2 Variables

Valores y tipos

Un **valor** es una de las cosas básicas con las que trabaja un programa, como una letra o un número. Los valores que hemos visto hasta ahora son `1`, `2` y "¡Hola, mundo!"

Estos valores pertenecen a diferentes tipos: `2` es un número entero (`int` de integer en inglés) y "¡Hola, mundo!" es una **cadena** (`string`), llamada así porque contiene una "cadena" de letras. Usted (y el intérprete) pueden identificar cadenas porque están entre comillas.

La declaración `print` también funciona para enteros. Usamos el comando `python` para iniciar el intérprete.

```
python
>>> print(4)
4
```

Si no estás seguro del tipo de valor que tiene, el intérprete te puede informar.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Las cadenas pertenecen al tipo `str` y los enteros pertenecen al tipo `int`. Los números con un punto decimal pertenecen a un tipo llamado `float`, porque estos números están representados en un formato llamado **punto flotante**.

```
>>> type(3.2)
<class 'float'>
```

Copyright 2026

¿Qué pasa con los valores como "17" y "3.2"? Parecen números, pero están entre comillas como cadenas.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Son cadenas.

Cuando escribe un entero grande, puede verse tentado a usar comas entre grupos de tres dígitos, como en `1,000,000`. Este no es un entero legal en Python, pero es legal:

```
>>> print(1,000,000)
1 0 0
```

Bueno, ¡eso no es lo que esperábamos en absoluto! Python interpreta `1,000,000` como una secuencia de enteros separados por comas, que imprime con espacios entre ellos.

Este es el primer ejemplo que hemos visto de un error semántico: el código se ejecuta sin producir un mensaje de error, pero no hace lo "correcto".

Variables

Una de las características más poderosas de un lenguaje de programación es la capacidad de manipular **variables**. Una variable es un nombre que se refiere a un valor.

Una **asignación** crea nuevas variables y les da valores:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

Este ejemplo hace tres asignaciones. El primero asigna una cadena a una nueva variable llamada `message`; el segundo asigna el número entero `17` a `n`; el tercero asigna el valor (aproximado) de **π** a `pi`.

Para mostrar el valor de una variable, puedes usar una declaración de impresión:

Copyright 2026

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

El tipo de una variable es el tipo del valor al que se refiere.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

Nombres de variables y palabras clave

Los programadores generalmente eligen nombres para sus variables que son significativos y documentan para qué se utiliza la variable.

Los nombres de las variables pueden ser arbitrariamente largos. Pueden contener letras y números, pero no pueden comenzar con un número. Es legal usar letras mayúsculas, pero es una buena idea comenzar los nombres de las variables con una letra minúscula (verás porqué más adelante).

El carácter de subrayado (`_`) puede aparecer en un nombre. A menudo se usa en nombres con varias palabras, como `my_name` o `airspeed_of_unladen_swallow`. Los nombres de las variables pueden comenzar con un carácter de subrayado, pero generalmente evitamos hacer esto a menos que estemos escribiendo el código de la biblioteca para que otros lo utilicen.

Si le das a una variable un nombre ilegal, obtendrás un error de sintaxis:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
```

Copyright 2026

```
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` es ilegal porque comienza con un número. `more@` es ilegal porque contiene un carácter ilegal, `@`. Pero, ¿qué pasa con `class`?

Resulta que `class` es una de las **palabras reservadas** de Python. El intérprete usa palabras clave para reconocer la estructura del programa, y no se pueden usar como nombres de variables.

Python reserva 33 palabras clave:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

Es posible que desees mantener esta lista a mano. Si el intérprete se queja de uno de sus nombres de variables y no sabe porqué, vea si está en esta lista.

Declaraciones

Una **instrucción** es una unidad de código que el intérprete de Python puede ejecutar. Hemos visto dos tipos de declaraciones: declaraciones de impresión (print()) y una asignación.

Cuando escribe una declaración en modo interactivo, el intérprete lo ejecuta y muestra el resultado, si lo hay.

Un script usualmente contiene una secuencia de sentencias. Si hay más de una declaración, los resultados aparecen uno por uno a medida que se ejecutan las declaraciones.

