Simple Mathematical Operators

All Versions

Python does common mathematical operators on its own, including integer and float division, multiplication, exponentiation, addition, and subtraction. The math module (included in all standard Python versions) offers expanded functionality like trigonometric functions, root operations, logarithms, and many more.

Examples

Division

Python does integer division when both operands are integers. The behavior of Python's division operators have changed from Python 2.x and 3.x (see also Integer Division).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x ≤2.7

In Python 2 the result of the ' / ' operator depends on the type of the numerator and denominator.

```
a / b                # = 1

a / c                # = 1.5

d / b                # = -2

b / a                # = 0

d / e                # = -1
```

Note that because both `a` and `b` are `int` s, the result is an `int` .

The result is always rounded down (floored).

Because `c` is a float, the result of `a / c` is a `float` .

You can also use the operator module:

```
import operator        # the operator module provides 2-argument arithmetic functions
operator.div(a, b)     # = 1
operator.__div__(a, b) # = 1
```

Python 2.x ≥2.2

What if you want float division:

Recommended:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                  # = 1.5
a // b                 # = 1
```

Okay (if you don't want to apply to the whole module):

```
a / (b * 1.0)          # = 1.5
1.0 * a / b            # = 1.5
a / b * 1.0            # = 1.0    (careful with order of operations)

from operator import truediv
truediv(a, b)          # = 1.5
```

Not recommended (may raise TypeError, eg if argument is complex):

```
float(a) / b           # = 1.5
a / float(b)           # = 1.5
```

Python 2.x ≥2.2

The ' // ' operator in Python 2 forces floored division regardless of type.

```
a // b               # = 1
a // c               # = 1.0
```

Python 3.x ≥3.0

In Python 3 the `/` operator performs 'true' division regardless of types. The `//` operator performs floor division and maintains type.

```
a / b                # = 1.5
e / b                # = 5.0
```

```
a / b                  # = 5.0
a // b                 # = 1
a // c                 # = 1.0

import operator            # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)     # = 1.5
operator.floordiv(a, b)    # = 1
operator.floordiv(a, c)    # = 1.0
```

Possible combinations (builtin types):

- int and int (gives an int in Python 2 and a float in Python 3)
- int and float (gives a float)
- int and complex (gives a complex)
- float and float (gives a float)
- float and complex (gives a complex)
- complex and complex (gives a complex)

See PEP 238 for more information.

Addition

```
a, b = 1, 2

# Using the "+" operator:
a + b                  # = 3

# Using the "in-place" "+=" operator to add and assign:
a += b                 # a = 3 (equivalent to a = a + b)

import operator        # contains 2 argument arithmetic functions for the examples

operator.add(a, b)     # = 5  since a is set to 3 right before this line

# The "+=" operator is equivalent to:
a = operator.iadd(a, b)    # a = 5 since a is set to 3 right before this line
```

Possible combinations (builtin types):

- int and int (gives an int)
- int and float (gives a float)
- int and complex (gives a complex)
- float and float (gives a float)
- float and complex (gives a complex)
- complex and complex (gives a complex)

Note: the + operator is also used for concatenating strings, lists and tuples:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Exponentation

```
a, b = 2, 3

(a ** b)               # = 8
pow(a, b)              # = 8

import math
math.pow(a, b)         # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b)     # = 8
```

Another difference between the built-in pow and math.pow is that the built-in pow can accept three arguments:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)           # 0, calculates (2 ** 3) % 2, but as per Python docs,
                       #    does so more efficiently
```

Special functions

The function `math.sqrt(x)` calculates the square root of `x`.

```
import math
import cmath
c = 4
math.sqrt(c)          # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)         # = (2+0j) (always complex)
```

To compute other roots, such as a cube root, raise the number to the reciprocal of the degree of the root. This could be done with any of the exponential functions or operator.

```
 import math
 x = 8
 math.pow(x, 1/3) # evaluates to 2.0
 x**(1/3) # evaluates to 2.0
```

The function `math.exp(x)` computes `e ** x`.

