Chapter 30. Operator Overloading

This chapter continues our in-depth survey of class mechanics by focusing on operator overloading. We looked briefly at operator overloading in prior chapters. Here, we'll fill in more details and explore a handful of commonly used overloading methods, most of which we haven't yet encountered. Although we don't have space to demonstrate each of the many operator-overloading methods available, those we will code here are a representative sample large enough to uncover the possibilities of this Python class feature.

The Basics

Really "operator overloading" simply means *intercepting* built-in operations in a class's methods—Python automatically invokes your methods when instances of the class appear in built-in operations, and your method's return value becomes the result of the corresponding operation. Here's a review of the key ideas behind overloading:

- Operator overloading lets classes intercept normal Python operations.
- Classes can overload all Python built-in expression operators.
- Classes can also overload other built-in operations such as printing, function calls, and attribute access.
- Overloading is implemented by providing specially named methods in a class.
- Python predefines the special method names that correspond to built-in operations.

In other words, when methods of predefined special names are provided in a class, Python automatically calls them when instances of the class appear in their associated built-in operations or expressions. Your class provides the behavior of the corresponding operation for instance objects created from it.

As you've learned, operator-overloading methods are never required and generally don't have defaults (apart from a handful that all classes get from the implied `object` root class). If you don't code or inherit an overloading

method, it just means that your class does not support the corresponding operation. When used, though, these methods allow classes to emulate the interfaces of built-in objects, which makes them consistent, and compatible with more code.

Constructors and Expressions: __init__ and __sub__

As a warm-up, consider the simple class in Example 30-1: its `Number` class, coded in module file *number.py*, provides a method to intercept instance construction (`__init__`), as well as one for catching subtraction expressions (`__sub__`). Special methods like these are the hooks that let you tie into built-in operations.

Example 30-1. number.py

```
class Number:
    def __init__(self, start):                 # On Nu
        self.data = start
    def __sub__(self, other):                  # On ir
        return Number(self.data - other)       # Resul
```

As we've already learned, the `__init__` constructor method seen in this code is the most commonly used operator-overloading method in Python; it's present in most classes and used to initialize the newly created instance object using any arguments passed to the class name. The `__sub__` method plays the binary-operator role that `__add__` did in Chapter 27's introduction, intercepting subtraction expressions and returning a new instance of the class as its result (and running `__init__` along the way):

```
>>> from number import Number                  # Fetch
>>> X = Number(5)                              # Numbe
>>> Y = X - 2                                  # Numbe
>>> Y.data                                     # Y is
3
```

We've already studied `__init__` and basic binary operators like `__sub__` in some depth, so we won't rehash their usage further here. In this chapter, we will tour some other tools available in this domain and look at example code that applies them in common use cases.

NOTE

Construction convolution: Technically, instance creation first triggers the `__new__` method, which creates and returns the new instance object, which is then passed into `__init__` for initialization. Since `__new__` has a built-in implementation and is redefined in only very limited roles, though, nearly all Python classes initialize by defining an `__init__` method. We'll explore one use case for `__new__` when we study *metaclasses* in Chapter 40; though rare, it is sometimes also used to customize creation of instances of immutable types.

Common Operator-Overloading Methods

Just about everything you can do to built-in objects such as integers and lists has a corresponding specially named method for overloading in classes. Table 30-1 lists a few of the most common; there are many more. In fact, many overloading methods come in multiple versions (e.g., `__add__`, `__radd__`, and `__iadd__` for addition), which is one reason there are so many. See the Python language reference manual for an exhaustive list of the special method names available.

Table 30-1. Common operator-overloading methods

Method	Implements	Called for
`__init__`	Constructor	Object creation: `X = Class(args)`
`__del__`	Destructor	Object reclamation of `X`
`__add__`	Operator `+` (among others)	`X + Y`, `X += Y` if no `__iadd__`
`__or__`	Operator `|` (bitwise OR)	`X | Y`, `X |= Y` if no `__ior__`
`__repr__`, `__str__`	Printing, conversions	`print(X)`, `X`, `repr(X)`, `str(X)`, `f'{X!r}'`
`__call__`	Function calls	`X(*pargs, **kargs)`
`__getattr__`	Attribute fetch	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__delattr__`	Attribute deletion	`del X.any`
`__getattribute__`	Attribute fetch	`X.any`
`__getitem__`	Indexing, slicing, iteration	`X[i]`, `X[i:j]`, `for` and other iterations if no `__iter__`
`__setitem__`	Index and slice assignment	`X[i] = value`, `X[i:j] = iterable`
`__delitem__`	Index and slice deletion	`del X[i]`, `del X[i:j]`
`__len__`	Length	`len(X)`, truth tests if no `__bool__`
`__bool__`	Boolean tests	`bool(X)`, truth tests
`__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`	Comparisons	`X < Y`, `X > Y`, `X <= Y`, `X >= Y`, `X == Y`, `X != Y`
`__radd__`	Right-side operators	`other + X`

Method	Implements	Called for
`__iadd__`	In-place augmented operators	`X += Y` (or else `__add__`)
`__iter__`, `__next__`	Iteration tools	`I=iter(X)`, `next(I)`, `for` and other iterations, `in` if no `__contains__`
`__contains__`	Membership test	`item in X`
`__index__`	Integer value	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]`
`__enter__`, `__exit__`	Context manager (Chapter 34)	`with obj as var:`
`__get__`, `__set__`, `__delete__`	Descriptor attributes (Chapter 38)	`X.attr`, `X.attr = value`, `del X.attr`
`__new__`	Creation (Chapter 40)	Object creation, before `__init__`

All overloading methods have names that start and end with two *underscores* to keep them distinct from other names you define in your classes. The mappings from special method names to expressions or operations are *predefined* by the Python language and documented in full in its standard language manual. For example, the name `__add__` always maps to `+` expressions by Python language definition, regardless of what an `__add__` method's code actually does; it's largely just a dispatch mechanism.

Operator-overloading methods may be *inherited* from superclasses if not defined, just like any other methods. Operator-overloading methods are also all *optional*—if you don't code or inherit one, that operation is simply unsupported by your class, and attempting it will raise an exception. Some built-in operations, like printing, have defaults inherited from the implied `object` root class, but most built-ins fail for class instances if no corresponding operator-overloading method is present.

As we go along here, keep in mind that most overloading methods are used only in advanced programs that require objects to behave like built-ins, though

the `__init__` constructor we've already met tends to appear in most classes. With that qualifier, let's explore some of the additional methods in Table 30-1 by example.

NOTE

Measure twice, post once: Although expressions trigger operator methods, be careful not to assume that there is a speed advantage to cutting out the middleperson and calling the operator method directly. In fact, calling the operator method directly might *be twice as slow*, presumably because of the overhead of a function call, which Python avoids or optimizes in built-in cases.

Here's the story for `len` and `__len__` using Chapter 21's timing techniques on Python 3.12 and macOS. Calling `__len__` directly takes twice as long (and has since Python 2.X):

```
$ python3 -m timeit -n 10000 -r 10 \
             -s "L = list(range(100))" "x = L.__len__()"
10000 loops, best of 10: 53.4 nsec per loop

$ python3 -m timeit -n 10000 -r 10 \
             -s "L = list(range(100))" "x = len(L)"
10000 loops, best of 10: 25.3 nsec per loop
```

This is not as contrived as it may seem—recommendations for using the slower alternative in the name of speed have been known to crop up in venues that shall remain nameless here.

Indexing and Slicing: __getitem__ and __setitem__

Our first new method set allows your classes to mimic some of the behaviors of sequences and mappings. If defined in a class (or inherited by it), the `__getitem__` method is called automatically for instance-indexing operations. When an instance `X` appears in an indexing expression like `X[i]`, Python calls the `__getitem__` method inherited by the instance, passing `X` to the first argument and the index `i` in brackets to the second argument.

For example, the following class returns the square of an index value—atypical perhaps but illustrative of the mechanism in general:

```
>>> class Indexer:
        def __getitem__(self, index):
            return index ** 2

>>> X = Indexer()
>>> X[2]                                # X[i] calls X.
4

>>> for i in range(5):
        print(X[i], end=' ')             # Runs __getite

0 1 4 9 16
```

It's up to your class to define what this expression means, though it should generally imitate a sequence index or mapping key fetch; returning the index squared as done here works but probably won't qualify as "best practice."

Intercepting Slices

Surprisingly, in addition to indexing, `__getitem__` is also called for *slice expressions*. Formally speaking, built-in object types handle slicing the same way. For example, the following demos slicing at work on a built-in list, using upper and lower bounds, omitted parts, and a stride (see Chapter 7 if you need a refresher on slicing):

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                              # Slice with sl
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[:-1]
[5, 6, 7, 8]
>>> L[::2]
[5, 7, 9]
```

Really, though, slicing bounds are bundled up into a *slice object* and passed to the list's implementation of indexing. In fact, you can always pass a slice

object manually—slice syntax is mostly syntactic sugar for indexing with a slice object:

```
>>> L[slice(2, 4)]                      # Slice with sl
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

This matters in classes with a `__getitem__` method—this method will be called *both* for basic indexing (with an index or key) and for slicing (with a slice object). Our previous class won't handle slicing because its math assumes integer indexes are passed, but the following class will. When called for indexing, the argument is an integer as before:

```
>>> class Indexer:
        def __init__(self, data):
            self.data = data
        def __getitem__(self, index):    # Called for i
            print('getitem:', index)
            return self.data[index]      # Perform inde

>>> X = Indexer([5, 6, 7, 8, 9])
>>> X[0]                                 # Indexing ser
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

When called for *slicing*, though, the method receives a slice object, which is simply passed along to the embedded list indexer in a new index expression:

```
>>> X[2:4]                              # Slicing sends
getitem: slice(2, 4, None)
```

```
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

Where needed, `__getitem__` can test the type of its argument, and extract slice object bounds—slice objects have attributes `start`, `stop`, and `step`, any of which can be `None` if omitted:

```
>>> class Indexer:
        def __getitem__(self, index):
            if isinstance(index, int):                #
                print('indexing', index)
            else:
                print('slicing', index.start, index.stc

>>> X = Indexer()
>>> X[99]
indexing 99
>>> X[1:99:2]
slicing 1 99 2
>>> X[1:]
slicing 1 None None
```

Run a `help(slice)` in a REPL for more info on this very special-case built-in object type, and see the section "Membership: __contains__, __iter__, and __getitem__" for another example of slice interception at work.

