

Números y operaciones aritméticas elementales

En esta lección se tratan los tipos de datos numéricos y las operaciones aritméticas elementales en Python.


Tipos de datos numéricos: enteros, decimales y números complejos

Enteros y decimales

Python distingue entre números enteros y decimales. Al escribir un número decimal, el separador entre la parte entera y la parte decimal es un punto.

```
>>> 3
3
>>> 4.5
4.5
```

Nota:

-  Si se escribe una coma como separador entre la parte entera y la decimal, Python no lo entiende como separador, sino como una pareja de números (concretamente, lo entiende como una tupla de dos elementos, un tipo de datos que se comenta en la [lección sobre tuplas](#)).

```
>>> 3,5
(3, 5)
```

Si se escribe un número con parte decimal 0, Python considera el número como número decimal.

```
>>> 3.0
3.0
```

Se puede escribir un número decimal sin parte entera, pero lo habitual es escribir siempre la parte entera:

```
>>> .3
0.3
```

Números complejos

Python puede hacer cálculos con número complejos. La parte imaginaria se acompaña de la letra "j".

```
>>> 1 + 1j + 2 + 3j
(3+4j)
```

```
>>> (1+1j) * 1j
(-1+1j)
```

La letra "j" debe ir acompañada siempre de un número y unida a él.

```
>>> 1 + j
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    1 + j
NameError: name 'j' is not defined
>>> 1 + 1 j
SyntaxError: invalid syntax
>>> 1 + 1j
(-1+1j)
```

Las cuatro operaciones básicas

Las cuatro operaciones aritméticas básicas son la suma (+), la resta (-), la multiplicación (*) y la división (/).

Al hacer operaciones en las que intervienen números enteros y decimales, el resultado es siempre decimal. En el caso de que el resultado no tenga parte decimal, Python escribe 0 como parte decimal para indicar que el resultado es un número decimal:

```
>>> 4.5 * 3
13.5
>>> 4.5 * 2
9.0
```

Al sumar, restar o multiplicar números enteros, el resultado es entero.

```
>>> 1 + 2
3
>>> 3 - 4
-1
>>> 5 * 6
30
```

Al dividir números enteros, el resultado es siempre decimal, aunque sea un número entero. Cuando Python escribe un número decimal, lo escribe siempre con parte decimal, aunque sea nula.

```
>>> 9 / 2
4.5
>>> 9 / 3
3.0
```

Dividir por cero genera un error:

```
>>> 5 / 0
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    5 / 0
ZeroDivisionError: division by zero
>>>
```



Al realizar operaciones con decimales, los resultados pueden presentar errores de redondeo:

```
>>> 100 / 3
33.33333333333336
```

Este error se debe a que Python almacena los números decimales en binario y pasar de decimal a binario provoca errores de redondeo, como se explica en el apartado [Representación de números decimales en binario](#). Es un error que sufren casi todos los lenguajes de programación. Si necesitamos precisión absoluta, debemos utilizar bibliotecas específicas.

Cuando en una fórmula aparecen varias operaciones, Python las efectúa aplicando las reglas usuales de prioridad de las operaciones (primero multiplicaciones y divisiones, después sumas y restas).

```
>>> 1 + 2 * 3
7
>>> 10 - 4 * 2
2
```

En caso de querer que las operaciones se realicen en otro orden, se deben utilizar paréntesis.

```
>>> (5+8) / (7-2)
2.6
```



Debido a los errores de redondeo, dos operaciones que debieran dar el mismo resultado pueden dar resultados diferentes:

```
>>> 4 * 3 / 5
2.4
>>> 4 / 5 * 3
2.4000000000000004
```

Se pueden escribir sumas y restas seguidas, pero no se recomienda hacerlo porque no es una notación habitual:

```
>>> 3 + - + 4
-1
>>> 3 + - + - 4
7
```

