Comparisons

All Versions

Examples

Chain Comparisons

You can compare multiple items with multiple comparison operators with chain comparison. For example

```
x > y > z
```

is just a short form of:

```
x > y and y > z
```

This will evaluate to True only if both comparisons are True .

The general form is

```
a OP b OP c OP d ...
```

Where OP represents one of the multiple comparison operations you can use, and the letters represent arbitrary valid expressions.

Note that 0 != 1 != 0 evaluates to True , even though 0 != 0 is False . Unlike the common mathematical notation in which x != y != z means that x , y and z have different values. Chaining == operations has the natural meaning in most cases, since equality is generally transitive.

Style

There is no theoretical limit on how many items and comparison operations you use as long you have proper syntax:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

The above returns True if each comparison returns True . However, using convoluted chaining is not a good style. A good chaining will be "directional", not more complicated than

```
1 > x > -4 > y != 8
```

Side effects

As soon as one comparison returns False , the expression evaluates immediately to False , skipping all remaining comparisons.

Note that the expression exp in a > exp > b will be evaluated only once, whereas in the case of

```
a > exp and exp > b
```

exp will be computed twice if a > exp is true.

Comparison by `is` vs `==`

A common pitfall is confusing the equality comparison operators is and == .

a == b compares the value of a and b .

a is b will compare the *identities* of a and b .

To illustrate:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False

a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b     # True
a is b     # True
b = a[:]   # b now references a copy of a
a == b     # True
a is b     # False [!!]
```

Basically, is can be thought of as shorthand for id(a) == id(b) .

Beyond this, there are quirks of the run-time environment that further complicate things. Short strings and small integers will return True when compared with is , due to the Python machine attempting to use less memory for identical objects.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

But longer strings and larger integers will be stored separately.

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

You should use is to test for None :

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

A use of is is to test for a “sentinel” (i.e. a unique object).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn’t provided
        pass
    else:
        # value was provided
        pass
```

Comparing Objects

In order to compare the equality of custom classes, you can override == and != by defining __eq__ and __ne__ methods. You can also override __lt__ (<), __le__ (<=), __gt__ (>), and __ge__ (>). Note that you only need to override two comparison methods, and Python can handle the rest (== is the same as not < and not > , etc.)

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item

a = Foo(5)
b = Foo(5)
a == b     # True
a != b     # False
a is b     # False
```

Note that this simple comparison assumes that other (the object being compared to) is the same object type. Comparing to another type will throw an error:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
    def __ne__(self, other):
        return self.other_item != other.other_item

c = Bar(5)
a == c    # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

Checking isinstance() or similar will help prevent this (if desired).

Equal To

```
x == y
```

This expression evaluates if `x` and `y` are the same value and returns the result as a boolean value. Generally both type and value need to match, so the int `12` is not the same as the string `'12'` .

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
# False
```

Note that each type has to define a function that will be used to evaluate if two values are the same. For builtin types these functions behave as you'd expect, and just evaluate things based on being the same value. However custom types could define equality testing as whatever they'd like, including always returning `True` or always returning `False` .

Greater than or less than

```
x > y
x < y
```

These operators compare two types of values, they're the less than and greater than operators. For numbers this simply compares the numerical values to see which is larger:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

For strings they will compare lexicographically, which is similar to alphabetical order but not quite the same.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

In these comparisons, lowercase letters are considered 'greater than' uppercase, which is why `"gamma" < "OMEGA"` is false. If they were all uppercase it would return the expected alphabetical ordering result:

```
"GAMMA" < "OMEGA"
# True
```

Each type defines it's calculation with the `<` and `>` operators differently, so you should investigate what the operators mean with a given type before using it.

Not equal to

```
x != y
```

This returns `True` if `x` and `y` are not equal and otherwise returns `False` .

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Common Gotcha: Python does not enforce typing

In many other languages, if you run the following (Java example)

```
if("asgdsrf" == 0) {
    //do stuff
}
```

... you'll get an error. You can't just go comparing strings to integers like that. In Python, this is a perfectly legal statement - it'll just resolve to `False` .

A common gotcha is the following

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

This comparison will evaluate to `False` without an error, every time, potentially hiding a bug or breaking a conditional.

Syntax

```
!= - Is not equal to
== - Is equal to
> - greater than
< - less than
>= - greater than or equal to
<= - less than or equal to
is - test if objects are the exact same object
is not = test if objects are not the exact same object
```

Parameters

Parameter	Details
x	First item to be compared
y	Second item to be compared

Remarks