10.3. operator — Standard operators as functions

Source code: Lib/operator.py

The operator module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, operator.add(x, y) is equivalent to the expression x+y. The function names are those used for special class methods; variants without leading and trailing \_\_ are also provided for convenience.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

operator.lt(a, b)operator.le(a, b)operator.eq(a, b)operator.ne(a, b)operator.ge(a, b)operator.gt(a, b)operator.\_\_lt\_\_(a, b)operator.\_\_le\_\_(a, b)operator.\_\_eq\_\_(a, b)operator.\_\_ne\_\_(a, b)operator.\_\_ge\_\_(a, b)operator.\_\_gt\_\_(a, b)

Perform “rich comparisons” between a and b. Specifically, lt(a, b) is equivalent to a < b, le(a, b) is equivalent to a <= b, eq(a, b) is equivalent to a == b, ne(a, b) is equivalent to a != b, gt(a, b) is equivalent to a > b and ge(a, b) is equivalent to a >= b. Note that these functions can return any value, which may or may not be interpretable as a Boolean value. See Comparisons for more information about rich comparisons.

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

operator.not\_(obj)operator.\_\_not\_\_(obj)

Return the outcome of not obj. (Note that there is no \_\_not\_\_() method for object instances; only the interpreter core defines this operation. The result is affected by the \_\_bool\_\_() and \_\_len\_\_() methods.)

operator.truth(obj)

Return True if obj is true, and False otherwise. This is equivalent to using the bool constructor.

operator.is\_(a, b)

Return a is b. Tests object identity.

operator.is\_not(a, b)

Return a is not b. Tests object identity.

The mathematical and bitwise operations are the most numerous:

operator.abs(obj)operator.\_\_abs\_\_(obj)

Return the absolute value of obj.

operator.add(a, b)operator.\_\_add\_\_(a, b)

Return a + b, for a and b numbers.

operator.and\_(a, b)operator.\_\_and\_\_(a, b)

Return the bitwise and of a and b.

operator.floordiv(a, b)operator.\_\_floordiv\_\_(a, b)

Return a // b.

operator.index(a)operator.\_\_index\_\_(a)

Return a converted to an integer. Equivalent to a.\_\_index\_\_().

operator.inv(obj)operator.invert(obj)operator.\_\_inv\_\_(obj)operator.\_\_invert\_\_(obj)

Return the bitwise inverse of the number obj. This is equivalent to ~obj.

operator.lshift(a, b)operator.\_\_lshift\_\_(a, b)

Return a shifted left by b.

operator.mod(a, b)operator.\_\_mod\_\_(a, b)

Return a % b.

operator.mul(a, b)operator.\_\_mul\_\_(a, b)

Return a \* b, for a and b numbers.

operator.neg(obj)operator.\_\_neg\_\_(obj)

Return obj negated (-obj).

operator.or\_(a, b)operator.\_\_or\_\_(a, b)

Return the bitwise or of a and b.

operator.pos(obj)operator.\_\_pos\_\_(obj)

Return obj positive (+obj).

operator.pow(a, b)operator.\_\_pow\_\_(a, b)

Return a \*\* b, for a and b numbers.

operator.rshift(a, b)operator.\_\_rshift\_\_(a, b)

Return a shifted right by b.

operator.sub(a, b)operator.\_\_sub\_\_(a, b)

Return a - b.

operator.truediv(a, b)operator.\_\_truediv\_\_(a, b)

Return a / b where 2/3 is .66 rather than 0. This is also known as “true” division.

operator.xor(a, b)operator.\_\_xor\_\_(a, b)

Return the bitwise exclusive or of a and b.

