4 Funciones

Llamadas de función

En el contexto de la programación, una **función** es una secuencia de instrucciones con nombre que realiza un cálculo. Cuando defines una función, especifica el nombre y la secuencia de instrucciones. Más tarde, puedes "llamar" a la función por su nombre. Ya hemos visto un ejemplo de una llamada a una función:

```
>>> type(32)
<class 'int'>
```

El nombre de la función es `type`. La expresión entre paréntesis se llama el **argumento** de la función. El argumento es un valor o variable que estamos pasando a la función como entrada. El resultado, para la función `type`, es el tipo del argumento.

Es común decir que una función "toma" un argumento y "devuelve" un resultado. El resultado se llama el **valor de retorno**.

Funciones incorporadas (built-in functions)

Python proporciona una serie de funciones integradas importantes que podemos usar sin necesidad de proporcionar la definición de la función. Los creadores de Python escribieron un conjunto de funciones para resolver problemas comunes y las incluyeron en Python para que las utilicemos.

Las funciones `max` y `min` nos dan los valores más grandes y más pequeños en una lista, respectivamente:

Copyright 2026

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

La función `max` nos dice el "carácter más grande" en la cadena (que resulta ser la letra "w") y la función `min` nos muestra el carácter más pequeño (que es un espacio).

Otra función incorporada muy común es la función `len` que nos dice cuántos elementos hay en su argumento. Si el argumento a `len` es una cadena, devuelve el número de caracteres en la cadena.

```
>>> len('Hello world')
11
>>>
```

Estas funciones no se limitan a mirar las cadenas. Pueden operar en cualquier conjunto de valores, como veremos en capítulos posteriores.

Debe tratar los nombres de las funciones incorporadas como palabras reservadas (es decir, evitar usar "max" como nombre de variable).

Funciones de conversión de tipos

Python también proporciona funciones integradas que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte en un entero, si puede, o se queja de lo contrario:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` puede convertir valores de coma flotante en enteros, pero no se redondea; corta la parte de la fracción

```
>>> int(3.99999)
3
```

Copyright 2026

```
>>> int(-2.3)
-2
```

`float` convierte números enteros y cadenas en números de punto flotante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, `str` convierte su argumento en una cadena:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Números aleatorios

Dadas las mismas entradas, la mayoría de los programas de ordenador generan las mismas salidas cada vez, por lo que se dice que son **deterministas**. El determinismo suele ser algo bueno, ya que esperamos que el mismo cálculo dé el mismo resultado. Para algunas aplicaciones, sin embargo, queremos que el ordenador sea impredecible. Los juegos son un ejemplo obvio, pero hay más.

Hacer que un programa sea realmente no determinista resulta no ser tan fácil, pero hay formas de que al menos parezca no determinista. Uno de ellos es utilizar **algoritmos** que generan **números pseudoaleatorios**. Los números pseudoaleatorios no son realmente aleatorios porque son generados por un cálculo determinista, pero con solo mirar los números es casi imposible distinguirlos de los aleatorios.

El módulo "aleatorio" proporciona funciones que generan números pseudoaleatorios (que a partir de ahora llamaré "aleatorios").

La función `random` devuelve un float aleatorio entre 0.0 y 1.0 (incluyendo 0.0 pero no 1.0). Cada vez que llamas a la función `random`, obtienes el siguiente número en una larga serie. Para ver una muestra, ejecuta este bucle:

Copyright 2026

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Este programa produce la siguiente lista de 10 números aleatorios entre 0.0 y hasta, pero sin incluir 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Ejercicio 1: Ejecuta el programa en su sistema y vea qué números obtiene. Ejecute el programa más de una vez y vea qué números obtiene.

