

operator — Functional Interface to Built-in Operators

Purpose: Functional interface to built-in operators.

Programming using iterators occasionally requires creating small functions for simple expressions. Sometimes, these can be implemented as lambda functions, but for some operations new functions are not needed at all. The operator module defines functions that correspond to built-in operations for arithmetic, comparison, and other operations corresponding to standard object APIs.

Logical Operations

There are functions for determining the boolean equivalent for a value, negating it to create the opposite boolean value, and comparing objects to see if they are identical.

```
# operator_boolean.py

from operator import *

a = -1
b = 5

print('a =', a)
print('b =', b)
print()

print('not_(a)      : ', not_(a))
print('truth(a)     : ', truth(a))
print('is_(a, b)     : ', is_(a, b))
print('is_not(a, b): ', is_not(a, b))
```

not_() includes the trailing underscore because not is a Python keyword. truth() applies the same logic used when testing an expression in an if statement or converting an expression to a bool. is_() implements the same check used by the is keyword, and is_not() does the same test and returns the opposite answer.

```
$ python3 operator_boolean.py

a = -1
b = 5

not_(a)      : False
truth(a)     : True
is_(a, b)     : False
is_not(a, b): True
```

Comparison Operators

All of the rich comparison operators are supported.

```
# operator_comparisons.py

from operator import *

a = 1
b = 5.0

print('a =', a)
print('b =', b)
for func in (lt, le, eq, ne, ge, gt):
    print('{}(a, b): {}'.format(func.__name__, func(a, b)))
```

The functions are equivalent to the expression syntax using <, <=, ==, >=, and >.

```
$ python3 operator_comparisons.py
```

```
a = 1
b = 5.0
lt(a, b): True
le(a, b): True
eq(a, b): False
ne(a, b): True
ge(a, b): False
gt(a, b): False
```

Arithmetic Operators

The arithmetic operators for manipulating numerical values are also supported.

```
# operator_math.py
```

```
from operator import *
```

```
a = -1
b = 5.0
c = 2
d = 6
```

```
print('a =', a)
print('b =', b)
print('c =', c)
print('d =', d)
```

```
print('\nPositive/Negative:')
print('abs(a):', abs(a))
print('neg(a):', neg(a))
print('neg(b):', neg(b))
print('pos(a):', pos(a))
print('pos(b):', pos(b))
```

```
print('\nArithmetic:')
print('add(a, b)      :', add(a, b))
print('floordiv(a, b):', floordiv(a, b))
print('floordiv(d, c):', floordiv(d, c))
print('mod(a, b)       :', mod(a, b))
print('mul(a, b)        :', mul(a, b))
print('pow(c, d)        :', pow(c, d))
print('sub(b, a)        :', sub(b, a))
print('truediv(a, b)    :', truediv(a, b))
print('truediv(d, c)    :', truediv(d, c))
```

```
print('\nBitwise:')
print('and_(c, d)      :', and_(c, d))
print('invert(c)       :', invert(c))
print('lshift(c, d):', lshift(c, d))
print('or_(c, d)        :', or_(c, d))
print('rshift(d, c):', rshift(d, c))
print('xor(c, d)        :', xor(c, d))
```

There are two separate division operators: `floordiv()` (integer division as implemented in Python before version 3.0) and `truediv()` (floating point division).

```
$ python3 operator_math.py
```

```
a = -1
b = 5.0
c = 2
d = 6
```

```
Positive/Negative:
abs(a): 1
neg(a): 1
neg(b): -5.0
```

```
neg(a): -1
pos(a): -1
pos(b): 5.0

Arithmetic:
add(a, b)      : 4.0
floordiv(a, b) : -1.0
floordiv(d, c) : 3
mod(a, b)      : 4.0
mul(a, b)      : -5.0
pow(c, d)      : 64
sub(b, a)      : 6.0
truediv(a, b)  : -0.2
truediv(d, c)  : 3.0

Bitwise:
and_(c, d)     : 2
invert(c)      : -3
lshift(c, d)   : 128
or_(c, d)      : 6
rshift(d, c)   : 1
xor(c, d)      : 4
```