Operations which work with sequences (some of them with mappings too) include:

operator.concat(a, b)operator.\_\_concat\_\_(a, b)

Return a + b for a and b sequences.

operator.contains(a, b)operator.\_\_contains\_\_(a, b)

Return the outcome of the test b in a. Note the reversed operands.

operator.countOf(a, b)

Return the number of occurrences of b in a.

operator.delitem(a, b)operator.\_\_delitem\_\_(a, b)

Remove the value of a at index b.

operator.getitem(a, b)operator.\_\_getitem\_\_(a, b)

Return the value of a at index b.

operator.indexOf(a, b)

Return the index of the first of occurrence of b in a.

operator.setitem(a, b, c)operator.\_\_setitem\_\_(a, b, c)

Set the value of a at index b to c.

Example: Build a dictionary that maps the ordinals from 0 to 255 to their character equivalents.

>>>>>> d = {}

>>> keys = range(256)

>>> vals = map(chr, keys)

>>> map(operator.setitem, [d]\*len(keys), keys, vals)

operator.length\_hint(obj, default=0)

Return an estimated length for the object o. First try to return its actual length, then an estimate using object.\_\_length\_hint\_\_(), and finally return the default value.

New in version 3.4.

The operator module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for map(), sorted(), itertools.groupby(), or other functions that expect a function argument.

operator.attrgetter(attr)operator.attrgetter(\*attrs)

Return a callable object that fetches attr from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

•After f = attrgetter('name'), the call f(b) returns b.name.

•After f = attrgetter('name', 'date'), the call f(b) returns (b.name, b.date).

•After f = attrgetter('name.first', 'name.last'), the call f(b) returns (b.name.first, b.name.last).

Equivalent to:

def attrgetter(\*items):

if any(not isinstance(item, str) for item in items):

raise TypeError('attribute name must be a string')

if len(items) == 1:

attr = items[0]

def g(obj):

return resolve\_attr(obj, attr)

else:

def g(obj):

return tuple(resolve\_attr(obj, attr) for attr in items)

return g

def resolve\_attr(obj, attr):

for name in attr.split("."):

obj = getattr(obj, name)

return obj

operator.itemgetter(item)operator.itemgetter(\*items)

Return a callable object that fetches item from its operand using the operand’s \_\_getitem\_\_() method. If multiple items are specified, returns a tuple of lookup values. For example:

•After f = itemgetter(2), the call f(r) returns r[2].

•After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3]).

Equivalent to:

def itemgetter(\*items):

if len(items) == 1:

item = items[0]

def g(obj):

return obj[item]

else:

def g(obj):

return tuple(obj[item] for item in items)

return g

The items can be any type accepted by the operand’s \_\_getitem\_\_() method. Dictionaries accept any hashable value. Lists, tuples, and strings accept an index or a slice:

>>>>>> itemgetter(1)('ABCDEFG')

'B'

>>> itemgetter(1,3,5)('ABCDEFG')

('B', 'D', 'F')

>>> itemgetter(slice(2,None))('ABCDEFG')

'CDEFG'

Example of using itemgetter() to retrieve specific fields from a tuple record:

>>>>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]

>>> getcount = itemgetter(1)

>>> list(map(getcount, inventory))

[3, 2, 5, 1]

>>> sorted(inventory, key=getcount)

[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]

operator.methodcaller(name[, args...])

Return a callable object that calls the method name on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

•After f = methodcaller('name'), the call f(b) returns b.name().

•After f = methodcaller('name', 'foo', bar=1), the call f(b) returns b.name('foo', bar=1).

Equivalent to:

def methodcaller(name, \*args, \*\*kwargs):

def caller(obj):

return getattr(obj, name)(\*args, \*\*kwargs)

return caller

10.3.1. Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the operator module.

