una tupla tenemos un diccionario. Puedes verificarlo de la siguiente forma con type().

def suma(\*\*kwargs):

print(type(kwargs))

suma(x=3)

#<class 'dict'>

Pero veamos un ejemplo más completo. A diferencia de \*args, los \*\*kwargs nos permiten dar un nombre a cada argumento de entrada, pudiendo acceder a ellos dentro de la función a través de un diccionario.

def suma(\*\*kwargs):

s = 0

for key, value in kwargs.items():

print(key, "=", value)

s += value

return s

suma(a=3, b=10, c=3)

#Salida

#a = 3

#b = 10

#c = 3

#16

Como podemos ver, es posible iterar los argumentos de entrada con items(), y podemos acceder a la clave key (o nombre) y el valor o value de cada argumento.

El uso de los \*\*kwargs es muy útil si además de querer acceder al valor de las variables dentro de la función, quieres darles un nombre que de una información extra.

Mezclando \*args y \*\*kwargs

Una vez entendemos el uso de \*args y \*\*kwargs, podemos complicar las cosas un poco más. Es posible mezclar argumentos normales con \*args y \*\*kwargs dentro de la misma función. Lo único que necesitas saber es que debes definir la función en el siguiente orden:

Primero argumentos normales.

Después los \*args.

Y por último los \*\*kwargs.

Veamos un ejemplo.

def funcion(a, b, \*args, \*\*kwargs):

print("a =", a)

print("b =", b)

for arg in args:

print("args =", arg)

for key, value in kwargs.items():

print(key, "=", value)

funcion(10, 20, 1, 2, 3, 4, x="Hola", y="Que", z="Tal")

#Salida

#a = 10

#b = 20

#args = 1

#args = 2

#args = 3

#args = 4

#x = Hola

#y = Que

#z = Tal

Y por último un truco que no podemos dejar sin mencionar es lo que se conoce como tuple unpacking. Haciendo uso de \*, podemos extraer los valores de una lista o tupla, y que sean pasados como argumentos a la función.

def funcion(a, b, \*args, \*\*kwargs):

print("a =", a)

print("b =", b)

for arg in args:

print("args =", arg)

for key, value in kwargs.items():

print(key, "=", value)

args = [1, 2, 3, 4]

kwargs = {'x':"Hola", 'y':"Que", 'z':"Tal"}

funcion(10, 20, \*args, \*\*kwargs)

#Salida

#a = 10

#b = 20

#args = 1

#args = 2

#args = 3

#args = 4

#x = Hola

#y = Que

#z = Tal

Anotaciones en Funciones

Function Annotations en Python

Las anotaciones en funciones o function annotations de Python nos permiten añadir el tipo de los argumentos de entrada y salida de una función. A continuación podemos ver un ejemplo con la función suma(), que recibe dos argumentos a, b y cuyo tipo se espera que sea int.

def suma(a: int, b: int) -> int:

return a + b

print(suma(7, 3))

# Salida: 10

Sin embargo es muy importante notar que Python ignora las anotaciones. Es decir, son una mera nota en el código que indica el tipo esperado, pero el siguiente código no daría ningún error. Más adelante explicaremos cómo realizar el chequeo de tipos.

suma(7.0, 3.0)

Las anotaciones en funciones fueron introducidas en la PEP3107 para Python 3, y más adelante se introdujo la PEP484 especificando la semántica que se debe usar.

Motivación y Necesidad de las Anotaciones

Python es un lenguaje de programación con tipado dinámico y duck typing, lo que significa que los tipos (int, string, etc) le dan igual. Precisamente esto es lo que hace que el siguiente código funcione. La función imprime puede ser llamada con cualquier tipo, ya que Python no realiza ninguna comprobación del tipo de var.

def imprime(var):

print(var)

imprime(1.0) # 1.0

imprime(3) # 3

imprime("Python") # Python

Sin embargo, en ciertas ocasiones esto nos puede traer problemas. ¿Y si queremos que la función imprime sólo acepte que var sea de un tipo concreto? Pues bien, las anotaciones en funciones o function annotations como acabamos de ver nos permiten especificar los tipos que se esperan recibir.

Ejemplos de Function Annotations

Antes de nada es importante notar que las anotaciones en funciones no definen per se una semántica propia. Es decir, podemos escribir lo que se nos ocurra después de cada argumento. Las anotaciones pueden ser accedidas usando \_\_annotations\_\_.

def suma(a: 'parametro 1', b: 'parametro 2') -> 'retorno':

return a + b

print(suma.\_\_annotations\_\_)

# Salida:

# {'a': 'parametro 1',

# 'b': 'parametro 2',

# 'return': 'retorno'}

Aunque como hemos dicho se puedan realizar anotaciones arbitrarias, suele ser común usar tipos de Python como int, str o float. En el siguiente ejemplo podemos ver como se combina una anotación con un valor por defecto [].

def filtrar\_pares(salida: 'list' = []) -> 'list':

return [i for i in salida if i%2 == 0]

print(filtrar\_pares([1, 2, 3, 4, 5, 6]))

# Salida: [2, 4, 6]

También es posible usar como anotaciones clases definidas por nosotros, como ClaseA.

class ClaseA:

pass

def funcion(a: ClaseA) -> ClaseA:

return a

a = ClaseA()

funcion(a)

Por último, las anotaciones no están limitadas a los argumentos de las funciones, sino que también se pueden asignar a variables que declaremos.

pi: float = 3.14

print(pi)

# Salida: 3.14

Sin embargo, como ya hemos visto anteriormente, Python no da error cuando los tipos no se corresponden como en el ejemplo anterior.