Intercepting Item Assignments

If used, the `__setitem__` index assignment method similarly intercepts both index and slice assignments—it receives a slice object for the latter, which may be passed along in another index assignment or used directly in the same way:

```
>>> class IndexSetter:
        def __init__(self, data):
            self.data = data
        def __setitem__(self, index, value):    # Catch
            print('setitem:', index)
            self.data[index] = value            # Assig

>>> X = IndexSetter([5, 6, 7, 8, 9])
>>> X[0] = 555
setitem: 0
>>> X[-2:] = [888, 999, 111]
setitem: slice(-2, None, None)

>>> X.data
[555, 6, 7, 888, 999, 111]
```

In fact, `__getitem__` may be called automatically in even more contexts than indexing and slicing—it's also an *iteration* fallback option, as you'll see in a moment. First, though, let's clear up a potential point of confusion in this category.

But __index__ Means As-Integer

Don't mistake the (perhaps unfortunately named) `__index__` method for `__getitem__` index interception. The `__index__` method returns an *integer value* for an instance when one is needed—and in retrospect, might have been better named `__asindex__`. For example, it's used by built-ins that convert to digit strings:

```
>>> class C:
        def __index__(self):
            return 255

>>> X = C()
>>> hex(X)              # Integer value
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

Although this method does not intercept instance indexing like `__getitem__`, it is also used in contexts that require an integer—including indexing and slicing:

```
>>> eds = [f'LP{i}e' for i in range(256)]
>>> eds[255]
'LP255e'
>>> X = C()
>>> eds[X]                # As index (not X[i]!)
'LP255e'
>>> eds[X:]               # As index (not X[i:]!)
['LP255e']
```

Though arguably misnamed, there is a rich history of former methods that `__index__` subsumes, which we'll mercifully omit here. The `__getitem__` method also subsumes former tools but retains a fallback role up next.

Index Iteration: __getitem__

Our next hook isn't always obvious to beginners but turns out to be surprisingly useful. In the absence of the more specific iteration methods we'll get to in the next section, the `for` statement works by repeatedly indexing an object from zero to higher indexes, until an out-of-bounds `IndexError` exception is detected. Because of that, `__getitem__` also turns out to be one way to overload *iteration* in Python—if only this method is defined, `for` loops call the class's `__getitem__` each time through, with successively higher offsets.

It's a case of "code one, get one free"—any built-in or user-defined object that responds to indexing also responds to `for` loop iteration:

```
>>> class StepperIndex:
        def __getitem__(self, i):
            return self.data[i]

>>> X = StepperIndex()                # X is a Stepperl
>>> X.data = 'hack'
>>>
>>> X[1]                              # Indexing calls
```

```
'a'
>>> for item in X:                      # for loops call
        print(item, end=' ')            # for indexes ite

h a c k
```

In fact, it's really a case of "code one, get a bunch free." Any class that supports `for` loops automatically supports all *iteration tools* in Python, many of which we've explored in earlier chapters (e.g., iteration tools were presented in Chapter 14). For instance, the `in` membership test, list comprehensions, the `map` built-in, list and tuple assignments, and some type constructors will also call `__getitem__` automatically to iterate if it's defined:

```
>>> 'k' in X                            # All call __geti
True

>>> [c for c in X]                      # Comprehension
['h', 'a', 'c', 'k']

>>> list(map(str.upper, X))             # map calls
['H', 'A', 'C', 'K']

>>> (a, b, c, d) = X                    # Sequence assigr
>>> a, d
('h', 'k')

>>> list(X), tuple(X), ''.join(X)       # And so on...
(['h', 'a', 'c', 'k'], ('h', 'a', 'c', 'k'), 'hack')

>>> X
<__main__.StepperIndex object at 0x10c4bcc20>
```

In practice, this technique can be used to create objects that provide a sequence interface and to add logic to built-in sequence type operations; we'll revisit this idea when extending built-in types in Chapter 32.

Iterable Objects: __iter__ and __next__

Although the `__getitem__` technique of the prior section works, it's really just a legacy fallback for iteration. Today, all iteration tools in Python will try

the `__iter__` method first before trying `__getitem__`. That is, they prefer the *iteration protocol* we learned about in Chapter 14 over repeatedly indexing an object; only if the object does not support the iteration protocol is indexing attempted instead. Generally speaking, you should prefer `__iter__` too—it supports general iteration tools better than `__getitem__` can.

For a review of this model's essentials, see Figure 14-1 in Chapter 14. In brief, iteration tools work by running an *iterable* object's `__iter__` method to fetch an *iterator* object. If this works as planned, Python then repeatedly calls this iterator object's `__next__` method to produce items until it raises a `StopIteration` exception. Built-in functions `iter` and `next` are also available as a conveniences for manual iterations— `iter(X)` is the same as `X.__iter__()` and `next(I)` is the same as `I.__next__()`, and Python internals may vary.

This iterable-object interface is given priority and attempted first. Only if no such `__iter__` method is found, Python falls back on the `__getitem__` scheme and repeatedly indexes by offsets as before until an `IndexError` exception is raised.

User-Defined Iterables

In the `__iter__` scheme, classes implement user-defined iterables by simply implementing the iteration protocol introduced in Chapter 14 and elaborated in Chapter 20. For example, the code in Example 30-2 uses a class to define a user-defined iterable that generates squares on demand, instead of all at once.

Example 30-2. squares.py

```
class Squares:
    def __init__(self, start, stop):    # Save state wh
        self.value = start - 1
        self.stop  = stop

    def __iter__(self):                  # Return iterat
        return self                      # Also called b

    def __next__(self):                  # Return a squa
        if self.value == self.stop:      # Also called b
```

```
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

When imported, its instances can appear in iteration tools just like built-ins:

```
$ python3
>>> from squares import Squares
>>> for i in Squares(1, 5):                # for calls __i
        print(i, end=' ')                  # Each iteratic

1 4 9 16 25
```

Here, the iterator object returned by `__iter__` is simply the instance `self` because the `__next__` method is part of this class itself. In more complex scenarios, the iterator object may be defined as a separate class and object with its own state information to support multiple active iterations over the same instance data (we'll code an example of this in a moment).

The end of the iteration is signaled with a Python `raise` statement—introduced in Chapter 29 and covered in full in the next part of this book, but which simply raises an exception as if Python itself had done so. Because of all this, manual iterations work the same on user-defined iterables as they do on built-in objects:

```
>>> X = Squares(1, 5)                   # Iterate manuc
>>> I = iter(X)                         # iter calls __
>>> next(I)                             # next calls __
1
>>> next(I)
4
…more omitted…
>>> next(I)
25
>>> next(I)                             # Can catch thi
StopIteration
```

An equivalent coding of this iterable with `__getitem__` might be less natural because the `for` would then iterate through all offsets zero and higher; the offsets passed in would be only indirectly related to the range of

values produced (`0…N` would need to map to `start…stop`). Because `__iter__` objects retain explicitly managed state between `next` calls, they can be more general than `__getitem__` .

On the other hand, iterables based on `__iter__` can sometimes be more complex and less functional than those based on `__getitem__` . They are really designed for iteration, not random indexing—in fact, they don't overload the indexing expression at all, though you can collect their items in a sequence such as a list to enable other operations:

```
>>> X = Squares(1, 5)
>>> X[1]
TypeError: 'Squares' object is not subscriptable
>>> list(X)[1]
4
```

Single versus multiple scans

The `__iter__` scheme is also the implementation for all the other iteration tools we saw in action for the `__getitem__` method—membership tests, type constructors, sequence assignment, and so on. Unlike our prior `__getitem__` example, though, we also need to be aware that a class's `__iter__` may be designed for a *single traversal* only, not many. Classes can choose either behavior explicitly in their code.

For example, because the current `Squares` class's `__iter__` always returns `self` with just one copy of iteration state, it is a *single-scan* iteration; once you've iterated over an instance of that class, it's empty. Calling `__iter__` again on the same instance returns `self` again—in whatever state it may have been left. As coded, you generally need to make a new iterable instance object for each new iteration:

```
>>> X = Squares(1, 5)                    # Make an iterc
>>> [n for n in X]                       # Exhausts item
[1, 4, 9, 16, 25]
>>> [n for n in X]                       # Now it's empt
[]

>>> [n for n in Squares(1, 5)]           # Make a new it
[1, 4, 9, 16, 25]
```

```
>>> list(Squares(1, 3))                 # A new object
[1, 4, 9]
```

To support multiple iterations more directly, we could also recode this example with an extra class or other technique, as we will in a moment. As is, though, by creating a *new instance* for each iteration, you get a fresh copy of iteration state:

```
>>> 36 in Squares(1, 10)                # Other iterati
True
>>> a, b, c = Squares(1, 3)             # Each calls __
>>> a, b, c
(1, 4, 9)
>>> ':'.join(map(str, Squares(1, 5)))
'1:4:9:16:25'
```

Just like single-scan built-ins such as `map`, converting to a *list* supports multiple scans as well but adds time and space performance costs, which may or may not be significant to a given program:

```
>>> X = Squares(1, 5)
>>> tuple(X), tuple(X)                  # Iterator exhc
((1, 4, 9, 16, 25), ())

>>> X = list(Squares(1, 5))
>>> tuple(X), tuple(X)
((1, 4, 9, 16, 25), (1, 4, 9, 16, 25))
```

We'll improve this to support multiple scans more directly ahead after a bit of compare and contrast.

