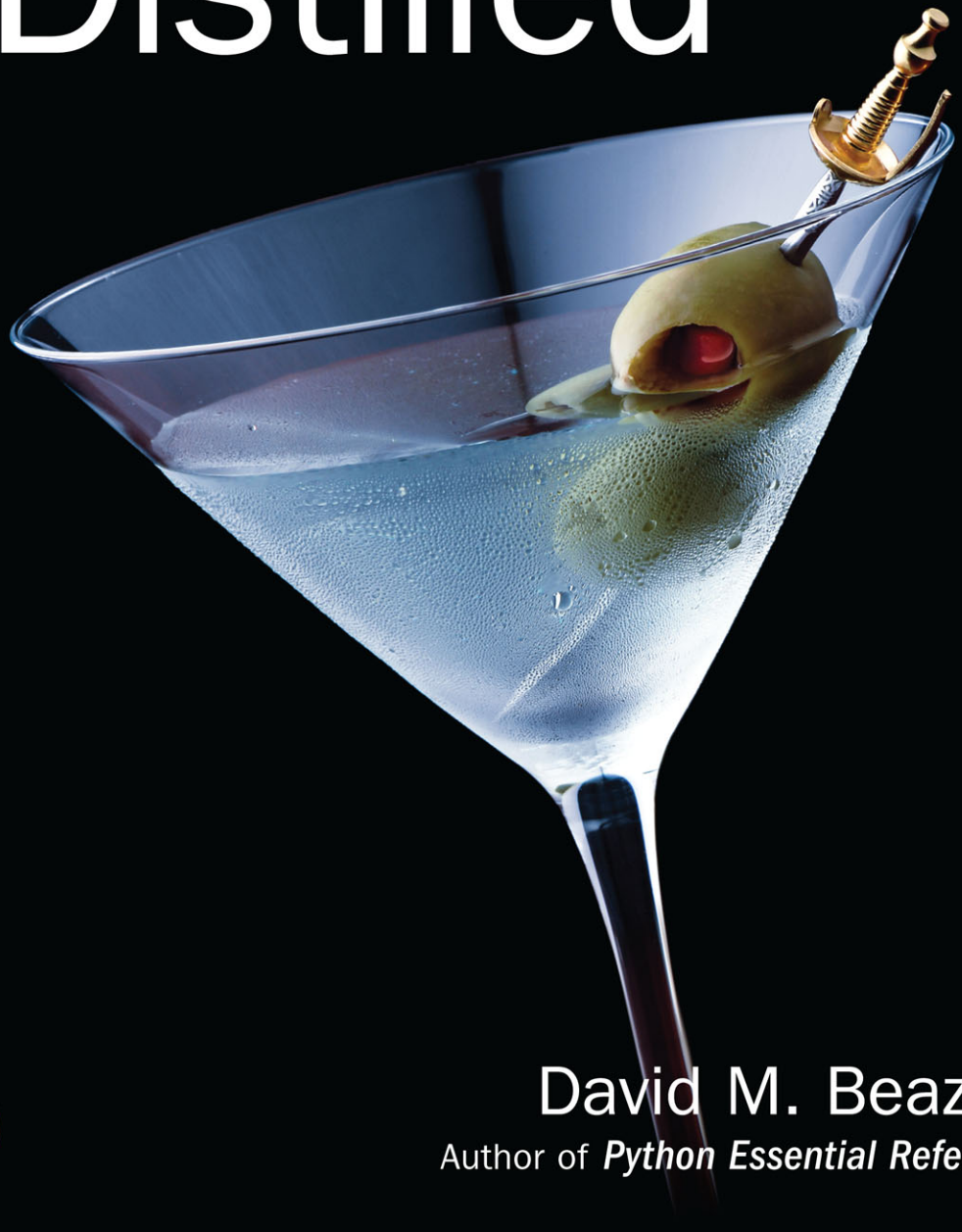
Python Distilled



David M. Beazley
Author of *Python Essential Reference*

Python Distilled

```
>>> b.__radd__(a)
45.7
>>>
```

This example might seem surprising but it reflects the fact that integers don't actually know anything about floating-point numbers. However, floating-point numbers do know about integers—as integers are, mathematically, a special kind of floating-point numbers. Thus, the reversed operand produces the correct answer.

The methods `__iadd__()`, `__isub__()`, and so forth are used to support in-place arithmetic operators such as `a += b` and `a -= b` (also known as augmented assignment). A distinction is made between these operators and the standard arithmetic methods because the implementation of the in-place operators might be able to provide certain customizations or performance optimizations. For instance, if the object is not shared, the value of an object could be modified in place without allocating a newly created object for the result. If the in-place operators are left undefined, an operation such as `a += b` is evaluated using `a = a + b` instead.

There are no methods that can be used to define the behavior of the logical `and`, `or`, or `not` operators. The `and` and `or` operators implement short-circuit evaluation where evaluation stops if the final result can already be determined. For example:

```
>>> True or 1/0    # Does not evaluate 1/0
True
>>>
```

This behavior involving unevaluated subexpressions can't be expressed using the evaluation rules of a normal function or method. Thus, there is no protocol or set of methods for redefining it. Instead, it is handled as a special case deep inside the implementation of Python itself.