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

The function `math.expm1(x)` computes `e ** x - 1`. When `x` is small, this gives significantly better precision than `math.exp(x) - 1`.

```
math.expm1(0)       # 0.0

math.exp(1e-6) - 1  # 1.0000004999621837e-06
math.expm1(1e-6)    # 1.0000005000001665e-06
# exact result      # 1.0000005000001666667083333416666...
```

Trigonometric Functions

```
a, b = 1, 2

import math

math.sin(a)  # returns the sine of 'a' in radians
# Out: 0.8414709848078965

math.cosh(b)  # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314

math.atan(math.pi)  # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115

math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

> Note that `math.hypot(x, y)` is also the length of the vector (or Euclidean distance) from the origin `(0, 0)` to the point `(x, y)`.
>
> To compute the Euclidean distance between two points `(x1, y1)` & `(x2, y2)` you can use `math.hypot` as follows
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

To convert from radians -> degrees and degrees -> radians respectively use `math.degrees` and `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Inplace Operations

It is common within applications to need to have code like this :

```
a = a + 1
```

or

```
a = a * 2
```

There is an effective shortcut for these in place operations :

```
a += 1
# and
a *= 2
```

Any mathematic operator can be used before the '=' character to make an inplace operation :

- `-=` decrement the variable in place
- `+=` increment the variable in place
- `*=` multiply the variable in place
- `/=` divide the variable in place
- `//=` floor divide the variable in place # Python 3
- `%=` return the modulus of the variable in place
- `**=` raise to a power in place

Other in place operators exist for the bitwise operators (`^`, `|` etc)

Logarithms

By default, the `math.log` function calculates the logarithm of a number, base e. You can optionally specify a base as the second argument.

```
import math
import cmath

math.log(5)         # = 1.6094379124341003
# optional base argument. Default is math.e
math.log(5, math.e) # = 1.6094379124341003
cmath.log(5)        # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (always returns float)
cmath.log(1000, 10)  # (3+0j)
```

Special variations of the `math.log` function exist for different bases.

```
# Logarithm base e - 1 (higher precision for low values)
math.log1p(5)       # = 1.791759469228055

# Logarithm base 2
math.log2(8)        # = 3.0

# Logarithm base 10
math.log10(100)     # = 2.0
cmath.log10(100)    # = (2+0j)
```

Modulus

Like in many other languages, Python uses the `%` operator for calculating modulus.

```
3 % 4     # 3
10 % 2    # 0
6 % 4     # 2
```

Or by using the `operator` module:

```
import operator

operator.mod(3 , 4)     # 3
operator.mod(10 , 2)    # 0
operator.mod(6 , 4)     # 2
```

You can also use negative numbers.

```
-9 % 7     # 5
9 % -7     # -5
-9 % -7    # -2
```

If you need to find the result of integer division and modulus, you can use the `divmod` function as a shortcut:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

```
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Multiplication

```
a, b = 2, 3

a * b                 # = 6

import operator
operator.mul(a, b)    # = 6
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Note: The `*` operator is also used for repeated concatenation of strings, lists, and tuples:

```
3 * 'ab'  # = 'abababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

Subtraction

```
a, b = 1, 2

# Using the "-" operator:
b - a                 # = 1


import operator       # contains 2 argument arithmetic functions
operator.sub(b, a)    # = 1
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Syntax

Parameters

Remarks

Numerical types and their metaclasses

The `numbers` module contains the abstract metaclasses for the numerical types:

subclasses	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	numbers.Complex
bool	✓	✓	✓	✓	✓
int	✓	✓	✓	✓	✓
fractions.Fr					

subclasses	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	numbers.Complex
fractions.Fraction	✓	—	✓	✓	✓
float	✓	—	—	✓	✓
complex	✓	—	—	—	✓
decimal.Decimal	✓	—	—	—	—