Sequence Operators

The operators for working with sequences can be divided into four groups: building up sequences, searching for items, accessing contents, and removing items from sequences.

```
# operator_sequences.py

from operator import *

a = [1, 2, 3]
b = ['a', 'b', 'c']

print('a =', a)
print('b =', b)

print('\nConstructive:')
print('  concat(a, b):', concat(a, b))

print('\nSearching:')
print('  contains(a, 1)  :', contains(a, 1))
print('  contains(b, "d"):', contains(b, "d"))
print('  countOf(a, 1)   :', countOf(a, 1))
print('  countOf(b, "d") :', countOf(b, "d"))
print('  indexOf(a, 5)   :', indexOf(a, 1))

print('\nAccess Items:')
print('  getitem(b, 1)           :',
      getitem(b, 1))
print('  getitem(b, slice(1, 3)) :',
      getitem(b, slice(1, 3)))
print('  setitem(b, 1, "d")      :', end=' ')
setitem(b, 1, "d")
print(b)
print('  setitem(a, slice(1, 3), [4, 5]):', end=' ')
setitem(a, slice(1, 3), [4, 5])
print(a)

print('\nDestructive:')
print('  delitem(b, 1)           :', end=' ')
delitem(b, 1)
print(b)
print('  delitem(a, slice(1, 3)):', end=' ')
delitem(a, slice(1, 3))
print(a)
```

Some of these operations, such as `setitem()` and `delitem()`, modify the sequence in place and do not return a value.

```
$ python3 operator_sequences.py
```

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
```

Constructive:

```
concat(a, b): [1, 2, 3, 'a', 'b', 'c']
```

Searching:

```
contains(a, 1) : True
contains(b, "d"): False
countOf(a, 1)  : 1
countOf(b, "d") : 0
indexOf(a, 5)   : 0
```

Access Items:

```
getitem(b, 1) : b
getitem(b, slice(1, 3)) : ['b', 'c']
setitem(b, 1, "d") : ['a', 'd', 'c']
setitem(a, slice(1, 3), [4, 5]): [1, 4, 5]
```

Destructive:

```
delitem(b, 1) : ['a', 'c']
delitem(a, slice(1, 3)): [1]
```

In-place Operators

In addition to the standard operators, many types of objects support “in-place” modification through special operators such as `+=`. There are equivalent functions for in-place modifications, too:

```
# operator_inplace.py
```

```
from operator import *
```

```
a = -1
b = 5.0
c = [1, 2, 3]
d = ['a', 'b', 'c']
print('a =', a)
print('b =', b)
print('c =', c)
print('d =', d)
print()
```

```
a = iadd(a, b)
print('a = iadd(a, b) =>', a)
print()
```

```
c = iconcat(c, d)
print('c = iconcat(c, d) =>', c)
```

These examples only demonstrate a few of the functions. Refer to the standard library documentation for complete details.

```
$ python3 operator_inplace.py
```

```
a = -1
b = 5.0
c = [1, 2, 3]
d = ['a', 'b', 'c']
```

```
a = iadd(a, b) => 4.0
```

```
c = iconcat(c, d) => [1, 2, 3, 'a', 'b', 'c']
```

Attribute and Item “Getters”

One of the most unusual features of the operator module is the concept of *getters*. These are callable objects constructed at runtime to retrieve attributes of objects or contents from sequences. Getters are especially useful when working with iterators or generator sequences, where they are intended to incur less overhead than a lambda or Python function.

generator sequences, where they are intended to incur less overhead than a lambda or Python function.