Classes versus generators

Notice that the `Squares` iterable of Example 30-2 that we've been using so far would probably be simpler if it was coded with *generator functions or expressions*—tools introduced in Chapter 20 that automatically produce iterable objects and retain local variable state between iterations:

```
>>> def gsquares(start, stop):                    # Gene
        for i in range(start, stop + 1):
            yield i ** 2

>>> for i in gsquares(1, 5):
        print(i, end=' ')

1 4 9 16 25

>>> for i in (x ** 2 for x in range(1, 6)):       # Gene
        print(i, end=' ')

1 4 9 16 25
```

Unlike classes, generator functions and expressions implicitly save their state and create the methods required to conform to the *iteration protocol*—with obvious advantages in code conciseness for simpler examples like these. That is, generators' automatic and implicit `__iter__` and `__next__` suffice here.

On the other hand, the class's more explicit attributes and methods, extra structure, inheritance hierarchies, and support for multiple behaviors may be better suited for richer use cases.

Of course, for this artificial example, you could in fact skip both techniques and simply use a `for` loop, `map`, or a list comprehension to build the list all at once. Barring performance data to the contrary, the best and fastest way to accomplish a task in Python is often also the simplest:

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

That said, classes are better at modeling more complex iterations, especially when they can benefit from the assets of classes in general. An iterable that produces items in a complex database or web service result, for example, might be able to take fuller advantage of classes—and leverage more flexible coding structures like that of the next section.

Multiple Iterators on One Object

Earlier, it was mentioned in passing that the *iterator* object (with a `__next__`) produced by an iterable may be defined as a separate class with its own state information to more directly support multiple active iterations over the same data. To understand this better, consider what happens when we step across a built-in type like a string:

```
>>> S = 'ace'
>>> for x in S:
        for y in S:
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Here, the outer loop grabs an iterator from the string by calling `iter`, and each nested loop does the same to get an independent iterator. Because each active iterator has its own state information, each loop can maintain its own position in the string, regardless of any other active loops. Moreover, we're not required to make a new string or convert to a list each time; the single string object itself supports multiple scans.

We saw related examples earlier, in Chapters 14 and 20. For instance, generator functions and expressions, as well as built-ins like `map` and `zip`, proved to be single-iterator objects, thus supporting a single active scan. By contrast, the `range` built-in, and other built-in types like lists, support multiple active iterators with independent positions.

When we code user-defined iterables with classes, it's up to us to decide whether we will support a single active iteration or many. To achieve the multiple-iterator effect, `__iter__` simply needs to define a new stateful object for the iterator instead of returning `self` for each iterator request.

For example, the class `SkipObject` in Example 30-3 defines an iterable object that skips every other item on iterations. Because its iterator object is created anew from a supplemental class for each iteration, it supports multiple active loops directly.

Example 30-3. skipper.py

```
class SkipObject:
    def __init__(self, wrapped):                    # Sav
        self.wrapped = wrapped

    def __iter__(self):
        return SkipIterator(self.wrapped)           # New

class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped                      # Ite
        self.offset  = 0

    def __next__(self):
        if self.offset >= len(self.wrapped):        # Ter
            raise StopIteration
        else:
            item = self.wrapped[self.offset]        # els
            self.offset += 2
            return item

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)                     # Mak
    I = iter(skipper)                               # Mak
    print(next(I), next(I), next(I))                # Vis

    for x in skipper:                   # for calls __iter
        for y in skipper:               # Nested fors call
            print(x + y, end=' ')       # Each iterator ha
```

When run, this example works like the earlier nested loops with built-in strings. Each active loop has its own position in the string because each obtains an independent iterator object that records its own state information:

```
$ python3 skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

By contrast, our earlier `Squares` class of Example 30-2 supports just one active iteration unless we call `Squares` again in nested loops to obtain new

objects (else the outer `for` loop would run just once). Here, there is just one `SkipObject` iterable, with multiple iterator objects created from it.

Classes versus slices

As before, we could achieve similar results with built-in tools—for example, slicing with a third bound to skip items:

```
>>> S = 'abcdef'
>>> for x in S[::2]:
        for y in S[::2]:              # New objects on e
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

This isn't quite the same, though, for two reasons. First, each slice expression here will *physically store* the result list all at once in memory; iterables, on the other hand, produce just one value at a time, which can save substantial space and startup time for large result lists.

Second, slices produce *new objects*, so we're not really iterating over the same object in multiple places here. To be closer to the class, we would need to make a single object to step across by slicing ahead of time:

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
        for y in S:                   # Same object, new
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

This is more similar to our class-based solution, but it still stores the slice result in memory all at once (there is no generator form of built-in slicing today), and it's only equivalent for this particular case of skipping every other item.

Because user-defined iterables coded with classes can do anything a class can do, they are much more general than this example may imply. Though such generality is not required in all applications, user-defined iterables are a powerful tool—they allow us to make arbitrary objects look and feel like the other sequences and iterables we have met in this book. We could use this technique with a database object, for example, to support iterations over large database fetches, with multiple cursors into the same query result.

Coding Alternative: __iter__ Plus yield

Now for something more implicit—but potentially useful nonetheless. In some applications, it's possible to minimize coding requirements for user-defined iterables by *combining* the `__iter__` method we're exploring here and the `yield` generator function statement we studied in Chapter 20. Because generator functions *automatically* save local-variable state and create required iterator methods, they fit this role well and complement the state retention and other utility we get from classes.

As a quick review, recall that any function that contains a `yield` statement is turned into a generator function. When called, it returns a new *generator object* with automatic retention of local scope and code position; an automatically created `__iter__` method that simply returns itself; and an automatically created `__next__` method that starts the function or resumes it where it last left off:

```
>>> def gen(x):
        for i in range(x): yield i ** 2

>>> G = gen(5)                  # Create a generator with
>>> G.__iter__() is G           # Both methods exist on th
True
>>> I = iter(G)                 # Runs __iter__: generator
>>> next(I), next(I)            # Runs __next__
(0, 1)
>>> list(gen(5))                # Iteration tools automati
[0, 1, 4, 9, 16]
```

This is still true even if the generator function with a `yield` happens to be a *method* named `__iter__` . Whenever invoked by an iteration tool, such a method will return a new *generator* object with the requisite `__next__` . As

an added bonus, generator functions coded as methods in classes have access to saved state in *both* instance attributes and local-scope variables.

For example, the class in Example 30-4 is equivalent to the initial `Squares` user-defined iterable we coded earlier in Example 30-2, but noticeably shorter (4 lines, for anyone counting).

Example 30-4. squares_yield.py

```
class Squares:                                  # __iter
    def __init__(self, start, stop):           # __next
        self.start = start
        self.stop  = stop

    def __iter__(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

As before, `for` loops and other iteration tools iterate through instances of this class automatically:

```
$ python3
>>> from squares_yield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')      # Ru

1 4 9 16 25
```

And as always, we can also look under the hood to see how this actually works in iteration tools like `for` . Running our class instance through `iter` obtains the result of calling `__iter__` as usual. In this case, though, the result is a generator object with an automatically created `__next__` of the same sort we always get when calling a generator function that contains a `yield` . The only difference here is that the generator function is automatically called on `iter` . Invoking the result object's `next` interface produces results on demand:

```
>>> S = Squares(1, 5)           # Runs __init__: class s
>>> S
<squares_yield.Squares object at 0x109e30b90>
```

```
>>> I = iter(S)                # Runs __iter__: returns
>>> I
<generator object Squares.__iter__ at 0x109ecb3e0>
>>> next(I)
1
>>> next(I)                    # Runs generator's __nex
4
…etc…
>>> next(I)                    # Generator has both ins
StopIteration
```

It may also help to notice that we could name the generator method something other than `__iter__` and call manually to iterate— `Squares(…).gen()`, for example. Using the `__iter__` name invoked automatically by iteration tools simply skips a manual attribute fetch and call step. Example 30-5 demos the idea.

Example 30-5. squares_yield_manual.py

```
import squares_yield                    # Reuse prior

class Squares(squares_yield.Squares):   # Non __iter_
    def gen(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

The example also imports Example 30-4 to inherit its constructor. When run, its results are the same, but we must call the `gen` method explicitly to fetch an iterable—a step that `__iter__` automates and obviates:

```
$ python3
>>> from squares_yield_manual import Squares
>>> for i in Squares(1, 5).gen(): print(i, end=' ')
…same results…

>>> S = Squares(1, 5)
>>> I = iter(S.gen())          # Call generator manuall
>>> next(I)
…same results…
```

Coding the generator as `__iter__` instead cuts out the middleperson in your code, though both schemes ultimately wind up creating a new generator object for each iteration:

- With `__iter__`, iteration triggers `__iter__`, which returns a new generator with `__next__`.
- Without `__iter__`, your code calls to make a generator, which returns itself for `__iter__`.

See Chapter 20 for more on `yield` and generators if this is puzzling, and compare it with the more explicit `__next__` version in Example 30-2 earlier. If you do, you'll notice that the *squares_yield.py* version is 4 lines shorter (7 versus 11, not counting whitespace). In a sense, this scheme reduces class coding requirements much like the closure functions of Chapter 17, but in this case does so with a *combination* of functional and OOP techniques instead of an alternative to classes. For example, the generator method still leverages `self` attributes.

This may also seem like one too many levels of *magic* to some observers—it relies on both the iteration protocol and the object creation of generators, both of which are highly implicit (in contradiction of longstanding Python goals). Opinions aside, it's important to understand the non-`yield` flavor of class iterables too, because it's explicit, general, and sometimes broader in scope.

Still, the `__iter__` / `yield` technique may prove effective in cases where it applies. It also comes with a substantial advantage—as the next section explains.

Multiple iterators with yield

Besides its code conciseness, the user-defined class iterable of the prior section based upon the `__iter__` / `yield` combination has an important added bonus—it also supports *multiple active iterators* automatically. This naturally follows from the fact that each call to `__iter__` is a new call to a generator function, which returns a new generator with its own copy of the local scope for state retention. Using Example 30-4 again:

```
$ python3
>>> from squares_yield import Squares   # Using the __i
>>> S = Squares(1, 5)
```

```
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> K = iter(S)                          # With yield, n
>>> next(K)
1
>>> next(I)                              # I is indepenc
9
```

Although generator functions are single-scan iterables by nature, the implicit calls to `__iter__` in iteration tools make new generators supporting new independent scans:

```
>>> S = Squares(1, 3)
>>> for i in S:                          # Each "for" cc
        for j in S:
            print(f'{i}:{j}', end=' ')

1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

To do the same without `yield` requires a supplemental class that stores iterator state explicitly and manually, using techniques of the preceding section. Per Example 30-6, this grows to 15 lines: 8 more than with `yield`.