La función `random` es solo una de las muchas funciones que manejan números aleatorios. La función `randint` toma los parámetros `low` y `high`, y devuelve un número entero entre `low` y `high` (incluyendo ambos).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para elegir un elemento de una secuencia al azar, puedes usar `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
```

Copyright 2026

```
3
```

El módulo `random` también proporciona funciones para generar valores aleatorios a partir de distribuciones continuas que incluyen Gaussian, exponential, gamma y algunas más.

Funciones matemáticas

Python tiene un módulo `math` que proporciona la mayoría de las funciones matemáticas conocidas. Antes de que podamos usar el módulo, tenemos que importarlo:

```
>>> import math
```

Esta declaración crea un **objeto de módulo** llamado math. Si imprime el objeto de módulo, obtiene alguna información al respecto:

```
>>> print(math)
<module 'math' (built-in)>
```

El objeto módulo contiene las funciones y variables definidas en el módulo. Para acceder a una de las funciones, debe especificar el nombre del módulo y el nombre de la función, separados por un punto.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

El primer ejemplo calcula la base logarítmica 10 de la relación señal-ruido. El módulo matemático también proporciona una función llamada `log` que calcula los logaritmos base `e`.

El segundo ejemplo encuentra el seno de 'radians'. El nombre de la variable es un indicio de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) toman argumentos en radianes. Para convertir de grados a radianes, divide por 360 y multiplica por 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
```

Copyright 2026

```
>>> math.sin(radians)
0.7071067811865476
```

La expresión `math.pi` obtiene la variable `pi` del módulo matemático. El valor de esta variable es una aproximación de **π**, con una precisión de aproximadamente 15 dígitos.

Si sabes trigonometría, puedes verificar el resultado anterior comparándolo con la raíz cuadrada de dos dividido por dos:

```
>>> math.sqrt(2)/2.0
0.7071067811865476
```

Agregar nuevas funciones

Hasta ahora, solo hemos estado usando las funciones que vienen con Python, pero también es posible agregar nuevas funciones. Una **definición de función** especifica el nombre de una nueva función y la secuencia de sentencias que se ejecutan cuando se llama a la función. Una vez que definimos una función, podemos reutilizarla una y otra vez a lo largo de nuestro programa.

Aquí hay un ejemplo:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

`def` es una palabra clave que indica que esta es una definición de función. El nombre de la función es `print_lyrics`. Las reglas para los nombres de funciones son las mismas que para los nombres de variables: las letras, los números y algunos signos de puntuación son legales, pero el primer carácter no puede ser un número. No puedes usar una palabra clave como el nombre de una función, y debes evitar tener una variable y una función con el mismo nombre.

Los paréntesis vacíos después del nombre indican que esta función no acepta ningún argumento. Más adelante construiremos funciones que toman argumentos como sus entradas.

La primera línea de la definición de la función se llama el **encabezado**; el resto se llama el **cuerpo** . El encabezado debe terminar con dos puntos y el cuerpo debe estar sangrado. Por convención, la sangría es siempre de cuatro espacios. El cuerpo puede contener cualquier número de declaraciones.

Copyright 2026

Las cadenas en las declaraciones impresas están entre comillas. Las comillas simples y las comillas dobles hacen lo mismo; la mayoría de las personas usan comillas simples, excepto en casos como éste, donde aparece una comilla simple (que también es un apóstrofe) en la cadena.

Si escribes una definición de función en modo interactivo, el intérprete imprime puntos suspensivos (...) para hacerle saber que la definición no está completa:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

Para finalizar la función, debe ingresar una línea vacía (esto no es necesario en un script).

La definición de una función crea una variable con el mismo nombre.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

El valor de `print_lyrics` es un **objeto**, que tiene el tipo "función ".

La sintaxis para llamar a la nueva función es la misma que para las funciones incorporadas:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Una vez que hayas definido una función, puedes usarla dentro de otra función. Por ejemplo, para repetir el refrán anterior, podríamos escribir una función llamada `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Y luego llamar a "repeat_lyrics":

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Pero así no es realmente cómo va la canción.

Definiciones y usos

Recopilando los fragmentos de código de la sección anterior, todo el programa se ve así:

https://trinket.io/embed/python3/3b61a1059e

Este programa contiene dos definiciones de funciones: `print_lyrics` y `repeat_lyrics`. Las definiciones de funciones se ejecutan igual que otras declaraciones, pero el efecto es crear objetos de funciones. Las sentencias dentro de la función no se ejecutan hasta que se llama a la función, y la definición de la función no genera salida.

Como es de esperar, debes crear una función antes de poder ejecutarla. En otras palabras, la definición de la función debe ejecutarse antes de la primera vez que se llama.

Ejercicio 2 Mueve la última línea de este programa a la parte superior, para que la llamada a la función aparezca antes de las definiciones. Ejecuta el programa y observa qué mensaje de error recibe.