4.11 Comparison Protocol

Objects can be compared in various ways. The most basic check is an identity check with the `is` operator. For example, `a is b`. Identity does not consider the values stored inside of an object, even if they happen to be the same. For example:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> c = [1, 2, 3]
>>> a is c
False
>>>
```

The `is` operator is an internal part of Python that can't be redefined. All other comparisons on objects are implemented by the methods in Table 4.3.

Table 4.3 Methods for Instance Comparison and Hashing

Method	Description
`__bool__(self)`	Returns `False` or `True` for truth-value testing
`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__le__(self, other)`	`self <= other`
`__gt__(self, other)`	`self > other`
`__ge__(self, other)`	`self >= other`
`__hash__(self)`	Computes an integer hash index

The `__bool__()` method, if present, is used to determine the truth value when an object is tested as part of a condition or conditional expression. For example:

```
if a:            # Executes a.__bool__()
   ...
else:
   ...
```

If `__bool__()` is undefined, then `__len__()` is used as a fallback. If both `__bool__()` and `__len__()` are undefined, an object is simply considered to be `True`.

The `__eq__()` method is used to determine basic equality for use with the `==` and `!=` operators. The default implementation of `__eq__()` compares objects by identity using the `is` operator. The `__ne__()` method, if present, can be used to implement special processing for `!=`, but is usually not required as long as `__eq__()` is defined.

Ordering is determined by the relational operators (`<`, `>`, `<=`, and `>=`) using methods such as `__lt__()` and `__gt__()`. As with other mathematical operations, the evaluation rules are subtle. To evaluate `a < b`, the interpreter will first try to execute `a.__lt__(b)` except where `b` is a subtype of `a`. In that one specific case, `b.__gt__(a)` executes instead. If this initial method is not defined or returns `NotImplemented`, the interpreter tries a reversed comparison, calling `b.__gt__(a)`. Similar rules apply to operators such as `<=` and `>=`. For example, evaluating `<=` first tries to evaluate `a.__le__(b)`. If not implemented, `b.__ge__(a)` is tried.

Each of the comparison methods takes two arguments and is allowed to return any kind of value, including a Boolean value, a list, or any other Python type. For instance, a numerical package might use this to perform an element-wise comparison of two matrices, returning a matrix with the results. If comparison is not possible, the methods should return the built-in object `NotImplemented`. This is not the same as the `NotImplementedError` exception. For example:

```
>>> a = 42      # int
>>> b = 52.3    # float
>>> a.__lt__(b)
NotImplemented
```

```
>>> b.__gt__(a)
True
>>>
```

It is not necessary for an ordered object to implement all of the comparison operations in Table 4.3. If you want to be able to sort objects or use functions such as `min()` or `max()`, then `__lt__()` must be minimally defined. If you are adding comparison operators to a user-defined class, the `@total_ordering` class decorator in the `functools` module may be of some use. It can generate all of the methods as long as you minimally implement `__eq__()` and one of the other comparisons.

The `__hash__()` method is defined on instances that are to be placed into a set or be used as keys in a mapping (dictionary). The value returned is an integer that should be the same for two instances that compare as equal. Moreover, `__eq__()` should always be defined together with `__hash__()` because the two methods work together. The value returned by `__hash__()` is typically used as an internal implementation detail of various data structures. However, it's possible for two different objects to have the same hash value. Therefore, `__eq__()` is necessary to resolve potential collisions.

4.12 Conversion Protocols

Sometimes, you must convert an object to a built-in type such as a string or a number. The methods in Table 4.4 can be defined for this purpose.

Table 4.4 Methods for Conversions

Method	Description
`__str__(self)`	Conversion to a string
`__bytes__(self)`	Conversion to bytes
`__format__(self, format_spec)`	Creates a formatted representation
`__bool__(self)`	`bool(self)`
`__int__(self)`	`int(self)`
`__float__(self)`	`float(self)`
`__complex__(self)`	`complex(self)`
`__index__(self)`	Conversion to a integer index `[self]`

The `__str__()` method is called by the built-in `str()` function and by functions related to printing. The `__format__()` method is called by the `format()` function or the `format()` method of strings. The `format_spec` argument is a string containing the format specification. This string is the same as the `format_spec` argument to `format()`. For example:

```
f'{x:spec}'                # Calls x.__format__('spec')
format(x, 'spec')          # Calls x.__format__('spec')
'x is {0:spec}'.format(x)  # Calls x.__format__('spec')
```

The syntax of the format specification is arbitrary and can be customized on an object-by-object basis. However, there is a standard set of conventions used for the built-in types. More information about string formatting, including the general format of the specifier, can be found in Chapter 9.

The `__bytes__()` method is used to create a byte representation if an instance is passed to `bytes()`. Not all types support byte conversion.

The numeric conversions `__bool__()`, `__int__()`, `__float__()`, and `__complex__()` are expected to produce a value of the corresponding built-in type.

Python never performs implicit type conversions using these methods. Thus, even if an object `x` implements an `__int__()` method, the expression `3 + x` will still produce a `TypeError`. The only way to execute `__int__()` is through an explicit use of the `int()` function.

The `__index__()` method performs an integer conversion of an object when it's used in an operation that requires an integer value. This includes indexing in sequence operations. For example, if `items` is a list, performing an operation such as `items[x]` will attempt to execute `items[x.__index__()]` if `x` is not an integer. `__index__()` is also used in various base conversions such as `oct(x)` and `hex(x)`.

4.13 Container Protocol

The methods in Table 4.5 are used by objects that want to implement containers of various kinds—lists, dicts, sets, and so on.

Table 4.5 Methods for Containers

Method	Description
`__len__(self)`	Returns the length of `self`
`__getitem__(self, key)`	Returns `self[key]`
`__setitem__(self, key, value)`	Sets `self[key] = value`
`__delitem__(self, key)`	Deletes `self[key]`
`__contains__(self, obj)`	`obj in self`

Here's an example:

```
a = [1, 2, 3, 4, 5, 6]
len(a)                  # a.__len__()
x = a[2]                # x = a.__getitem__(2)
a[1] = 7                # a.__setitem__(1,7)
del a[2]                # a.__delitem__(2)
5 in a                  # a.__contains__(5)
```

The `__len__()` method is called by the built-in `len()` function to return a non-negative length. This function also determines truth values unless the `__bool__()` method has also been defined.

For accessing individual items, the `__getitem__()` method can return an item by key value. The key can be any Python object, but it is expected to be an integer for ordered sequences such as lists and arrays. The `__setitem__()` method assigns a value to an element. The `__delitem__()` method is invoked whenever the `del` operation is applied to a single element. The `__contains__()` method is used to implement the `in` operator.

Slicing operations such as `x = s[i:j]` are also implemented using `__getitem__()`, `__setitem__()`, and `__delitem__()`. For slices, a special `slice` instance is passed as the key. This instance has attributes that describe the range of the slice being requested. For example:

```
a = [1,2,3,4,5,6]
x = a[1:5]           # x = a.__getitem__(slice(1, 5, None))
a[1:3] = [10,11,12]  # a.__setitem__(slice(1, 3, None), [10, 11, 12])
del a[1:4]           # a.__delitem__(slice(1, 4, None))
```

The slicing features of Python are more powerful than many programmers realize. For example, the following variations of extended slicing are all supported and may be useful for working with multidimensional data structures such as matrices and arrays:

```
a = m[0:100:10]           # Strided slice (stride=10)
b = m[1:10, 3:20]         # Multidimensional slice
c = m[0:100:10, 50:75:5]  # Multiple dimensions with strides
m[0:5, 5:10] = n          # extended slice assignment
del m[:10, 15:]           # extended slice deletion
```

The general format for each dimension of an extended slice is `i:j[:stride]`, where `stride` is optional. As with ordinary slices, you can omit the starting or ending values for each part of a slice.

In addition, the `Ellipsis` (written as `...`) is available to denote any number of trailing or leading dimensions in an extended slice:

```
a = m[..., 10:20]    # extended slice access with Ellipsis
m[10:20, ...] = n
```

When using extended slices, the `__getitem__()`, `__setitem__()`, and `__delitem__()` methods implement access, modification, and deletion, respectively. However, instead of an integer, the value passed to these methods is a tuple containing a combination of slice or `Ellipsis` objects. For example,

```
a = m[0:10, 0:100:5, ...]
```

invokes `__getitem__()` as follows:

```
a = m.__getitem__((slice(0,10,None), slice(0,100,5), Ellipsis))
```

Python strings, tuples, and lists currently provide some support for extended slices. No part of Python or its standard library make use of multidimensional slicing or the `Ellipsis`. Those features are reserved purely for third-party libraries and frameworks. Perhaps the most common place you would see them used is in a library such as `numpy`.

4.14 Iteration Protocol

If an instance, `obj`, supports iteration, it provides a method, `obj.__iter__()`, that returns an iterator. An iterator `iter`, in turn, implements a single method, `iter.__next__()`, that returns the next object or raises `StopIteration` to signal the end of iteration. These methods are used by the implementation of the `for` statement as well as other operations that implicitly perform iteration. For example, the statement `for x in s` is carried out by performing these steps:

```
_iter = s.__iter__()
while True:
    try:
         x = _iter.__next__()
    except StopIteration:
         break
    # Do statements in body of for loop
    ...
```

An object may optionally provide a reversed iterator if it implements the `__reversed__()` special method. This method should return an iterator object with the same interface as a normal iterator (that is, a `__next__()` method that raises `StopIteration` at the end of iteration). This method is used by the built-in `reversed()` function. For example:

```
>>> for x in reversed([1,2,3]):
...     print(x)
3
2
1
>>>
```

A common implementation technique for iteration is to use a generator function involving `yield`. For example:

```
class FRange:
    def __init__(self, start, stop, step):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        x = self.start
        while x < self.stop:
            yield x
            x += self.step

# Example use:
```