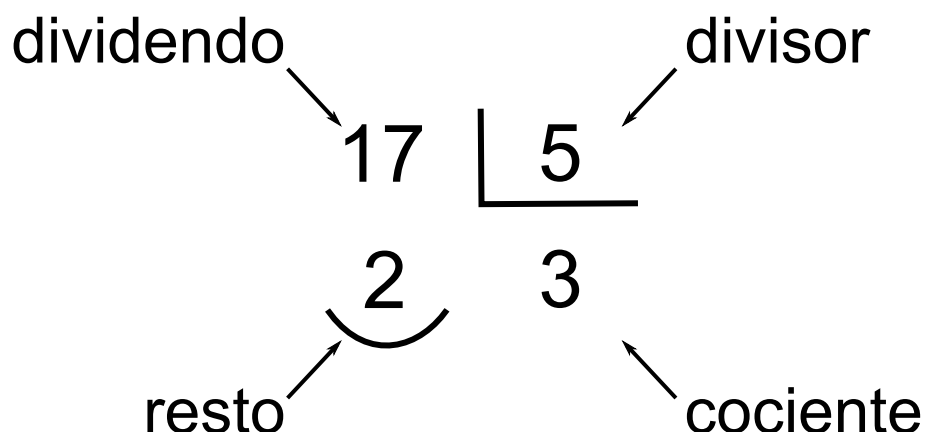
Lo que no se puede hacer es escribir multiplicaciones y divisiones seguidas:

```
>>> 3 * / 4
SyntaxError: invalid syntax
>>>
```

Cociente y resto de una división

El cociente y resto de una división están relacionados con el dividendo y divisor mediante la fórmula:

$$\text{dividendo} = \text{divisor} * \text{cociente} + \text{resto}$$



El cociente es siempre un número entero, pero el resto puede ser entero o decimal. En Python, el resto y el divisor tienen siempre el mismo signo y en valor absoluto (sin signo) el resto es siempre inferior al divisor.

Nota: Como se comenta en la lección [Detalles del lenguaje](#), el cociente y el divisor de una división en la que intervienen números enteros negativos no se calcula de la misma manera en los distintos lenguajes de programación.

Cociente de una división

El cociente de una división se calcula en Python con el operador `//`. El resultado es siempre un número entero, pero será de tipo entero o decimal dependiendo del tipo de los números empleados (en caso de ser decimal, la parte decimal es siempre cero). Por ejemplo:

```
>>> 10 // 3
3
>>> 10 // 4
2
>>> 20.0 // 7
2.0
>>> 20 // 6.0
3.0
```

El operador cociente `//` tiene la misma prioridad que la división:

```
>>> 26 // 5 / 2
2.5
>>> (26//5) / 2
2.5
```

```
>>> 26 // (5/2)
10.0
```

```
>>> 26 / 5 // 2
2.0
>>> (26/5) // 2
2.0
>>> 26 / (5//2)
13.0
```

Resto de una división

El resto de una división se calcula en Python con el operador %. El resultado tendrá tipo entero o decimal, de acuerdo con el resultado de la operación

```
>>> 10 % 3
1
>>> 10 % 4
2
>>> 10 % 5
0
>>> 10.5 % 3
1.5
```



Cuando el resultado es decimal, pueden aparecer los problemas de redondeo comentados anteriormente.

```
>>> 10.2 % 3
1.1999999999999999 # El resultado correcto es 1.2
>>> 10 % 4.2
1.5999999999999996 # El resultado correcto es 1.6
>>> 10.1 % 5.1
5.0 # Este resultado coincide con el resultado correcto
```

El operador resto % tiene la misma prioridad que la división:

```
>>> 26 % 5 / 2
0.5
>>> (26%5) / 2
0.5
>>> 26 % (5/2)
1.0
```

```
>>> 26 / 5 % 2
1.2000000000000002
>>> (26/5) % 2
1.2000000000000002
>>> 26 / (5%2)
26.0
```

La función integrada `divmod()`

La función integrada `divmod(x, y)` devuelve una tupla formada por el cociente y el resto de la división de `x` entre `y`.

```
>>> divmod(13, 4)
(3, 1)
```

Potencias y raíces

Las potencias se calculan con el operador **, teniendo en cuenta que $x ** y = x^y$.