Ejercicio 3 Mueve la llamada de la función a la parte inferior y mueve la definición de `print_lyrics` después de la definición de `repeat_lyrics`. ¿Qué pasa cuando ejecutas este programa?

Flujo de ejecución

Para garantizar que una función se define antes de su primer uso, debes conocer el orden en que se ejecutan las instrucciones, lo que se denomina **flujo de ejecución**.

Copyright 2026

La ejecución siempre comienza en la primera declaración del programa. Las declaraciones se ejecutan de una en una, en orden de arriba a abajo.

Las definiciones de la función no alteran el flujo de ejecución del programa, pero recuerde que las sentencias dentro de la función no se ejecutan hasta que se llama a la función.

Una llamada de función es como un desvío en el flujo de ejecución. En lugar de ir a la siguiente instrucción, el flujo salta al cuerpo de la función, ejecuta todas las declaraciones allí y luego vuelve a retomar donde se detuvo. Eso suena bastante simple, hasta que recuerdas que una función puede llamar a otra. Mientras se encuentra en medio de una función, el programa podría tener que ejecutar las instrucciones en otra función. ¡Pero mientras ejecuta esa nueva función, el programa podría tener que ejecutar otra función!

Afortunadamente, Python es bueno para mantener un registro de dónde está, por lo que cada vez que una función se completa, el programa retoma el lugar donde se quedó en la función que lo llamó. Cuando llega al final del programa, termina.

¿Cuál es la moraleja de este sórdido cuento? Cuando lees un programa, no siempre quieres leer de arriba a abajo. A veces tiene más sentido si sigues el flujo de ejecución.

Parámetros y argumentos

Algunas de las funciones incorporadas que hemos visto requieren argumentos. Por ejemplo, cuando llamas `math.sin` pasas un número como argumento. Algunas funciones toman más de un argumento: `math.pow` toma dos, la base y el exponente.

Dentro de la función, los argumentos se asignan a variables llamadas **parámetros**. Aquí hay un ejemplo de una función definida por el usuario que toma un argumento:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro llamado `bruce`. Cuando se llama a la función, imprime el valor del parámetro (cualquiera que sea) dos veces.

Esta función funciona con cualquier valor que se pueda imprimir.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

Las mismas reglas de composición que se aplican a las funciones integradas también se aplican a las funciones definidas por el usuario, por lo que podemos usar cualquier tipo de expresión como argumento para `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

El argumento se evalúa antes de llamar a la función, por lo que en los ejemplos las expresiones "Spam '* 4 y math.cos (math.pi)` solo se evalúan una vez.

También puedes usar una variable como argumento:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

El nombre de la variable que pasamos como argumento (`michael`) no tiene nada que ver con el nombre del parámetro (`bruce`). No importa cómo se llamó el valor de regreso a casa (en la persona que llama). Aquí en `print_twice`, llamamos a todos `bruce`.

Copyright 2026

Funciones fructíferas y funciones nulas

Algunas de las funciones que utilizamos, como las funciones matemáticas, producen resultados. A falta de un nombre mejor, las llamo **funciones fructíferas**. Otras funciones, como `print_twice`, realizan una acción pero no devuelven un valor. Se les llama **funciones nulas** (void functions).

Cuando llama a una función fructífera, casi siempre quiere hacer algo con el resultado; por ejemplo, puede asignarlo a una variable o usarlo como parte de una expresión:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Cuando llamas a una función en modo interactivo, Python muestra el resultado:

```
>>> math.sqrt(5)
2.23606797749979
```

Pero en un script, si llama a una función fructífera y no almacena el resultado de la función en una variable, el valor de retorno desaparece en la nada.

```
math.sqrt(5)
```

Este script calcula la raíz cuadrada de 5, pero como no almacena el resultado en una variable ni muestra el resultado, no es muy útil.