Por ejemplo, el guión.

```
print(1)
x = 2
```

Copyright 2026

```
print(x)
```

produce la salida

```
1
2
```

La sentencia de asignación no produce salida.

Operadores y operandos

Los operadores son símbolos especiales que representan cálculos como la suma y la multiplicación. Los valores a los que se aplica el operador se denominan **operandos**.

Los operadores `+`, `-`, `*`, `/` y `**` realizan suma, resta, multiplicación, división y exponenciación, como en los siguientes ejemplos:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

Ha habido un cambio en el operador de división entre Python 2.x y Python 3.x. En Python 3.x, el resultado de esta división es un resultado de punto flotante:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

El operador de división en Python 2.0 dividiría dos enteros y truncaría el resultado a un entero:

```
>>> minute = 59
>>> minute/60
0
```

Para obtener la misma respuesta en Python 3.0 use división dividida (// integer).

En Python 3.0, la división de enteros funciona mucho más como cabría esperar si ingresara la expresión en una calculadora.

Copyright 2026

```
>>> minute = 59
>>> minute//60
0
```

Expresiones

Una **expresión** es una combinación de valores, variables y operadores. Un valor en sí mismo se considera una expresión, y también lo es una variable, por lo que las siguientes son todas expresiones legales (asumiendo que a la variable `x` se le ha asignado un valor):

```
17
x
x + 17
```

Si escribes una expresión en modo interactivo, el intérprete **la evalúa** y muestra el resultado:

```
>>> 1 + 1
2
```

¡Pero en un guión, una expresión por sí misma no hace nada! Esta es una fuente común de confusión para los principiantes.

Ejercicio 1: escriba las siguientes declaraciones en el intérprete de Python para ver qué hacen:

```
5
x = 5
x + 1
```

Orden de operaciones

Cuando aparece más de un operador en una expresión, el orden de evaluación depende de las **reglas de precedencia**. Para los operadores matemáticos, Python sigue la convención matemática. El acrónimo **PEMDAS** es una forma útil de recordar las reglas:

- **P**arentheses tienen la mayor prioridad y se pueden usar para forzar a una expresión a evaluar en el orden que desee. Como las expresiones entre paréntesis se evalúan primero, `2 * (3-1)` es 4, y `(1 + 1) ** (5-2)` es 8. También puede usar paréntesis para hacer que una expresión sea más fácil de leer , como en `(minuto * 100)/60`, incluso si no cambia el resultado.
- **E**xponentiation tiene la siguiente prioridad más alta, por lo que `2**1+1` es 3, no 4, y `3*1**3` es 3, no 27.
- **M**ultiplication y **D**ivision tienen la misma precedencia, que es mayor que **A**ddition y **S**ubtraction, que también tienen la misma precedencia. Entonces `2*3-1` es 5, no 4, y `6+4/2` es 8.0, no 5.
- Los operadores con la misma precedencia son evaluados de izquierda a derecha. Entonces, la expresión `5-3-1` es 1, no 3, porque el `5-3` sucede primero y luego se le resta `1` de `2`.

En caso de duda, pon siempre paréntesis en tus expresiones para asegurarte de que los cálculos se realicen en el orden que deseas.

Operador de módulo

El **operador de módulo** funciona en enteros y produce el resto cuando el primer operando se divide por el segundo. En Python, el operador de módulo es un signo de porcentaje (`%`). La sintaxis es la misma que para otros operadores:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Así que 7 dividido por 3 es 2 con 1 sobrante.

El operador de módulo resulta ser sorprendentemente útil. Por ejemplo, puede verificar si un número es divisible por otro: si `x%y` es cero, entonces `x` es divisible por `y`.

También puede extraer el dígito o dígitos más a la derecha de un número. Por ejemplo, `x%10` produce el dígito más a la derecha de `x` (en base 10). De manera similar, `x%100` produce los dos últimos dígitos.

Copyright 2026

Operaciones con cadenas

El operador `+` trabaja con cadenas, pero no es una suma en el sentido matemático. En su lugar, realiza **concatenación**, lo que significa unir las cadenas uniéndolas de extremo a extremo. Por ejemplo:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

La salida de este programa es `100150`.