Example 30-6. squares_nonyield.py

```
class Squares:
    def __init__(self, start, stop):                #
        self.start = start                           #
        self.stop  = stop

    def __iter__(self):
        return SquaresIter(self.start, self.stop)

class SquaresIter:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop  = stop

    def __next__(self):
        if self.value == self.stop:
            raise StopIteration
```

```
        self.value += 1
        return self.value ** 2
```

This works the same as the `yield` multiscan version, but with more—and more explicit—code:

```
$ python3
>>> from squares_nonyield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')

1 4 9 16 25

>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> K = iter(S)                          # Multiple ite
>>> next(K)
1
>>> next(I)
9

>>> S = Squares(1, 3)
>>> for i in S:                          # Each "for" c
        for j in S:
            print(f'{i}:{j}', end=' ')

1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

Finally, the generator-based approach could similarly remove the need for an extra iterator class in the prior item-skipper example, *skipper.py* of Example 30-3, thanks to its automatic methods and local variable state retention. Example 30-7 codes the mod, which checks in at 9 lines versus the original's 16, sans the original's self-test.

Example 30-7. skipper_yield.py

```
class SkipObject:                         # Another _
    def __init__(self, wrapped):          # Instance
        self.wrapped = wrapped            # Local scc
```

```
    def __iter__(self):
        offset = 0
        while offset < len(self.wrapped):
            item = self.wrapped[offset]
            offset += 2
            yield item
```

This works the same as the non- `yield` multiscan version, but with less—and less explicit—code:

```
$ python3
>>> from skipper_yield import SkipObject
>>> skipper = SkipObject('abcdef')
>>> I = iter(skipper)
>>> next(I); next(I); next(I)
'a'
'c'
'e'
>>> for x in skipper:                # Each "for" calls
        for y in skipper:
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Of course, these are all artificial examples that could be replaced with simpler tools like comprehensions, and their code may or may not scale up in kind to more realistic tasks. Study these alternatives to see how they compare. As so often in programming, the best tool for the job will likely be the best tool for your job.

Membership: __contains__, __iter__, and __getitem__

The iteration story is even richer than told thus far. Operator overloading is often *layered*: classes may provide specific methods or more general alternatives used as fallback options. We've already seen this for general iteration (`__iter__` or else `__getitem__`), and will encounter another example ahead when we meet Boolean values.

Also in the iterations domain, classes can implement the `in` membership operator as an iteration, using either the `__iter__` or `__getitem__` methods. To support more specific membership, though, classes may code a `__contains__` method—when present, this method is preferred over `__iter__`, which is preferred over `__getitem__`. The `__contains__` method should define membership as applying to keys for a *mapping* (and can use quick lookups) and as a search for *sequences*.

Consider the class `Iters` in Example 30-8. It codes all three methods and tests membership and various iteration tools applied to an instance. To demo, its methods print trace messages when called, its self-test code cycles through tests coded with `match` to call them out, and its methods' traces show up before those of its tests.

Example 30-8. contains.py

```
def trace(msg, end=''):
    print(f'{msg} ', end=end)                  # print s

class Iters:
    def __init__(self, value):
        self.data = value

    def __getitem__(self, i):                  # Fallbac
        trace(f'@get[{i}]')                    # Also fc
        return self.data[i]

    def __iter__(self):                        # Preferr
        trace('@iter')                         # Allows
        self.ix = 0
        return self

    def __next__(self):
        trace('@next')
        if self.ix == len(self.data): raise StopIterati
        item = self.data[self.ix]
        self.ix += 1
        return item

    def __contains__(self, x):                 # Preferr
        trace('@contains')
        return x in self.data
```

```
def self_test(Iters):
    X = Iters([1, 2, 3, 4])
    tests = 'In', 'For', 'Comp', 'Map', 'Manual'
    for test in tests:
        trace(test.ljust(max(map(len, tests)) + 1))
        match test:
            case 'In':
                trace(3 in X)
            case 'For':
                for i in X:
                    trace(i, end='| ')
            case 'Comp':
                trace([i ** 2 for i in X])
            case 'Map':
                trace(list(map(bin, X)))
            case 'Manual':
                I = iter(X)
                while True:
                    try:
                        trace(next(I), end='| ')
                    except StopIteration:
                        break
        print()

if __name__ == '__main__': self_test(Iters)
```

As is, the class in this file has an `__iter__` that supports only a single active scan at any point in time (e.g., nested loops won't work) because each iteration attempt resets the scan cursor to the front. Now that you know about `yield` in iteration methods, you should be able to tell that Example 30-9 is equivalent but allows multiple active scans—and judge for yourself whether its more implicit nature is worth the nested-scan support (and 5 lines shaved).

Example 30-9. contains-yield.py

```
from contains import *

class ItersYield(Iters):
    def __iter__(self):                    # Preferred
        trace('@iter @next')               # Allows mul
        for x in self.data:                # Implicit g
            yield x
            trace('@next')
```

```
if __name__ == '__main__': self_test(ItersYield)      #
```

When either version of this file runs, its output is as follows—the specific `__contains__` intercepts membership, the general `__iter__` catches other iteration tools such that `__next__` (whether explicitly coded or implied by `yield`) is called repeatedly, and `__getitem__` is never called:

```
$ python3 contains.py
In      @contains True
For     @iter @next 1 | @next 2 | @next 3 | @next 4 | @
Comp    @iter @next @next @next @next @next [1, 4, 9, 1
Map     @iter @next @next @next @next @next ['0b1', '0b
Manual  @iter @next 1 | @next 2 | @next 3 | @next 4 | @
```

Watch, though, what happens to this code's output if we comment out its `__contains__` method—membership is now routed to the general `__iter__` instead (add triple quotes above and below the method to test live):

```
$ python3 contains.py
In      @iter @next @next @next True
For     @iter @next 1 | @next 2 | @next 3 | @next 4 | @
Comp    @iter @next @next @next @next @next [1, 4, 9, 1
Map     @iter @next @next @next @next @next ['0b1', '0b
Manual  @iter @next 1 | @next 2 | @next 3 | @next 4 | @
```

And finally, here is the output if *both* `__contains__` and `__iter__` are commented out—the indexing `__getitem__` fallback is called with successively higher indexes until it raises `IndexError`, for membership and other iteration tools:

```
$ python3 contains.py
In      @get[0] @get[1] @get[2] True
For     @get[0] 1 | @get[1] 2 | @get[2] 3 | @get[3] 4 |
Comp    @get[0] @get[1] @get[2] @get[3] @get[4] [1, 4,
Map     @get[0] @get[1] @get[2] @get[3] @get[4] ['0b1',
Manual  @get[0] 1 | @get[1] 2 | @get[2] 3 | @get[3] 4 |
```

As we've seen, the `__getitem__` method does other work too: besides iterations, it also intercepts explicit indexing as well as slicing. Slice expressions trigger `__getitem__` with a slice object containing bounds, both for built-in types and user-defined classes, so slicing is automatic in our class. With `__iter__` enabled or not:

```
$ python3
>>> from contains import Iters
>>> X = Iters('hack')
>>> X[0]                                # Indexing: __geti
@get[0] 'h'

>>> X[1:]                               # Slicing: __getit
@get[slice(1, None, None)] 'ack'
>>> X[:-1]
@get[slice(None, -1, None)] 'hac'
```

Iterations, though, are more selective—we get the first of the following if `__iter__` is still commented out and the second if it's not (be sure to restart or `reload` after the file mod either way):

```
>>> list(X)
@get[0] @get[1] @get[2] @get[3] @get[4] ['h', 'a', 'c',

>>> list(X)
@iter @next @next @next @next @next ['h', 'a', 'c', 'k'
```

In more realistic iteration use cases that are not sequence-oriented, though, the `__iter__` method may be easier to write since it must not manage an integer index, and `__contains__` allows for membership optimization as a special case. While iteration is a rich topic, it's time to move on to the next stop on our overloading tour.

Attribute Access: __getattr__ and __setattr__

In Python, classes can also intercept basic *attribute* access (a.k.a. qualification) when needed or useful. Specifically, for an `object` created

from a class, the dot operator expression `object.attribute` can be implemented by your code too, for reference, assignment, and deletion contexts. We explored a limited example in this category which rerouted attribute fetches in the tutorial of Chapter 28, but will review and expand on the topic here.

Attribute Reference

The `__getattr__` method intercepts attribute references. It's called with the attribute name as a string whenever you try to qualify an instance with an *undefined* (nonexistent) attribute name. It is *not* called if Python can find the attribute using its inheritance tree search procedure.