```
# operator_attrgetter.py

from operator import *

class MyObj:
    """example class for attrgetter"""

    def __init__(self, arg):
        super().__init__()
        self.arg = arg

    def __repr__(self):
        return 'MyObj({})'.format(self.arg)

l = [MyObj(i) for i in range(5)]
print('objects  :', l)

# Extract the 'arg' value from each object
g = attrgetter('arg')
vals = [g(i) for i in l]
print('arg values:', vals)

# Sort using arg
l.reverse()
print('reversed  :', l)
print('sorted    :', sorted(l, key=g))
```

Attribute getters work like `lambda x, n='attrname': getattr(x, n)`:

```
$ python3 operator_attrgetter.py

objects  : [MyObj(0), MyObj(1), MyObj(2), MyObj(3), MyObj(4)]
arg values: [0, 1, 2, 3, 4]
reversed  : [MyObj(4), MyObj(3), MyObj(2), MyObj(1), MyObj(0)]
sorted    : [MyObj(0), MyObj(1), MyObj(2), MyObj(3), MyObj(4)]
```

Item getters work like `lambda x, y=5: x[y]`:

```
# operator_itemgetter.py

from operator import *

l = [dict(val=-1 * i) for i in range(4)]
print('Dictionaries:')
print(' original:', l)
g = itemgetter('val')
vals = [g(i) for i in l]
print('   values:', vals)
print('   sorted:', sorted(l, key=g))

print()
l = [(i, i * -2) for i in range(4)]
print('\nTuples:')
print(' original:', l)
g = itemgetter(1)
vals = [g(i) for i in l]
print('   values:', vals)
print('   sorted:', sorted(l, key=g))
```

Item getters work with mappings as well as sequences.

```
$ python3 operator_itemgetter.py

Dictionaries:
original: [{'val': 0}, {'val': -1}, {'val': -2}, {'val': -3}]
values: [0, -1, -2, -3]
```

```
sorted: [{'val': -3}, {'val': -2}, {'val': -1}, {'val': 0}]
```

Tuples:

```
original: [(0, 0), (1, -2), (2, -4), (3, -6)]
values: [0, -2, -4, -6]
sorted: [(3, -6), (2, -4), (1, -2), (0, 0)]
```

Combining Operators and Custom Classes

The functions in the operator module work via the standard Python interfaces for their operations, so they work with user-defined classes as well as the built-in types.

```
# operator_classes.py

from operator import *

class MyObj:
    """Example for operator overloading"""

    def __init__(self, val):
        super(MyObj, self).__init__()
        self.val = val

    def __str__(self):
        return 'MyObj({})'.format(self.val)

    def __lt__(self, other):
        """compare for less-than"""
        print('Testing {} < {}'.format(self, other))
        return self.val < other.val

    def __add__(self, other):
        """add values"""
        print('Adding {} + {}'.format(self, other))
        return MyObj(self.val + other.val)

a = MyObj(1)
b = MyObj(2)

print('Comparison:')
print(lt(a, b))

print('\nArithmetic:')
print(add(a, b))
```

Refer to the Python reference guide for a complete list of the special methods used by each operator.

```
$ python3 operator_classes.py

Comparison:
Testing MyObj(1) < MyObj(2)
True

Arithmetic:
Adding MyObj(1) + MyObj(2)
MyObj(3)
```

See also

- [Standard library documentation for operator](#)
- [functools](#) – Functional programming tools, including the `total_ordering()` decorator for adding rich comparison methods to a class.
- [itertools](#) – Iterator operations.
- [collections](#) – Abstract types for collections.
- [numbers](#) – Abstract types for numerical values.

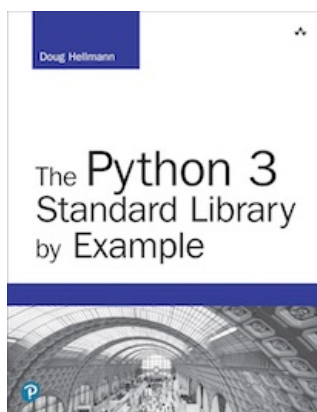
Quick Links

[Logical Operations](#)
[Comparison Operators](#)
[Arithmetic Operators](#)
[Sequence Operators](#)
[In-place Operators](#)
[Attribute and Item “Getters”](#)
[Combining Operators and Custom Classes](#)

This page was last updated 2017-07-30.

Navigation

[↗ itertools — Iterator Functions](#)
[↗ contextlib — Context Manager Utilities](#)





[Get the book](#)

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for [examples for Python 2?](#)

This Site

 [Module Index](#)
 [Index](#)



© Copyright 2019, Doug Hellmann



Other Writing

 [Blog](#)
 [The Python Standard Library By Example](#)