Preguntando al usuario por la entrada

A veces nos gustaría tomar el valor para una variable del usuario a través de su teclado. Python proporciona una función llamada `input` que obtiene información desde el teclado[1]. Cuando se llama a esta función, el programa se detiene y espera a que el usuario escriba algo. Cuando el usuario presiona `Return` o `Enter`, el programa se reanuda y `input` devuelve lo que el usuario escribió como una cadena.

```
>>> input = input()
Some silly stuff
>>> print(input)
Some silly stuff
```

Antes de recibir información del usuario, es una buena idea imprimir un mensaje que indique al usuario qué debe ingresar. Puede pasar una cadena a `input` para que se muestre al usuario antes de hacer una pausa para ingresar:

Copyright 2026

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

La secuencia `\n` al final de la solicitud representa una **nueva línea**, que es un carácter especial que causa un salto de línea. Es por eso que la entrada del usuario aparece debajo del indicador.

Si esperas que el usuario escriba un número entero, puedes intentar convertir el valor de retorno a `int` utilizando la función `int()`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

Pero si el usuario escribe algo más que una cadena de dígitos, obtendrás un error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

Veremos cómo manejar este tipo de error más adelante.

Comentarios

A medida que los programas se hacen más grandes y más complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es difícil mirar un fragmento de código y descubrir qué está haciendo, o por qué.

Por esta razón, es una buena idea agregar notas a tus programas para explicar en lenguaje natural lo que está haciendo el programa. Estas notas se llaman **comentarios**, y en Python comienzan con el símbolo `#`:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

En este caso, el comentario aparece solo en una línea. También puedes poner comentarios al final de una línea:

```
percentage = (minute * 100) / 60     # percentage of an hour
```

Todo desde el `\#` hasta el final de la línea se ignora. No tiene efecto en el programa.

Los comentarios son más útiles cuando documentan características no obvias del código. Es razonable suponer que el lector puede averiguar lo que hace el código; es mucho más útil explicar por qué.

Este comentario es redundante e inútil:

```
v = 5     # assign 5 to v
```

Este comentario contiene información útil que no está en el código:

```
v = 5     # velocity in meters/second.
```

Los buenos nombres de variables pueden reducir la necesidad de comentarios, pero los nombres largos pueden hacer que las expresiones complejas sean difíciles de leer, por lo que hay que buscar un punto de equilibrio.

Eligiendo nombres de variables mnemónicas

Siempre que sigas las reglas simples de los nombres de variables y evites las palabras reservadas, tienes muchas opciones al nombrar tus variables. Al principio, esta opción puede ser confusa tanto cuando lees un programa como cuando escribes tus propios programas. Por ejemplo, los siguientes

tres programas son idénticos en términos de lo que logran, pero son muy diferentes cuando los lees y tratas de entenderlos.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

El intérprete de Python ve a estos tres programas como **exactamente iguales** pero los humanos ven y entienden estos programas de manera muy diferente. Los humanos entenderán más rápidamente la **intención** del segundo programa porque el programador ha elegido nombres de variables que reflejan su intención con respecto a qué datos se almacenarán en cada variable.

Llamamos a estos nombres de variables sabiamente elegidos "nombres de variables mnemónicas". La palabra **mnemotécnia**[2] significa "ayuda de memoria". Elegimos nombres de variables mnemónicas para ayudarnos a recordar porqué creamos la variable en primer lugar.

Si bien todo esto suena bien, y es una muy buena idea usar nombres de variables mnemónicas, los nombres de variables mnemotécnicas pueden interferir en la capacidad de un programador principiante para analizar y comprender el código. Esto se debe a que los programadores principiantes aún no han memorizado las palabras reservadas (solo hay 33 de ellas) y, a veces, las variables con nombres que son demasiado descriptivas comienzan a parecer parte del lenguaje y no solo los nombres de variables bien elegidos.

Eche un vistazo rápido al siguiente código de ejemplo de Python que recorre algunos datos. Cubriremos los bucles pronto, pero por ahora trataremos de descifrar lo que esto significa:

```
for word in words:
    print(word)
```

¿Que está sucediendo aquí? ¿Cuáles de los tokens (para, palabra, en, etc.) son palabras reservadas y cuáles son solo nombres de variables? ¿Python entiende en un nivel fundamental la noción de palabras? Los programadores principiantes tienen problemas para separar qué partes del código **deben** ser las mismas que en este ejemplo y qué partes del código son simplemente elecciones hechas por el programador.