Because of its behavior, `__getattr__` is useful as a hook for responding to attribute requests in a generic fashion. It's commonly used to delegate calls to embedded (or "wrapped") objects from a proxy controller object—of the sort introduced in Chapter 28's introduction to *delegation*. This method can also be used to adapt classes to an interface or add *accessors* for data attributes after the fact—logic in a method that validates or computes an attribute after it's already being used with simple dot notation (possibly after a rename of the original).

The basic mechanism underlying these goals is straightforward—the following class catches attribute references, computing the value for one dynamically, and triggering an error for others unsupported with the `raise` statement described earlier in this chapter for iterators (and again, fully covered in Part VII):

```
>>> class Empty:
        def __getattr__(self, attrname):           # Or
            if attrname == 'age':
                return 40
            else:
                raise AttributeError(attrname)

>>> X = Empty()
>>> X.age                   # Becomes X.__getattr__('age
40
>>> X.name                  # Unsupported attribute
```

```
…error text omitted…
AttributeError: name
```

Here, the `Empty` class and its instance `X` have no real attributes of their own, so the access to `X.age` gets routed to the `__getattr__` method; `self` is assigned the instance (`X`), and `attrname` is assigned the undefined attribute name string (`'age'`). The class makes `age` look like a real attribute by returning a real value as the result of the `X.age` qualification expression (`40`). In effect, `age` becomes a *dynamically computed* attribute —its value is formed by running code, not fetching an object.

For attributes that the class doesn't know how to handle, `__getattr__` raises the built-in `Attribute Error` exception to tell Python that these are bona fide undefined names; asking for `X.name` triggers the error. You'll see `__getattr__` again when we explore delegation and properties at work in the next two chapters; let's move on to related tools here.

Attribute Assignment and Deletion

In the same department, the `__setattr__` intercepts *all* attribute assignments. If this method is defined or inherited, `self.attr = value` becomes `self.__setattr__('attr', value)` . Like `__getattr__` , this allows your class to catch attribute changes and validate or transform as desired.

This method is a bit trickier to use, though, because assigning to any `self` attributes within `__setattr__` calls `__setattr__` again, potentially causing an infinite *recursion loop* (and a fairly quick stack overflow exception!). In fact, this applies to all `self` attribute assignments anywhere in the class—all are routed to `__setattr__` , even those in other methods, and those to names other than that which may have triggered `__setattr__` in the first place. So be warned: this catches *all* attribute assignments.

If you wish to use this method, you can avoid loops by coding instance attribute assignments as assignments to attribute dictionary keys. That is, use `self.__dict__['name'] = x` , not `self.name = x` ; because you're not assigning to `__dict__` itself, this avoids the loop:

```
>>> class Accesscontrol:
        def __setattr__(self, attr, value):
```

```
            if attr == 'age':
                self.__dict__[attr] = value + 10       #
            else:
                raise AttributeError(attr + ' not allow
```

```
>>> X = Accesscontrol()
>>> X.age = 40                  # Becomes X.__setattr__('
>>> X.age                       # Found in __dict__ as us
50
>>> X.name = 'Pat'              # Unsupported attribute
…text omitted…
AttributeError: name not allowed
```

If you change the `__dict__` assignment within this class to either of the following, it triggers the infinite recursion loop and exception—both dot notation and its `setattr` built-in function equivalent (the assignment analog of `getattr`) fail when `age` is assigned outside the class:

```
self.age = value + 10                          # Loop
setattr(self, attr, value + 10)                # Loop
```

An assignment to any other `self` attribute within the class triggers a recursive `__setattr__` call too, though in this class ends less dramatically in the manual `AttributeError` exception:

```
self.other = 99                                # Recu
```

It's also possible to avoid recursive loops in a class that uses `__setattr__` by *rerouting* any attribute assignments to a higher superclass with a call, instead of assigning keys in `__dict__`:

```
object.__setattr__(self, attr, value + 10)    # OK:
```

This uses an explicit-class call to the implied `object` superclass above all topmost classes (and has a `super().__setattr__(…)` equivalent sans `self` per the sidebar "The super Alternative"). In fact, this alternative may be *required* in some classes per the upcoming note, though this is rare in practice.

A third attribute-management method, `__delattr__`, is passed the attribute name string and invoked on all attribute deletions (i.e., `del object.attr`). Like `__setattr__`, it must avoid recursive loops by running attribute deletions within the class using `__dict__`, or rerouting them to a superclass.

NOTE

Attribute outliers: Preceding chapters mentioned that attributes coded with advanced class tools such as *slots* and *properties* are not physically stored in the instance's `__dict__` namespace dictionary—and *slots* may even preclude a `__dict__` altogether. As noted, `dir` and `getattr` might be needed for listing and fetching attributes in classes using these tools, but assignment is similarly impacted: to support such "virtual" attributes, `__setattr__` may need to use the `object.__setattr__` scheme shown here, not `self.__dict__` indexing. You'll learn much more about these attribute tools in upcoming chapters.

Other Attribute-Management Tools

The three attribute-access overloading methods we've met so far allow you to control or specialize access to attributes in your objects. They tend to play highly specialized roles, some of which we'll explore later in this book. For another example of `__getattr__` at work, see Chapter 28's *person-composite.py* (Example 28-11).

And for future reference, keep in mind that there are other ways to manage attribute access in Python:

- The `__getattribute__` method intercepts *all* attribute fetches, not just those that are undefined; like `__setattr__`, it must avoid loops for other attribute fetches in the class, usually with `object` rerouting.
- The `property` built-in function allows us to associate methods with fetch and set operations on a *specific* class attribute; it cannot catch accesses generically but can define what some do.
- *Descriptors* provide a protocol for associating `__get__` and `__set__` methods of a class with accesses to a *specific* instance or class attribute; they are as focused as `property` and, in fact, are used to implement it.
- *Slots* attributes are declared in classes but create implicit storage in each instance; if present, generic tools may need to list, fetch, and assign with schemes described in the preceding note.

Because these are all advanced tools that are not of interest to every Python programmer, we'll defer the details of other attribute-management techniques until Chapter 32, and await their focused coverage in Chapter 38.

Emulating Privacy for Instance Attributes: Part 1

As another use case for attribute tools, the code in Example 30-10—file *private0.py*—generalizes the previous example, to allow each subclass to have its own list of private names that cannot be *assigned* to its instances (and raises a built-in exception with `raise`, which you'll have to take on faith until Part VII).

Example 30-10. private0.py

```
class Privacy:
    def __setattr__(self, attr, value):            # C
        if attr in self.privates:
            raise NameError(f'{attr!r} for {self}')
        else:
            self.__dict__[attr] = value             # A

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Pat'               # T

if __name__ == '__main__':
    x = Test1()
    x.name = 'Sue'      # Works
    print(x.name)
   #x.age = 40          # Fails

    y = Test2()
    y.age = 30          # Works
    print(y.age)
   #y.name = 'Bob'      # Fails
```

This is a first-cut solution for an implementation of *attribute privacy* in Python—disallowing changes to attribute names outside a class. Although

Python doesn't support private declarations per se, techniques like this can emulate much of their purpose.

This particular attempt, though, is a partial—and even clumsy—solution. To make it more effective, we must augment it to allow classes to set their private attributes more naturally, without having to go through `__dict__` each time, as the constructor must do here to avoid triggering `__setattr__` and an exception. A better and more complete approach might require a wrapper ("proxy") class to check for private attribute accesses made outside the class only and a `__getattr__` to validate attribute fetches too.

We'll postpone a more complete solution to attribute privacy until Chapter 39, where we'll use *class decorators* to intercept and validate attributes more generally. Even though privacy can be emulated this way, though, it almost never is in practice. Python programmers are able to write large OOP frameworks and applications without private declarations—an interesting finding about access controls in general that is beyond the scope of our purposes here.

Still, catching attribute references and assignments is generally a useful technique; it supports *delegation*, a design technique that allows controller objects to wrap up embedded objects, add new behaviors, and route other operations back to the wrapped objects. Because they involve design topics, we'll revisit delegation and wrapper classes in the next chapter. Here, it's time to move ahead in the operator overloading domain.

String Representation: __repr__ and __str__

Our next methods deal with display formats—a topic we've already explored in prior chapters but will summarize and formalize here. To serve as a guinea pig, the following codes the `__init__` constructor and the `__add__` overload method, both of which we've already seen (`+` is an in-place operation here, just to show that it can be; this may be better coded as a named method per Chapter 27, or the in-place `__iadd__` covered ahead). As we've learned, the default display of instance objects for a class like this is neither generally useful nor aesthetically pretty:

```
>>> class adder:
        def __init__(self, value=0):
            self.data = value                    # Init
        def __add__(self, other):
            self.data += other                   # Add

>>> x = adder()                                  # Defa
>>> print(x)                                     # str
<__main__.adder object at 0x106110c80>
>>> x                                            # repr
<__main__.adder object at 0x106110c80>
```

But coding or inheriting string representation methods allows us to customize the display—as in the following, which defines a `__repr__` method in a subclass that returns a string representation for its instances:

```
>>> class addrepr(adder):                        # Inhe
        def __repr__(self):                      # Add
            return f'addrepr({self.data})'       # Conv

>>> x = addrepr(2)
>>> x + 1
>>> x                                  # Runs __repr_
addrepr(3)
>>> print(x)                           # Runs __repr_
addrepr(3)
>>> str(x), repr(x)                    # Runs __repr_
('addrepr(3)', 'addrepr(3)')
```

If defined, `__repr__` (or its close relative, `__str__`) is called automatically when class instances are printed or converted to strings. These methods allow you to define a better display format for your objects than the default instance display. Here, `__repr__` uses basic string formatting to convert the managed `self.data` object to a more human-friendly string for display.

Why Two Display Methods?