Operation

Syntax

Function

Addition a + b add(a, b)

Concatenation seq1 + seq2 concat(seq1, seq2)

Containment Test obj in seq contains(seq, obj)

Division a / b truediv(a, b)

Division a // b floordiv(a, b)

Bitwise And a & b and\_(a, b)

Bitwise Exclusive Or a ^ b xor(a, b)

Bitwise Inversion ~ a invert(a)

Bitwise Or a | b or\_(a, b)

Exponentiation a \*\* b pow(a, b)

Identity a is b is\_(a, b)

Identity a is not b is\_not(a, b)

Indexed Assignment obj[k] = v setitem(obj, k, v)

Indexed Deletion del obj[k] delitem(obj, k)

Indexing obj[k] getitem(obj, k)

Left Shift a << b lshift(a, b)

Modulo a % b mod(a, b)

Multiplication a \* b mul(a, b)

Negation (Arithmetic) - a neg(a)

Negation (Logical) not a not\_(a)

Positive + a pos(a)

Right Shift a >> b rshift(a, b)

Slice Assignment seq[i:j] = values setitem(seq, slice(i, j), values)

Slice Deletion del seq[i:j] delitem(seq, slice(i, j))

Slicing seq[i:j] getitem(seq, slice(i, j))

String Formatting s % obj mod(s, obj)

Subtraction a - b sub(a, b)

Truth Test obj truth(obj)

Ordering a < b lt(a, b)

Ordering a <= b le(a, b)

Equality a == b eq(a, b)

Difference a != b ne(a, b)

Ordering a >= b ge(a, b)

Ordering a > b gt(a, b)

10.3.2. Inplace Operators

Many operations have an “in-place” version. Listed below are functions providing a more primitive access to in-place operators than the usual syntax does; for example, the statement x += y is equivalent to x = operator.iadd(x, y). Another way to put it is to say that z = operator.iadd(x, y) is equivalent to the compound statement z = x; z += y.

In those examples, note that when an in-place method is called, the computation and assignment are performed in two separate steps. The in-place functions listed below only do the first step, calling the in-place method. The second step, assignment, is not handled.

For immutable targets such as strings, numbers, and tuples, the updated value is computed, but not assigned back to the input variable:

>>>>>> a = 'hello'

>>> iadd(a, ' world')

'hello world'

>>> a

'hello'

For mutable targets such as lists and dictionaries, the inplace method will perform the update, so no subsequent assignment is necessary:

>>>>>> s = ['h', 'e', 'l', 'l', 'o']

>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])

['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

>>> s

['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

operator.iadd(a, b)operator.\_\_iadd\_\_(a, b)

a = iadd(a, b) is equivalent to a += b.

operator.iand(a, b)operator.\_\_iand\_\_(a, b)

a = iand(a, b) is equivalent to a &= b.

operator.iconcat(a, b)operator.\_\_iconcat\_\_(a, b)

a = iconcat(a, b) is equivalent to a += b for a and b sequences.

operator.ifloordiv(a, b)operator.\_\_ifloordiv\_\_(a, b)

a = ifloordiv(a, b) is equivalent to a //= b.

operator.ilshift(a, b)operator.\_\_ilshift\_\_(a, b)

a = ilshift(a, b) is equivalent to a <<= b.

operator.imod(a, b)operator.\_\_imod\_\_(a, b)

a = imod(a, b) is equivalent to a %= b.

operator.imul(a, b)operator.\_\_imul\_\_(a, b)

a = imul(a, b) is equivalent to a \*= b.

operator.ior(a, b)operator.\_\_ior\_\_(a, b)

a = ior(a, b) is equivalent to a |= b.

operator.ipow(a, b)operator.\_\_ipow\_\_(a, b)

a = ipow(a, b) is equivalent to a \*\*= b.

operator.irshift(a, b)operator.\_\_irshift\_\_(a, b)

a = irshift(a, b) is equivalent to a >>= b.

operator.isub(a, b)operator.\_\_isub\_\_(a, b)

a = isub(a, b) is equivalent to a -= b.

operator.itruediv(a, b)operator.\_\_itruediv\_\_(a, b)

a = itruediv(a, b) is equivalent to a /= b.

operator.ixor(a, b)

operator.\_\_ixor\_\_(a, b)

a = ixor(a, b) is equivalent to a ^= b.