El siguiente código es equivalente al código anterior:

```
for slice in pizza:
    print(slice)
```

Es más fácil para el programador principiante mirar este código y saber qué partes son palabras reservadas definidas por Python y qué partes son simplemente nombres de variables elegidos por el programador. Es bastante claro que Python no tiene una comprensión fundamental de la pizza y las rebanadas y el hecho de que una pizza consiste en un conjunto de una o más rebanadas.

Pero si nuestro programa realmente trata de leer datos y buscar palabras en los datos, `pizza` y `slice` son nombres de variables muy poco mnemotécnicos. Elegirlos como nombres de variables distrae del significado del programa.

Después de un período de tiempo bastante corto, conocerá las palabras reservadas más comunes y comenzará a ver las palabras reservadas saltando hacia usted:

for word **in** words: **print**(word)

Las partes del código definidas por Python (`for`, `in`, `print` y `:`) están en negrita y las variables elegidas por el programador (`word` y `words`) no están en negrita . Muchos editores de texto conocen la sintaxis de Python y colorearán las palabras reservadas de manera diferente para darle pistas sobre cómo mantener separadas sus variables y palabras reservadas. Después de un tiempo, comenzarás a leer Python y determinar rápidamente qué es una variable y qué es una palabra reservada.

Depuración

En este punto, el error de sintaxis que probablemente cometas es un nombre de variable ilegal, como `class` y `yield`, que son palabras clave, o `odd~job` y `US$`, que contienen caracteres ilegales.

Si colocas un espacio en el nombre de una variable, Python cree que son dos operandos sin un operador:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

```
>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

Para los errores de sintaxis, los mensajes de error no ayudan mucho. Los mensajes más comunes son `SyntaxError: syntax invalid` y `SyntaxError: invalid token`, ninguno de los cuales es muy informativo.

El error de tiempo de ejecución que es más probable que cometa es un "use before def;" es decir, tratar de usar una variable antes de que haya asignado un valor. Esto puede suceder si escribes mal el nombre de una variable:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Los nombres de las variables distinguen entre mayúsculas y minúsculas, por lo que `LaTeX` no es lo mismo que `latex`.

En este punto, la causa más probable de un error semántico es el orden de las operaciones. Por ejemplo, para evaluar `1/2π`, puede tener la tentación de escribir

```
>>> 1.0/2.0 * pi
```

Pero la división ocurre primero, así que obtendrías `π/2`, ¡que no es lo mismo! Python no tiene forma de saber lo que querías escribir, por lo que en este caso no recibes un mensaje de error; acabas de obtener la respuesta incorrecta.

Ejercicios

Ejercicio 2: escribe un programa que use `input` para pedirle a un usuario su nombre y luego le de la bienvenida.

```
Enter your name: Chuck
Hello Chuck
```

Ejercicio 3: escribe un programa para pedirle al usuario las horas y la tarifa por hora para calcular el pago bruto.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

No nos preocuparemos por asegurarnos de que nuestra paga tenga exactamente dos dígitos después del lugar decimal por ahora. Si lo deseas, puede jugar con la función Python `round` incorporada para redondear correctamente la paga resultante a dos decimales.

Ejercicio 4: Supongamos que ejecutamos las siguientes instrucciones de asignación:

```
width = 17
height = 12.0
```

Para cada una de las siguientes expresiones, escriba el valor de la expresión y el tipo (del valor de la expresión).

1. `width//2`
2. `width/ 2.0`
3. `height/ 3`
4. `1+2\*5`

Use el intérprete de Python para verificar sus respuestas.

Ejercicio 5: escribe un programa que solicite al usuario una temperatura en grados Celsius, convierta la temperatura a Fahrenheit e imprima la temperatura convertida.

> [1]. En Python 2.0, esta función se denominó `raw_input`. ↩

> [2]. Consulte http://en.wikipedia.org/wiki/Mnemonic para obtener una descripción ampliada de la palabra "mnemonic". ↩

Copyright 2026

Revision #2
Created 2025-04-05 08:53:37 CEST by Javier Quintana
Updated 2025-04-05 08:58:26 CEST by Javier Quintana