So far, this section has largely been review. But while these methods are generally straightforward to use, their roles and behavior have some subtle

implications for both design and coding. In particular, Python provides its two display methods to support alternative displays for different audiences:

- `__str__` is tried first for the `print` operation and the `str` built-in function (the internal equivalent of which `print` runs). It generally should return a *user-friendly* display.
- `__repr__` is used in all other contexts: for interactive echoes, the `repr` function, and nested appearances, as well as by `print` and `str` if no `__str__` is present. It should generally return an *as-code* string that could be used to re-create the object or a *detailed* display for developers.

That is, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. This means you can code a `__repr__` to define a single display format used everywhere and may code a `__str__` to either support `print` and `str` exclusively or to provide an alternative display for them.

General tools may also prefer `__str__` to leave other classes the option of adding an alternative `__repr__` display for use in other contexts, as long as `print` and `str` displays suffice for the tool. Conversely, a general tool that codes a `__repr__` still leaves clients the option of adding alternative displays with a `__str__` for `print` and `str` . In other words, if you code either, the other is available for an additional display. In cases where the choice isn't clear, use `__str__` for higher-level displays and `__repr__` for lower-level displays and all-inclusive roles.

Let's write some code to illustrate these two methods' distinctions in more concrete terms. The prior example in this section showed how `__repr__` is used as the fallback option in many contexts. However, while printing falls back on `__repr__` if no `__str__` is defined, the inverse is not true—other contexts, such as interactive echoes, use `__repr__` only and don't try `__str__` at all:

```
>>> class addstr(adder):
        def __str__(self):                          # __st
            return f'[Value: {self.data}]'          # Conv

>>> x = addstr(3)
>>> x + 1
>>> x                                               # Defa
<__main__.addstr object at 0x106111b20>
```

```
>>> print(x)                                    # Runs
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr object at 0x106111b20>
```

Because of this, `__repr__` may be best if you want a *single* display for all contexts. By defining *both* methods, though, you can support different displays in different contexts—for example, a class's end-user display with `__str__`, and a class's developer display with `__repr__`. In effect, `__str__` simply overrides `__repr__` for more user-friendly display contexts:

```
>>> class addboth(adder):
        def __str__(self):
            return f'[Value: {self.data}]'       # User
        def __repr__(self):
            return f'addboth({self.data})'       # As-c

>>> x = addboth(4)
>>> x + 1
>>> x                                           # Runs
addboth(5)
>>> print(x)                                    # Runs
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

Bonus: your classes' `__str__` and `__repr__` are also run automatically by all three *string-formatting* tools of Chapter 7. This may not be a shocker, given these tools are defined to work like `str` and `repr` in this role, but display overloading is not just about REPL echoes and `print`:

```
>>> f'{x!s} {x!r}',  '{!s} {!r}'.format(x, x),  '%s %r'
('[Value: 5] addboth(5)', '[Value: 5] addboth(5)', '[Va
```

Display Usage Notes

Though generally simple to use, three points regarding these display methods are worth calling out here. First, keep in mind that `__str__` and `__repr__` must both return *strings*; other result types are not converted and

raise errors, so be sure to run them through a to-string converter (e.g., `str` or `f'…'`) if needed.

Second, depending on a container's string-conversion logic, the user-friendly display of `__str__` might only apply when objects appear at the top level of a print operation; objects *nested* in larger objects might still print with their `__repr__` or its default. The following illustrates both of these points:

```
>>> class Printer:
        def __init__(self, val):
            self.val = val
        def __str__(self):                  # Used for
            return str(self.val)            # Convert t

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)                 # __str__ r
                                            # But not w
2
3
>>> print(objs)
[<__main__.Printer object at 0x106110c80>, <__main__.Pr
>>> objs
[<__main__.Printer object at 0x106110c80>, <__main__.Pr
```

To ensure that a custom display is run in all contexts regardless of the container, code `__repr__`, not `__str__`; the former is run in all cases if the latter doesn't apply, including nested appearances:

```
>>> class Printer:
        def __init__(self, val):
            self.val = val
        def __repr__(self):                 # __repr__
            return str(self.val)            # __repr__

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)                 # No __str_

2
3
>>> print(objs)                             # Runs __re
[2, 3]
```

```
>>> objs
[2, 3]
```

Third, and perhaps most subtle, the display methods also have the potential to trigger infinite *recursion loops* in rare contexts—because some objects' displays include displays of other objects, it's not impossible that a display may trigger a display of an object being displayed, and thus loop. This is rare and obscure enough to skip here but watch for an example of this looping potential to appear for these methods in a note near the end of the next chapter about its *listinherited.py* class of Example 31-12, where `__repr__` can loop.

In practice, `__str__` and its more inclusive relative, `__repr__`, seem to be the second most commonly used operator-overloading methods in Python scripts, behind `__init__`. Anytime you can print an object and see a custom display, one of these two tools is probably in use. For additional examples of these tools at work and the design trade-offs they imply, see Chapter 28's case study and Chapter 31's class-lister mix-ins, as well as their role in Chapter 35's exception classes, where `__str__` is required over `__repr__`.

Right-Side and In-Place Ops: __radd__ and __iadd__

Our next group of overloading methods extends the functionality of binary operator methods such as `__add__` and `__sub__` (called for `+` and `-`), which we've already seen. As mentioned earlier, part of the reason there are so many operator-overloading methods is that they come in multiple flavors—for every binary expression, we can implement a *left*, *right*, and *in-place* variant. Though defaults are also applied if you don't code all three, your objects' roles dictate how many variants you'll need to code.

Right-Side Addition

For instance, the `__add__` methods coded so far technically do not support the use of instance objects on the *right* side of the `+` operator:

```
>>> class Adder:
        def __init__(self, value=0):
```

```
            self.data = value
        def __add__(self, other):
            return self.data + other

>>> x = Adder(5)
>>> x + 2
7
>>> 2 + x
TypeError: unsupported operand type(s) for +: 'int' and
```

To implement more general expressions and hence support *commutative*-style operators, code the `__radd__` method as well. Python calls `__radd__` only when the object on the right side of the `+` is your class instance, but the object on the left is not an instance of your class. The `__add__` method for the object on the left is called instead in all other cases. As a demo, consider the class in Example 30-11 (which we'll be adding to in a moment).

Example 30-11. commuter.py (start)

```
class Commuter1:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val
```

Because this defines both left- and right-side overloads for `+`, instances can appear on either side, or both:

```
>>> from commuter import Commuter1
>>> x = Commuter1(88)
>>> y = Commuter1(99)

>>> x + 1                       # __add__: instance + no
add 88 1
89
>>> 1 + y                       # __radd__: noninstance
radd 99 1
```

```
100
>>> x + y                       # __add__: instance + ir
add 88 <commuter.Commuter1 object at 0x1011b63f0>
radd 99 88
187
```

Notice how the order is reversed in `__radd__`: `self` is really on the right of the `+`, and `other` is on the left. Also note that `x` and `y` are instances of the same class here; when instances of different classes appear mixed in an expression, Python prefers the class of the one on the left. When we add the two instances of this class together, Python runs `__add__`, which in turn triggers `__radd__` by simplifying the left operand and re-adding.

Reusing __add__ in __radd__

For truly commutative operations that do not require special-casing by position, it is also sometimes sufficient to reuse `__add__` for `__radd__`, either by calling `__add__` directly; by swapping order and re-adding to trigger `__add__` indirectly; or by simply assigning `__radd__` to be an alias for `__add__` at the top level of the `class` statement (i.e., in the class's scope). The alternatives in Example 30-12 implement all three of these schemes and return the same results as the original—though the last saves an extra call or dispatch and hence may be quicker (in all, `__radd__` is run when `self` is on the right side of a `+`).

Example 30-12. commuter.py (continued)

```
class Commuter2:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        return self.__add__(other)              # Call

class Commuter3:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
```

```
            print('add', self.val, other)
            return self.val + other

        def __radd__(self, other):
            return self + other                         # Swap

    class Commuter4:
        def __init__(self, val):
            self.val = val

        def __add__(self, other):
            print('add', self.val, other)
            return self.val + other

        __radd__ = __add__                              # Alias
```

In all these, right-side instance appearances trigger the single, shared `__add__` method, passing the right operand to `self`, to be treated the same as a left-side appearance. Run these on your own for more insight; their names differ, but their usage and returned values are the same as the original `Commuter1` of Example 30-11.

Propagating class type

In more realistic classes where the class type may need to be propagated in results, things can become trickier: type testing may be required to tell whether it's safe to convert and thus avoid nesting. For instance, without the built-in `isinstance` test in Example 30-13, we could wind up with a `Commuter5` whose `val` is another `Commuter5` when two instances are added and `__add__` triggers `__radd__`.

Example 30-13. commuter.py (continued)

```
    class Commuter5:                                    # Propc
        def __init__(self, val):
            self.val = val

        def __add__(self, other):
            if isinstance(other, Commuter5):            # Type
                other = other.val
            return Commuter5(self.val + other)          # Else
```

```
    def __radd__(self, other):
        return Commuter5(other + self.val)

    def __repr__(self):
        return f'Commuter5({self.val})'
```

When this version is run, `+` results retain the `Commuter5` type for future operations:

```
>>> from commuter import Commuter5
>>> x = Commuter5(88)
>>> y = Commuter5(99)

>>> x
Commuter5(88)
>>> x + 1                       # Result is another Comm
Commuter5(89)
>>> 1 + y
Commuter5(100)

>>> z = x + y                   # Not nested: doesn't re
>>> z
Commuter5(187)
>>> z + 10
Commuter5(197)
>>> z + z
Commuter5(374)
>>> z + z + 1
Commuter5(375)
```

The need for the `isinstance` type test here is very subtle—uncomment, run, and trace to see why it's required. If you do, you'll see that the last part of the preceding test winds up differing and nesting objects—which still do the math correctly but kick off pointless recursive calls to simplify their values and extra constructor calls to build results:

```
>>> z = x + y                   # With isinstance test+c
>>> z
Commuter5(Commuter5(187))
>>> z + 10
Commuter5(Commuter5(197))
>>> z + z
```

```
Commuter5(Commuter5(Commuter5(Commuter5(374))))
>>> z + z + 1
Commuter5(Commuter5(Commuter5(Commuter5(375))))
```

Another way out of this dilemma is to test and simplify in the *constructor* instead, per Example 30-14.

Example 30-14. commuter.py (continued)

```
class Commuter6:                                # Propc
    def __init__(self, val):
        if isinstance(val, Commuter6):          # Type
            self.val = val.val
        else:
            self.val = val

    def __add__(self, other):
        return Commuter6(self.val + other)

    def __radd__(self, other):
        return Commuter6(other + self.val)

    def __repr__(self):
        return f'Commuter6({self.val})'
```

This last version works the same as the non-nesting `Commuter5` . To test all of this section's classes, the rest of *commuter.py* in Example 30-15 looks and runs like this—like functions, classes can again be used in tuples naturally because they are first-class objects.

Example 30-15. commuter.py (conclusion)

```
if __name__ == '__main__':
    for klass in (Commuter1, Commuter2, Commuter3, Comm
        print('-' * 50)
        x = klass(88)
        y = klass(99)
        print(x + 1)
        print(1 + y)
        print(x + y)
```

```
$ python3 commuter.py
--------------------------------------------------
```

```
add 88 1
89
radd 99 1
100
add 88 <__main__.Commuter1 object at 0x101edc2f0>
radd 99 88
187
------------------------------------------------------
…etc…
------------------------------------------------------
Commuter6(89)
Commuter6(100)
Commuter6(187)
```

Experiment with these classes on your own for more insight. Aliasing `__radd__` to `__add__` in `Commuter5` and `Commuter6`, for example, works and saves a line but doesn't prevent object nesting without these classes' `isinstance` tests. See also Python's manuals for a discussion of other options in this domain; for example, classes may also return the special `NotImplemented` object for unsupported operands to influence method selection (this is treated as though the method were not defined—and differs from the prior chapter's `NotImplementedError`).

In-Place Addition

To also implement `+=` in-place augmented addition, code either an `__iadd__` or an `__add__`. The latter is used if the former is absent. In fact, the prior section's `Commuter` classes already support `+=` for this reason—Python runs `__add__` and assigns the result manually. The `__iadd__` method, though, allows for more efficient in-place changes to be coded where applicable; its return value is assigned to the target on the left of the `+=`:

```
>>> class Number:
        def __init__(self, val):
            self.val = val
        def __iadd__(self, other):             # __iadd
            self.val += other                  # Usuall
            return self                        # Else N

>>> x = Number(5)
>>> x += 1
>>> x += 1
```

```
>>> x.val
7
```

For mutable objects, this method can often specialize for quicker in-place changes:

```
>>> y = Number([1])                          # In-pla
>>> y += [2]
>>> y += [3]
>>> y.val
[1, 2, 3]
```

The normal `__add__` method is run as a fallback, but may not be able to optimize in-place cases:

```
>>> class Number:
        def __init__(self, val):
            self.val = val
        def __add__(self, other):             # __add_
            return Number(self.val + other)   # Propag

>>> x = Number(5)
>>> x += 1
>>> x += 1                                    # And +=
>>> x.val
7
```

Though we've focused on `+` here, keep in mind that *every* binary operator has similar right-side and in-place overloading methods that work the same (e.g., `__mul__`, `__rmul__`, and `__imul__`). Still, these methods tend to be uncommon in practice; you only code right-side methods for either-side roles and in-place methods for code economy or speed—and only if you need to support such operators at all. For instance, a `Vector` class may use these tools, but a `Button` class probably would not. Button presses, though, may run the next section's method.

Call Expressions: __call__

On to our next overloading method: the `__call__` method is called when your instance is called. No, this isn't a circular definition—if defined, Python runs a `__call__` method for function-call expressions applied to your instances, passing along whatever positional or keyword arguments were sent. This allows instances to conform to a function-based API:

```
>>> class Callee:
        def __call__(self, *pargs, **kargs):        # Ir
            print(f'Called: {pargs=} {kargs=}')     # Ac

>>> C = Callee()
>>> C(1, 2, 3)                                      # C
Called: pargs=(1, 2, 3) kargs={}
>>> C(1, 2, 3, x=4, y=5)
Called: pargs=(1, 2, 3) kargs={'x': 4, 'y': 5}
```

More formally, all the argument-passing modes we explored in Chapter 18 are supported by the `__call__` method—whatever is passed to the instance is passed to this method, along with the usual implied instance argument `self`. For example, the method definitions:

```
class C:
    def __call__(self, a, b, c=5, d=6): ...        # Nc

class C:
    def __call__(self, *pargs, **kargs): ...       # Cc

class C:
    def __call__(self, *pargs, d=6, **kargs): ...  # 3.
```

all match all the following instance calls after assigning `X` to `C()`:

```
X(1, 2)                                            # Or
X(1, 2, 3, 4)                                      # Pc
X(a=1, b=2, d=4)                                   # Ke
```

```
X(*[1, 2], **dict(c=3, d=4))                    # Ur
X(1, *(2,), c=3, **dict(d=4))                   # Mi
```

See Chapter 18 for a refresher on function arguments. The net effect is that classes and instances with a `__call__` support the exact same argument syntax and semantics as normal functions and methods.

Intercepting call expression like this allows class instances to emulate the look and feel of things like functions, but also retain state information for use during calls. The following, for example, defines callable objects with per-call info specified by explicit attribute assignments:

```
>>> class Prod:
        def __init__(self, value):                  # Ac
            self.value = value
        def __call__(self, other):
            return self.value * other

>>> x = Prod(2)                                     # "P
>>> x(3)                                            # 3
6
>>> x(4)
8
```

In this example, the `__call__` may seem a bit gratuitous at first glance. A simple method can provide similar utility:

```
>>> class Prod:
        def __init__(self, value):
            self.value = value
        def comp(self, other):
            return self.value * other

>>> x = Prod(3)
>>> x.comp(3)
9
```

However, `__call__` can become more beneficial when using APIs (i.e., libraries) that expect functions—it allows us to code objects that conform to an expected function-call interface but also retain state information and other class assets such as inheritance. In fact, it may be the third most commonly

used operator-overloading method, behind the `__init__` constructor and the `__str__` and `__repr__` display-format alternatives (qualitatively speaking).

Function Interfaces and Callback-Based Code

As an example, Python's `tkinter` GUI module lets you register functions as event handlers (a.k.a. *callbacks*)—when events occur, `tkinter` calls the registered objects. If you want an event handler to retain state between events, you can register a class *instance* that conforms to the expected interface with `__call__`. Chapter 17's *closure functions* can achieve similar effects but don't provide as much support for multiple operations or customization.

To demo the concept, here's a hypothetical example of `__call__` applied to the GUI domain. The following class defines an object that supports a function-call interface but also has state information that remembers the color a button should change to when it is later pressed:

```
class Callback:
    def __init__(self, color):                # Functior
        self.color = color
    def __call__(self):                       # Support
        print('turn', self.color)
```

In the context of a GUI, we can register instances of this class as event handlers for buttons, even though the GUI expects to be able to invoke event handlers as simple functions with no arguments:

```
cb1 = Callback('blue')                        # Remember
cb2 = Callback('green')                       # Remember

B1 = Button(command=cb1)                      # Register
B2 = Button(command=cb2)
```

When the button is later pressed, the instance object is called as a simple function with no arguments, exactly like in the following calls. Because it retains state as instance attributes, though, it remembers what to do—it becomes a *stateful function* object:

```
cb1()                                   # On event
cb2()                                   # On event
```

In fact, some consider such classes to be the best way to retain state information in the Python language. With OOP, the state remembered is made explicit with attribute assignments. This is different than other state retention techniques (e.g., global variables, enclosing function scope references, and default mutable arguments), which rely on more limited or implicit behavior. Moreover, the added structure and customization in classes goes beyond state retention.

On the other hand, tools such as closure functions are useful in basic state retention roles too, and the `nonlocal` statement makes enclosing scopes a viable alternative in more programs. We'll revisit such trade-offs when we start coding substantial decorators in Chapter 39, but here's a quick *closure* equivalent:

```
def callback(color):                    # Enclosir
    def oncall():
        print('turn', color)
    return oncall

cb3 = callback('yellow')                # Handler
cb3()                                   # On event
```

Before we move on, there are two other ways that Python programmers sometimes tie information to a callback function like this. One option is to use default arguments in `lambda` functions:

```
cb4 = (lambda color='red':              # Defaults
           print('turn', color))        # lambda c
cb4()                                   # On event
```

The other is to use *bound methods* of a class—covered in the next chapter but simple enough to preview here. A bound-method object is created for instance methods referenced but not called and remembers both the instance and the referenced function. This object may therefore be called later as a simple function without an instance:

```
class Callback:
    def __init__(self, color):                # Class wi
        self.color = color
    def changeColor(self):                    # A normal
        print('turn', self.color)

cb1 = Callback('blue')
cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor)          # Bound me
B2 = Button(command=cb2.changeColor)          # Remember
```

In this case, when this button is later pressed it's as if the GUI does the following, which invokes the instance's `changeColor` method to process the `cb1` object's state information instead of calling the instance itself:

```
cb1 = Callback('blue')
cmd = cb1.changeColor                         # Register
cmd()                                         # On event
```

Note that a `lambda` is not required here unless extra arguments must be passed, because a bound method reference by itself already *defers* a call until later. This technique doesn't require overloading calls with `__call__`, but either scheme may be preferred for a given program. Again, watch for more about bound methods in the next chapter.

You'll also see another `__call__` example in Chapter 32, where we will use it to implement a function decorator—a callable object often used to add a layer of logic on top of an embedded function. Because `__call__` allows us to attach state information to a callable object, it's a natural implementation technique for a function that must remember to call another function when called itself. For more `__call__` examples, also watch for the more advanced decorators and metaclasses of Chapters 39 and 40.

Comparisons: __lt__, __gt__, and Others

Our next batch of overloading methods supports object comparisons. As suggested in Table 30-1, classes can define methods to catch all six

comparison operators: `<`, `>`, `<=`, `>=`, `==`, and `!=`. These methods are generally straightforward to use, but keep the following qualifications in mind:

- Unlike the `__add__` / `__radd__` pairings discussed earlier, there are no *right-side* variants of comparison methods. Instead, reflective methods are used when only one operand supports comparison (e.g., `__lt__` and `__gt__` are each other's reflection).
- There are no implicit *relationships* among the comparison operators. The truth of `==` does not imply that `!=` is false, for example, so both `__eq__` and `__ne__` should be defined to ensure that both operators behave correctly.

We don't have space for an in-depth exploration of comparison methods, but as a quick introduction, consider the following class and tests:

```
>>> class Vetter:
        data = 'hack'
        def __gt__(self, other):
            print(f'gt: {self=} {other=}')
            return self.data > other
        def __lt__(self, other):
            print(f'lt: {self=} {other=}')
            return self.data < other

>>> X = Vetter()
>>> X > 'code', X < 'code'
gt: self=<__main__.Vetter object at 0x10898f650> other=
lt: self=<__main__.Vetter object at 0x10898f650> other=
(True, False)

>>> 'code' < X, 'code' > X
gt: self=<__main__.Vetter object at 0x10898f650> other=
lt: self=<__main__.Vetter object at 0x10898f650> other=
(True, False)

>>> X < X
lt: self=<__main__.Vetter object at 0x10898f650> other=
gt: self=<__main__.Vetter object at 0x10898f650> other=
False
```

When run, the class's methods intercept and implement comparison expressions as noted by their trace outputs. Importantly, `__lt__` is also used for *sorts*—both the list method and built-in function:

```
>>> class Order:
        def __init__(self, data):
            self.data = data
        def __lt__(self, other):
            return self.data < other.data
        def __repr__(self):
            return f'Order({self.data})'

>>> sorted(Order(i) for i in [3, 1, 4, 2])
[Order(1), Order(2), Order(3), Order(4)]
```

Consult Python's manuals for more details in this category. As you'll find there, the `__eq__` method run for value equality is coupled with the `__hash__` method run for as-key and set-object roles (in ways that should send most readers screaming into the night); and comparison methods can also return `NotImplemented` for unsupported arguments (but again, not `NotImplementedError`, an exception with a similar name but very different roles).

Boolean Tests: __bool__ and __len__

The next set of methods is truly useful (pun intended). As you've learned, every object is inherently true or false in Python. When you code classes, you can define what this means for your objects by coding methods that give the `True` or `False` values of instances on request.

In Boolean contexts, Python first tries `__bool__` to obtain a direct Boolean value; if that method is missing, Python tries `__len__` to infer a truth value from the object's length—a length of zero means empty, which is always false. The first of these generally uses object state or other information to produce a Boolean result:

```
>>> class Truth:
        def __bool__(self): return True

>>> X = Truth()
```

```
>>> if X: print('yes!')

yes!

>>> class Truth:
        def __bool__(self): return False

>>> X = Truth()
>>> bool(X)
False
```

If this method is missing, Python falls back on length because a nonempty object is considered true. That is, a nonzero length is taken to mean the object is true, and a zero length means it is false—just as for built-in objects:

```
>>> class Truth:
        def __len__(self): return 0             # Empty n

>>> X = Truth()
>>> if not X: print('no!')

no!
```

If *both* methods are present Python prefers `__bool__` over `__len__`, because it is more specific:

```
>>> class Truth:
        def __bool__(self): return True         # Preferr
        def __len__(self): return 0             # Object

>>> if Truth(): print('yes!')

yes!
```

If *neither* truth method is defined, the object is vacuously considered true (though any existential implications of this are strictly out of scope here):

```
>>> class Truth:
        pass

>>> X = Truth()
```

```
>>> bool(X)
True
```

But now that we've managed to cross over into the realm of philosophy, let's move on to look at one last overloading context: *object demise*.

Object Destruction: __del__

It's time to close out this chapter—and learn how to do the same for our class objects. You've seen how the `__init__` *constructor* is called whenever an instance is generated (and noted how `__new__` is run first to make the object). Its counterpart, the *destructor* (less commonly known as *finalizer*) method `__del__`, is run automatically when an instance's space is being reclaimed (i.e., at garbage-collection time):

```
>>> class Life:
        def __init__(self, name):
            print('Hello', name)
            self.name = name
        def live(self):
            print(self.name + '...')
        def __del__(self):
            print('Goodbye', self.name)

>>> pat = Life('Pat')
Hello Pat
>>> pat.live()
Pat...
>>> pat = 'end'
Goodbye Pat
```

Here, when `pat` is assigned a string at the end, we lose the last reference to the `Life` instance and so trigger its destructor method. This works, and it may be useful for implementing some cleanup activities, such as terminating a server connection. However, destructors are not as commonly used in Python as in some OOP languages, for a number of reasons that the next section describes.

Destructor Usage Notes

The destructor method works as documented, but it has some well-known caveats and a few outright dark corners that make it somewhat rare to see in Python code:

Need

For one thing, destructors may not be as useful in Python as they are in some other OOP languages. Because Python automatically reclaims all *memory space* held by an instance when the instance is reclaimed, destructors are not necessary for space management. In the current CPython implementation of Python, you also don't need to close *file objects* held by the instance in destructors because they are automatically closed when reclaimed. As mentioned in Chapter 9, though, it's still sometimes best to run file close methods anyhow because this autoclose behavior may vary in alternative Python implementations.

Predictability

For another, you cannot always easily predict when an instance will be reclaimed. In some cases, there may be lingering references to your objects in system tables that prevent destructors from running when your program expects them to be triggered. Python also does not guarantee that destructor methods will be called for objects that still exist when the interpreter exits.

Exceptions

In fact, `__del__` can be tricky to use for even more subtle reasons. Exceptions raised within it, for example, simply print a warning message to `sys.stderr` (the standard error stream) rather than triggering an exception event, because of the unpredictable context under which it is run by the garbage collector—it's not always possible to know where such an exception should be delivered.

Cycles

In addition, cyclic (a.k.a. circular) references among objects may prevent garbage collection from happening when you expect it to. An optional cycle detector, enabled by default, can automatically collect

such objects eventually (including objects with `__del__` methods, as of Python 3.4). Since this is relatively obscure, we'll ignore further details here; see Python's standard manuals' coverage of both `__del__` and the `gc` garbage collector module for more information.

Because of these downsides, it's often better to code termination activities in an explicitly called method (e.g., `shutdown`). As described in the next part of the book, the `try`/`finally` statement also supports termination actions, as does the `with` statement for objects that support its context-manager model.

Chapter Summary

That's as many overloading examples as we have space for here. Most of the other operator-overloading methods work similarly to the ones we've explored, and all are just hooks for intercepting built-in type operations. Some overloading methods, for example, have unique argument lists or return values, but the general usage pattern is the same. You'll see a few others in action later in the book:

- Chapter 34 uses `__enter__` and `__exit__` in `with` statement context managers.
- Chapter 38 uses the `__get__` and `__set__` class descriptor fetch/set methods.
- Chapter 40 uses the `__new__` object creation method in the context of metaclasses.

In addition, some of the methods we've studied here, such as `__call__` and `__str__`, will be employed by later examples in this book. For complete coverage, though, it must defer to other documentation sources—see Python's language manual or other reference resources for details on additional overloading methods.

In the next chapter, we leave the realm of class mechanics behind to explore *design*—the ways that classes are commonly used and combined to optimize code reuse. After that, we'll survey a gumbo of advanced class topics and move on to exceptions, the last core subject of this book. Before you read on,

though, take a moment to work through the chapter quiz below to review the concepts we've covered here.

Test Your Knowledge: Quiz

1. What two operator-overloading methods can you use to support iteration in your classes?
2. What two operator-overloading methods handle printing, and in what contexts?
3. How can you intercept slice operations in a class?
4. How can you catch in-place addition in a class?
5. When should you provide operator overloading?

Test Your Knowledge: Answers

1. Classes can support iteration by defining (or inheriting) `__getitem__` or `__iter__`. In all iteration tools, Python tries to use `__iter__` first, which returns an object that supports the iteration protocol with a `__next__` method: if no `__iter__` is found by inheritance search, Python falls back on the `__getitem__` indexing method, which is called repeatedly, with successively higher indexes. If used, the `yield` statement can create the `__next__` method automatically.
2. The `__str__` and `__repr__` methods implement object print displays. The former is called by the `print` and `str` built-in functions; the latter is called by `print` and `str` if there is no `__str__`, and always by the `repr` built-in, interactive echoes, and nested appearances. That is, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. A `__str__` is usually used for user-friendly displays; `__repr__` gives extra details or the object's as-code form. String formatting runs these methods too.
3. Slicing is caught by the `__getitem__` indexing method: it is called with a `slices` object instead of a simple integer index, and slice objects may be passed on or inspected as needed.
4. In-place addition tries `__iadd__` first, and `__add__` with an assignment second. The same policy holds true for all binary operators. The `__radd__` method is also available for right-side addition.

5. When a class naturally matches, or needs to emulate, a built-in type's interfaces. For example, collections might imitate sequence or mapping interfaces, and callables might be coded for use with an API that expects a function. You generally shouldn't implement expression operators if they don't naturally map to your objects naturally and logically, though—use normally named methods instead.