Las potencias tienen prioridad sobre las multiplicaciones y divisiones.

Utilizando exponentes negativos o decimales se pueden calcular potencias inversas o raíces n-ésimas.

$$a^{-b} = \frac{1}{a^b} \quad a^{\frac{1}{b}} = \sqrt[b]{a}$$

```
>>> 2 ** 3
8
>>> 10 ** -4
0.0001          # Recuerde que a-b= 1/ab
>>> 9 ** 0.5
3.0             # Recuerde que a1/b es la raíz b-ésima de a
>>> (-1) ** 0.5 # Esto va a dar error porque es la raíz cuadrada de -1
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in ?
    (-1)**0.5 ValueError: negative number cannot be raised to a fractional power
>>> -9 ** 0.5   # Esto no da error porque hace primero la raíz y luego le pone el -
-3
```

También se pueden calcular potencias o raíces mediante la función integrada `pow(x,y)`. Si se da un tercer argumento, `pow(x, y, z)`, la función calcula primero x elevado a y después calcula el resto de la división por z.

```
>>> pow(2, 3)
8
>>> pow(4, 0.5)
2.0
>>> pow(2, 3, 5)
3
```

Redondear un número

Aunque no se puede dar una regla válida en todas las situaciones, normalmente es conveniente redondear el resultado de un cálculo cuando se muestra al usuario, sobre todo si tiene muchos decimales, para facilitar su lectura.

Lo que no se debe hacer nunca es redondear resultados intermedios que se vayan a utilizar en cálculos posteriores, porque el resultado final será diferente.

La función integrada `round()`

Para redondear un número (por ejemplo, cuando se muestra al usuario el resultado final de un cálculo), se puede utilizar la función integrada `round()`. La función integrada `round()` admite uno o dos argumentos numéricos.

- Si sólo hay un argumento, la función devuelve el argumento redondeado al entero más próximo:

```
>>> round(4.35)
4
>>> round(4.62)
5
>>> round(-4.35)
-4
>>> round(-4.62)
-5
```

- Si se escriben dos argumentos, siendo el segundo un número entero, la función integrada `round()` devuelve el primer argumento redondeado en la posición indicada por el segundo argumento.
 - Si el segundo argumento es positivo, el primer argumento se redondea con el número de decimales indicado:

```
>>> round(4.3527, 2)
4.35
>>> round(4.3527, 1)
4.4
>>> round(4.3527, 3)
4.353
```

Si se piden más decimales de los que tiene el número, se obtiene el primer argumento, sin cambios:

```
>>> round(4.3527, 7)
4.3527
>>> round(435, 2)
435
```

- Si el segundo argumento es 0, se redondea al entero más próximo, como cuando no se escribe segundo argumento, pero la diferencia es que el resultado es decimal y no entero.

```
>>> round(4.3527, 0)
4.0
>>> round(4.3527)
4
```

Si el primer argumento es un número entero, el resultado sigue siendo el mismo número entero:

```
>>> round(435, 0)
435
```

- Si el segundo argumento es negativo, se redondea a decenas, centenas, etc.

```
>>> round(43527, -1)
43530
>>> round(43527, -2)
43500
>>> round(43527, -3)
44000
>>> round(43527, -4)
40000
>>> round(43527, -5)
0
```

La función integrada `round()` redondea correctamente al número más próximo del orden de magnitud deseado (entero, décimas, decenas, centésimas, centenas, etc). El problema es cuando el número a redondear está justo en medio (por ejemplo, redondear 3.5 a entero, 4.85 a décimas, etc.). En Matemáticas se suele redondear siempre hacia arriba, pero en Python se sigue otro criterio:

- Cuando se redondea a enteros, decenas, centenas, etc., Python redondea de manera que la última cifra (la redondeada) sea par.

```
>>> round(3.5)
4
>>> round(4.5)
4
>>> round(5.5)
6
>>> round(6.5)
6
```

```
>>> round(450, -2)
400
>>> round(350, -2)
400
>>> round(250, -2)
200
>>> round(150, -2)
200
>>> round(50, -2)
0
```

- Cuando se redondea a décimas, centésimas, etc., Python redondea en unos casos para arriba y en otros para abajo, debido a la forma en que se representan internamente los números decimales (como se explica en el apartado [Representación de números decimales en binario](#)):

```
>>> round(3.15, 1)
3.1
>>> round(3.25, 1)
3.2
>>> round(3.35, 1)
3.4
>>> round(3.45, 1)
3.5
>>> round(3.55, 1)
3.5
>>> round(3.65, 1)
3.6
```



```
>>> round(0.315, 2)
0.32
>>> round(0.325, 2)
0.33
>>> round(0.335, 2)
0.34
>>> round(0.345, 2)
0.34
>>> round(0.355, 2)
0.35
>>> round(0.365, 2)
0.36
```

Este redondeo se debe a la forma en que Python representa internamente los números decimales. Es un problema que presentan la mayoría de lenguajes de programación y se explica en el apartado siguiente.

Redondear al entero anterior o posterior: las funciones **floor()** y **ceil()**

Para redondear un número al entero anterior o posterior, se pueden utilizar las funciones **floor()** y **ceil()**, que están incluidas en la biblioteca **math**. Estas funciones sólo admiten un argumento numérico y devuelven valores enteros.

Antes de utilizar estas funciones, hay que importarlas, o se generará un error.

```
>>> floor(5 / 2)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    floor(5 / 2)
NameError: name 'floor' is not defined
>>> ceil(5 / 2)
Traceback (most recent call last):
  File "<pyshell#0>", line 2, in <module>
    ceil(5 / 2)
NameError: name 'ceil' is not defined
```

```
>>> from math import floor
>>> floor(5 / 2)
2
>>> from math import ceil
>>> ceil(5 / 2)
3
```

Representación de números decimales en binario

En Python los números decimales se almacenan internamente en binario con 53 bits de precisión (en concreto, se trata del formato de coma flotante de doble precisión de la norma IEEE-754). Cuando un programa pide a Python un cálculo con números decimales, Python convierte esos números decimales a binario, realiza la operación en binario y convierte el resultado de nuevo en decimal para mostrárselo al usuario.

El problema es que muchos números decimales no se pueden representarse de forma exacta en binario, por lo que los resultados no pueden ser exactos. Eso explica, por ejemplo, los resultados del ejemplo anterior:

```
>>> round(3.45, 1)
3.5
>>> round(3.55, 1)
3.5
```

En el primer ejemplo, al convertir 3.45 a binario con 53 bits de precisión, el valor obtenido es realmente 3.45000000000000017763568394002504646778106689453125, es decir, ligeramente mayor que 3.45, por lo que al redondear con décimas, Python muestra el valor 3.5.

En el segundo ejemplo, al convertir 3.55 a binario con 53 bits de precisión, el valor obtenido es realmente 3.54999999999999982236431605997495353221893310546875, es decir, ligeramente inferior a 3.55, por lo que al redondear con décimas, Python muestra también el valor 3.5.

Nota:

- Para ver el valor que se obtiene al convertir un número decimal a binario se puede utilizar el tipo Decimal de la biblioteca decimal:

```
>>> from decimal import Decimal
>>> Decimal(3.45)
Decimal('3.45000000000000017763568394002504646778106689453125')
>>> Decimal(3.55)
Decimal('3.54999999999999982236431605997495353221893310546875')
```

El problema del redondeo se agudiza cuando se hacen operaciones, puesto que los errores pueden acumularse (aunque a veces se compensan y pasan desapercibidos).

