Overloading All Versions

Examples

Operator overloading

Below are the operators that can be overloaded in classes, along with the method definitions that are required, and an example of the operator in use within an expression.

N.B. The use of `other` as a variable name is not mandatory, but is considered the norm.

Operator	Method	Expression
+ Addition	`__add__(self, other)`	a1 + a2
- Subtraction	`__sub__(self, other)`	a1 - a2
* Multiplication	`__mul__(self, other)`	a1 * a2
@ Matrix Multiplication	`__matmul__(self, other)`	a1 @ a2 (*Python 3.5*)
/ Division	`__div__(self, other)`	a1 / a2 (*Python 2 only*)
/ Division	`__truediv__(self, other)`	a1 / a2 (*Python 3*)
// Floor Division	`__floordiv__(self, other)`	a1 // a2
% Modulo/Remainder	`__mod__(self, other)`	a1 % a2
** Power	`__pow__(self, other[, modulo])`	a1 ** a2
<< Bitwise Left Shift	`__lshift__(self, other)`	a1 << a2
>> Bitwise Right Shift	`__rshift__(self, other)`	a1 >> a2
& Bitwise AND	`__and__(self, other)`	a1 & a2
^ Bitwise XOR	`__xor__(self, other)`	a1 ^ a2
| (Bitwise OR)	`__or__(self, other)`	a1 | a2
- Negation (Arithmetic)	`__neg__(self)`	-a1
+ Positive	`__pos__(self)`	+a1
~ Bitwise NOT	`__invert__(self)`	~a1
< Less than	`__lt__(self, other)`	a1 < a2
<= Less than or Equal to	`__le__(self, other)`	a1 <= a2
== Equal to	`__eq__(self, other)`	a1 == a2
!= Not Equal to	`__ne__(self, other)`	a1 != a2
> Greater than	`__gt__(self, other)`	a1 > a2
>= Greater than or Equal to	`__ge__(self, other)`	a1 >= a2
[index] Index operator	`__getitem__(self, index)`	a1[index]
in In operator	`__contains__(self, other)`	a2 in a1
(*args, ...) Calling	`__call__(self, *args, **kwargs)`	a1(*args, **kwargs)

The optional parameter `modulo` for `__pow__` is only used by the `pow` built-in function.

Each of the methods corresponding to a *binary* operator has a corresponding "right" method which start with `__r` , for example `__radd__` :

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
        return other + self.a

A(1) + 2  # Out:  3
2 + A(1)  # prints radd. Out: 3
```

as well as a corresponding inplace version, starting with `__i`:

```
class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
        print("iadd")
        return self

b = B(2)
b.b       # Out: 2
b += 1    # prints iadd
b.b       # Out: 3
```

Since there's nothing special about these methods, many other parts of the language, parts of the standard library, and even third-party modules add magic methods on their own, like methods to cast an object to a type or checking properties of the object. For example, the builtin `str()` function calls the object's `__str__` method, if it exists. Some of these uses are listed below.

Function	Method	Expression
Casting to `int`	`__int__(self)`	`int(a1)`
Absolute function	`__abs__(self)`	`abs(a1)`
Casting to `str`	`__str__(self)`	`str(a1)`
Casting to `unicode`	`__unicode__(self)`	`unicode(a1)` (Python 2 only)
String representation	`__repr__(self)`	`repr(a1)`
Casting to `bool`	`__nonzero__(self)`	`bool(a1)`
String formatting	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hashing	`__hash__(self)`	`hash(a1)`
Length	`__len__(self)`	`len(a1)`
Reversed	`__reversed__(self)`	`reversed(a1)`
Floor	`__floor__(self)`	`math.floor(a1)`
Ceiling	`__ceil__(self)`	`math.ceil(a1)`

There are also the special methods `__enter__` and `__exit__` for context managers, and many more.

Magic/Dunder Methods

Magic (also called dunder as an abbreviation for double-underscore) methods in Python serve a similar purpose to operator overloading in other languages. They allow a class to define its behavior when it is used as an operand in unary or binary operator expressions. They also serve as implementations called by some built-in functions.

Consider this implementation of two-dimensional vectors.

```
import math

class Vector(object):
    # instantiation
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # unary negation (-v)
    def __neg__(self):
        return Vector(-self.x, -self.y)

    # addition (v + u)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # subtraction (v - u)
    def __sub__(self, other):
        return self + (-other)

    # equality (v == u)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # abs(v)
    def __abs__(self):
```

```
    def __abs__(self):
        return math.hypot(self.x, self.y)

    # str(v)
    def __str__(self):
        return '<{0.x}, {0.y}>'.format(self)

    # repr(v)
    def __repr__(self):
        return 'Vector({0.x}, {0.y})'.format(self)
```

Now it is possible to naturally use instances of the `Vector` class in various expressions.

```
v = Vector(1, 4)
u = Vector(2, 0)

u + v           # Vector(3, 4)
print(u + v)    # "<3, 4>" (implicit string conversion)
u - v           # Vector(1, -4)
u == v          # False
u + v == v + u  # True
abs(u + v)      # 5.0
```

Container and sequence types

It is possible to emulate container types, which support accessing values by key or index.

Consider this naive implementation of a sparse list, which stores only its non-zero elements to conserve memory.

```
class sparselist(object):
    def __init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    def __getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)
        try:
            return self.data[index]
        except KeyError:
            return 0.0

    # l[index] = value
    def __setitem__(self, index, value):
        self.data[index] = value

    # del l[index]
    def __delitem__(self, index):
        if index in self.data:
            del self.data[index]

    # value in l
    def __contains__(self, value):
        return value == 0.0 or value in self.data.values()

    # len(l)
    def __len__(self):
        return self.size

    # for value in l: ...
    def __iter__(self):
        return (self[i] for i in range(self.size)) # use xrange for python2
```

Then, we can use a `sparselist` much like a regular `list`.

```
l = sparselist(10 ** 6)  # list with 1 million elements
0 in l                   # True
10 in l                  # False

l[12345] = 10
10 in l                  # True
l[12345]                 # 10

for v in l:
    pass  # 0, 0, 0, ... 10, 0, 0 ... 0
```

Callable types

```
class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1)  # 3
add2(2)  # 4
```

Handling unimplemented behaviour

If your class doesn't implement a specific overloaded operator for the argument types provided, it should `return NotImplemented` (**note** that this is a special constant , not the same as `NotImplementedError`). This will allow Python to fall back to trying other methods to make the operation work:

> When `NotImplemented` is returned, the interpreter will then try the reflected operation on the other type, or some other fallback, depending on the operator. If all attempted operations return `NotImplemented` , the interpreter will raise an appropriate exception.

For example, given `x + y` , if `x.__add__(y)` returns unimplemented, `y.__radd__(x)` is attempted instead.

```
class NotAddable(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return NotImplemented


class Addable(NotAddable):

    def __add__(self, other):
        return Addable(self.value + other.value)

    __radd__ = __add__
```

As this is the *reflected* method we have to implement `__add__` **and** `__radd__` to get the expected behaviour in all cases; fortunately, as they are both doing the same thing in this simple example, we can take a shortcut.

In use:

```
>>> x = NotAddable(1)>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

Syntax

Parameters

Remarks