# Ojo: No sería correcto, pero Python no da error

pi: int = 3.14

print(pi)

# Salida: 3.14

Entonces uno se puede preguntar, ¿y para que sirven las function annotation, si Python las ignora? Pues bien, aunque las ignore, tenemos herramientas que nos permiten saber cuando no se están respetando. Lo vemos a continuación con el análisis estático del código.

Uso de mypy y Static Type Checking

Una primera forma de verificar que las funciones se llaman con los parámetros especificados por las anotaciones, sería lo siguiente. Sin embargo el error que obtendríamos sería en tiempo de ejecución. Es decir, nos encontraríamos con el error una vez el código estuviera ejecutándose. Por lo tanto, no recomendamos el uso del siguiente código.

# Nota: Ejemplo didáctico, no recomendado

def suma(a: int, b: int) -> int:

if isinstance(a, suma.\_\_annotations\_\_['a']) and isinstance(b, suma.\_\_annotations\_\_['b']):

return a + b

else:

raise Exception("Error de tipos")

print(suma(7, 3))

# Salida: 10

print(suma(7.0, 3.0))

# Salida: Exception: Error de tipos

Afortunadamente, tenemos herramientas como mypy que nos permiten hacer un chequeo estático de los tipos, obteniendo el error antes de que el código se ejecute. Lo podemos instalar de la siguiente manera.

$ pip instal mypy

Y volviendo al ejemplo anterior de la suma, podemos ver como el siguiente código si que pasaría los checks de mypy.

# suma\_correcta.py

def suma(a: int, b: int) -> int:

return a + b

print(suma(7, 3))

$ mypy suma\_correcta.py

Success: no issues found in 1 source file

Sin embargo si cambiamos los tipos de los parámetros de entrada, obtendremos un error.

# suma\_incorrecta.py

def suma(a: int, b: int) -> int:

return a + b

print(suma(7.0, "3"))

$ mypy suma\_incorrecta.py

suma\_incorrecta.py:5: error: Argument 1 to "suma" has incompatible type "float"; expected "int"

suma\_incorrecta.py:5: error: Argument 2 to "suma" has incompatible type "str"; expected "int"

Found 2 errors in 1 file (checked 1 source file)

Como hemos indicado, la ventaja de mypy es que realiza un análisis estático, es decir, sin ejecutar el código. Esto es algo muy importante ya que si de verdad queremos reforzar que se verifiquen los tipos, no tendría mucho sentido hacerlo en tiempo de ejecución, ya que sería demasiado tarde y tendríamos un error.

Recursividad

¿En qué trabajas? Estoy intentando arreglar los problemas que creé cuando intentaba arreglar los problemas que creé cuando intentaba arreglar los problemas que creé. Y así nació la recursividad.

La recursividad o recursión es un concepto que proviene de las matemáticas, y que aplicado al mundo de la programación nos permite resolver problemas o tareas donde las mismas pueden ser divididas en subtareas cuya funcionalidad es la misma. Dado que los subproblemas a resolver son de la misma naturaleza, se puede usar la misma función para resolverlos. Dicho de otra manera, una función recursiva es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma hasta llegar a un punto de salida.

Cualquier función recursiva tiene dos secciones de código claramente divididas:

Por un lado tenemos la sección en la que la función se llama a sí misma.

Por otro lado, tiene que existir siempre una condición en la que la función retorna sin volver a llamarse. Es muy importante porque de lo contrario, la función se llamaría de manera indefinida.

Veamos unos ejemplos con el factorial y la serie de fibonacci.

Calcular factorial

Uno de los ejemplos mas usados para entender la recursividad, es el cálculo del factorial de un número n!. El factorial de un número n se define como la multiplicación de todos sus números predecesores hasta llegar a uno. Por lo tanto 5!, leído como cinco factorial, sería 5\*4\*3\*2\*1.

Utilizando un enfoque tradicional no recursivo, podríamos calcular el factorial de la siguiente manera.

def factorial\_normal(n):

r = 1

i = 2

while i <= n:

r \*= i

i += 1

return r

factorial\_normal(5) # 120

Dado que el factorial es una tarea que puede ser dividida en subtareas del mismo tipo (multiplicaciones), podemos darle un enfoque recursivo y escribir la función de otra manera.

def factorial\_recursivo(n):

if n == 1:

return 1

else:

return n \* factorial\_recursivo(n-1)

factorial\_recursivo(5) # 120

Lo que realmente hacemos con el código anterior es llamar a la función factorial\_recursivo() múltiples veces. Es decir, 5! es igual a 5 \* 4! y 4! es igual a 4 \* 3! y así sucesivamente hasta llegar a 1.

Algo muy importante a tener en cuenta es que si realizamos demasiadas llamadas a la función, podríamos llegar a tener un error del tipo RecursionError. Esto se debe a que todas las llamadas van apilándose y creando un contexto de ejecución, algo que podría llegar a causar un stack overflow. Es por eso por lo que Python lanza ese error, para protegernos de llegar a ese punto.

Serie de Fibonacci

Otro ejemplo muy usado para ilustrar las posibilidades de la recursividad, es calcular la serie de fibonacci. Dicha serie calcula el elemento n sumando los dos anteriores n-1 + n-2. Se asume que los dos primeros elementos son 0 y 1.

def fibonacci\_recursivo(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fibonacci\_recursivo(n-1) + fibonacci\_recursivo(n-2)

Podemos ver que siempre que la n sea distinta de cero o uno, se llamará recursivamente a la función, buscando los dos elementos anteriores. Cuando la n valga cero o uno se dejará de llamar a la función y se devolverá un valor concreto. Podemos calcular el elemento 7, que será 0,1,1,2,3,5,8,13, es decir, 13.

fibonacci\_recursivo(7)

# 13