En algunos casos extremos, el error es apreciable en cálculos muy sencillos:

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

En la mayoría de situaciones, estos errores no tienen consecuencias importantes en los resultados finales, pero si en una aplicación concreta se necesita total exactitud, se deben utilizar bibliotecas específicas. Python incluye la biblioteca [decimal](#) y existen bibliotecas como [mpmath](#) que admiten precisión arbitraria.

Para saber más:

- IEEE-754 [precisión simple](#), [precisión doble](#) (Wikipedia en inglés)
- Manual de Python 3: [limitaciones de los decimales en Python](#)
- Artículo [The perils of Floating Point](#), de Bruce M. Bush
- Artículo [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#), de David Goldberg
- Web [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)

Otras funciones integradas (*built-in functions*)

El intérprete de Python incorpora varias funciones integradas que realizan operaciones muy habituales.

Valor absoluto: `abs()`

La función integrada `abs()` calcula el valor absoluto de un número, es decir, el valor sin signo.

```
>>> abs(-6)
6
>>> abs(7)
7
```

Máximo: `max()`

La función integrada `max()` calcula el valor máximo de un conjunto de valores (numéricos o alfabéticos). En el caso de cadenas, el valor máximo corresponde al último valor en orden alfabético, sin importar la longitud de la cadena.

```
>>> max(4, 5, -2, 8, 3.5, -10)
8
>>> max("David", "Alicia", "Tomás", "Emilio")
'Tomás'
```

Las vocales acentuadas, la letra ñ o ç se consideran posteriores al resto de vocales y consonantes

```
>>> max("Ángeles", "Roberto")
'Ángeles'
```

Mínimo: `min()`

La función integrada `min()` calcula el valor mínimo de un conjunto de valores (numéricos o alfabéticos). En el caso de cadenas, el valor mínimo corresponde al primer valor en orden alfabético, sin importar la longitud de la cadena.

```
>>> min(4, 5, -2, 8, 3.5, -10)
-10
>>> min("David", "Alicia", "Tomás", "Emilio")
'Alicia'
```

Las vocales acentuadas, la letra ñ o ç se consideran posteriores al resto de vocales y consonantes

```
>>> min("Ángeles", "Roberto")
'Roberto'
```

Suma: `sum()`

La función integrada `sum()` calcula la suma de un conjunto de valores. El conjunto de valores debe ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario).

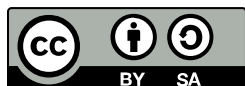
```
>>> sum((1, 2, 3, 4, 5))
15
>>> sum([1, 2, 3, 4, 5])
15
>>> sum(range(6))
15
>>> sum({1, 2, 3, 4, 5})
15
```

Ordenación: `sorted()`

La función integrada `sorted()` ordena un conjunto de valores. El conjunto de valores debe ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario). El conjunto de valores no se modifica, la función devuelve una lista con los elementos ordenados.

```
>>> sorted((10, 2, 8, -3, 6))
[-3, 2, 6, 8, 10]
>>> sorted([10, 2, 8, -3, 6])
[-3, 2, 6, 8, 10]
>>> sorted({10, 2, 8, -3, 6})
[-3, 2, 6, 8, 10]
>>> sorted(("David", "Alicia", "Tomás", "Emilio"))
['Alicia', 'David', 'Emilio', 'Tomás']
>>> sorted(["David", "Ángeles", "Tomás", "Óscar", "Emilio"])
['David', 'Emilio', 'Tomás', 'Ángeles', 'Óscar']
```

Última modificación de esta página: 25 de febrero de 2018



Esta página forma parte del curso [Introducción a la programación con Python](#) por [Bartolomé Sintes Marco](#)

que se distribuye bajo una [Licencia Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional \(CC BY-SA 4.0\)](#).