Las funciones nulas pueden mostrar algo en la pantalla o tener algún otro efecto, pero no tienen un valor de retorno. Si intenta asignar el resultado a una variable, obtiene un valor especial llamado `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

Copyright 2026

El valor `None` no es el mismo que la cadena `None`. Es un valor especial que tiene su propio tipo:

```
>>> print(type(None))
<class 'NoneType'>
```

Para devolver un resultado de una función, usamos la declaración `return` en nuestra función. Por ejemplo, podríamos hacer una función muy simple llamada `addtwo` que suma dos números y devuelve un resultado.

https://trinket.io/embed/python3/a50501b656

Cuando este script se ejecute, la instrucción `print` imprimirá" 8 "porque la función `addtwo` fue llamada con 3 y 5 como argumentos. Dentro de la función, los parámetros `a` y `b` fueron 3 y 5 respectivamente. La función calculó la suma de los dos números y la colocó en la variable de función local denominada `added`. Luego usó la declaración `return` para enviar el valor calculado al código de llamada como el resultado de la función, que se asignó a la variable `x` y se imprimió.

¿Para qué crear funciones?

Puede que no esté claro por qué vale la pena dividir un programa en funciones. Hay varias razones:

- La creación de una nueva función le brinda la oportunidad de nombrar un grupo de declaraciones, lo que hace que su programa sea más fácil de leer, comprender y depurar.
- Las funciones pueden hacer que un programa sea más pequeño eliminando el código repetitivo. Más tarde, si realiza un cambio, solo tendrá que hacerlo en un lugar.
- La división de un programa largo en funciones le permite depurar las partes una a la vez y luego ensamblarlas en un todo de trabajo.
- Las funciones bien diseñadas suelen ser útiles para muchos programas. Una vez que escribes y depuras una, puedes reutilizarla.

A lo largo del resto del libro, a menudo usaremos una definición de función para explicar un concepto. Parte de la habilidad de crear y usar funciones es hacer que una función capture adecuadamente una idea como "encontrar el valor más pequeño en una lista de valores". Más adelante, te mostraremos el código que encuentra el más pequeño en una lista de valores y lo presentaremos como una función llamada `min` que toma una lista de valores como su argumento y devuelve el valor más pequeño de la lista.

Copyright 2026

Depurando

Si estás utilizando un editor de texto para escribir tus scripts, es posible que tengas problemas con los espacios y las tabulaciones. La mejor manera de evitar estos problemas es usar espacios exclusivamente (sin tabulaciones). La mayoría de los editores de texto que conocen Python lo hacen de forma predeterminada, pero algunos no.

Las tabulaciones y los espacios suelen ser invisibles, lo que los hace difíciles de depurar, así que trata de encontrar un editor que administre la sangría por ti.

Además, no olvides guardar tu programa antes de ejecutarlo. Algunos entornos de desarrollo lo hacen automáticamente, pero otros no. En ese caso, el programa que estás viendo en el editor de texto no es el mismo que el programa que estás ejecutando.

La depuración puede llevar mucho tiempo si continúas ejecutando el mismo programa incorrecto una y otra vez.

Asegúrate de que el código que estás viendo es el código que estás ejecutando. Si no estás seguro, coloca algo como `print("hola")` al principio del programa y vuelve a ejecutarlo. ¡Si no ves `hola`, no estás ejecutando el programa correcto!

Ejercicios

Ejercicio 4 ¿Cuál es el propósito de la palabra clave "def" en Python?

a) Es una jerga que significa "el siguiente código es realmente genial"
b) Indica el inicio de una función
c) Indica que la siguiente sección de código con sangría debe almacenarse para más adelante
d) `b` y `c` son verdaderas
e) Ninguna de las anteriores

Ejercicio 5 ¿Qué imprimirá el siguiente programa de Python?

```
def fred():
   print("Zap")

def jane():
```

```
    print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Ejercicio 6 reescribe su cálculo de pago con tiempo y medio para horas extras y crea una función llamada `computepay` que tome dos parámetros (`hours` y `rate`).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Ejercicio 7 reescribe el programa de calificación del capítulo anterior utilizando una función llamada `computegrade` que tome una puntuación como parámetro y devuelva una nota como una cadena.

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F
```

```
Program Execution:
```

```
Enter score: 0.95
A
```

```
Enter score: perfect
Bad score
```

```
Enter score: 10.0
Bad score
```

```
Enter score: 0.75
C
```

```
Enter score: 0.5
F
```

Ejecuta el programa repetidamente para probar los diferentes valores de entrada.

Revision #2
Created 2025-04-05 09:26:58 CEST by Javier Quintana
Updated 2025-04-05 09:27:34 CEST by Javier Quintana

Copyright 2026