Chapter 11. Assignments, Expressions, and Prints

Now that we've had a first introduction to Python statement syntax, this chapter begins our in-depth tour of specific Python statements. We'll begin with the basics: assignment statements, expression statements, and print operations. We've already seen all of these in action, but here we'll fill in important details we've skipped so far. Although they're relatively simple, as you'll see, there are optional variations for each of these statement types that will come in handy once you begin writing realistic Python programs.

Assignments

We've been using the Python assignment statement for a while to retain objects in examples. In its basic form, you write the *target* of an assignment on the left of an equals sign, and the *object* to be assigned on the right. The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that creates an object. For the most part, assignments are straightforward, but here are a few key properties to note up front:

- **Assignments create object references.** As discussed in Chapter 6, Python assignments store references to objects in names or data structure components. They always create *references* to objects instead of copying the objects. Because of that, Python variables are more like pointers than data storage areas.
- **Names are created when first assigned.** Python creates a variable name the first time you *assign* it a value (i.e., an object reference), so there's no need to predeclare names ahead of time. Some (but not all) data structure slots are created when assigned, too (e.g., dictionary entries, some object attributes). Once assigned, a name is replaced with the value it references whenever it appears in an expression.
- **Names must be assigned before being referenced.** It's an error to use a name to which you haven't yet assigned a value. Python raises an

exception if you try, rather than returning some sort of ambiguous *default* value. This turns out to be crucial in Python because names are not predeclared—if Python provided default values for unassigned names used in your program instead of treating them as errors, it would be much more difficult for you to spot name typos in your code.

- **Some operations perform assignments implicitly.** In this section, we're concerned with the `=` statement and its `:=` expression relative, but assignment occurs in many contexts in Python. For instance, you'll see later that module imports, function and class definitions, `for` loop variables, and function arguments are all implicit assignments. Because assignment works the same everywhere it pops up, all these contexts simply *bind* (i.e., assign) names and object components to object references at runtime.

With those preliminaries in hand, let's move on to the code.

Assignment Syntax Forms

Although assignment is a general and pervasive concept in Python, in this chapter we are primarily interested in assignment *statements*, plus one limited *expression*. Table 11-1 illustrates the different syntax forms available for coding assignments in Python.

Table 11-1. Assignment statement and expression forms

Operation	Interpretation
`target = 'Hack'`	Basic assignment
`code, hack = 'py', 'PY'`	Tuple assignment
`[code, hack] = ['py', 'PY']`	List assignment
`a, b, c, d = 'hack'`	Sequence assignment
`a, *b = 'hack'`	Extended-unpacking assignment
`code = hack = 'python'`	Multiple-target assignment
`code += 1, hack *= 2`	Augmented assignments
`(python := 3.12) + 0.01`	Named assignment expression

The *basic* form atop Table 11-1 is by far the most common: binding a single target (a name or data structure component) to a single object (an expression result). In fact, you could get all your work done with this form alone. The other table entries represent forms that are all optional, but that programmers often find convenient in practice:

Tuple and list assignments

The second and third forms in the table are related. When you code a tuple or list on the left side of the `=`, Python *pairs* objects on the right side with targets on the left by *position* and assigns them from left to right. For example, in the second line of Table 11-1, both sides are tuples (sans parentheses), and the names `code` and `hack` are assigned `'py'` and `'PY'`, respectively. The left of the `=` is special syntax but the right is a real object, which is why this is called "unpacking" assignment—components on the right are unpacked into targets on the left.

Sequence assignment

Tuple and list assignments were later generalized into instances of what we now call *sequence assignment*—any sequence of targets can be assigned to any sequence (really, iterable) of values, and Python assigns the items one at a time by position. We can even mix and match the types of the sequences involved. The fourth line in Table 11-1, for example, pairs a tuple of names with a string of characters: `a` is assigned `'h'`, `b` is assigned `'a'`, and so on. Despite this flexibility, the item on the left of the `=` is still a tuple or list of assignment targets.

Extended-unpacking assignment

An even later assignment form allows more flexibility in how we assign portions of a sequence—or other iterable—to a sequence of targets. The fifth line in Table 11-1, for example, matches `a` with the first character in the string on the right, and the *starred* name `b` with the rest: `a` is assigned `'h'`, and `b` is assigned `['a', 'c', 'k']`. This provides an alternative to assigning the results of slicing operations. Starred collectors like this have also somewhat usurped the term *unpacking*, though this is an artifact more historical than technical.

Multiple-target assignment

The sixth line in Table 11-1 shows the multiple-target form of assignment. In this form, Python assigns a reference to the same object (the object farthest to the right) to all the targets on the left. In the table, the names `code` and `hack` are both assigned references to the same string object, `'python'`. The effect is the same as if we had run `hack = 'python'` followed by `code = hack`, as `hack` evaluates to the original string object (i.e., not a separate copy of that object).

Augmented assignments

The second-to-last line in Table 11-1 is an example of *augmented assignment*—a shorthand that combines an expression and an assignment in a concise way. Saying `code += 1`, for example, has the same effect as `code = code + 1`, but the augmented form requires less typing and is generally quicker to run. In addition, if the target of the assignment is *mutable*, an augmented assignment may run even quicker by choosing an *in-place* update operation instead of an object copy. As you'll see, there is one augmented assignment statement for most binary expression operators in Python (even the unused `@`!).

Named assignment expression

New in Python 3.8, the `:=` operator allows you to code assignment as an *expression*, which returns the value it assigns to a name. This expression can be nested in places where assignment statements don't work syntactically, and in common roles allows you to both assign a name and use its value in the same place in your code.

Basic Assignments

Let's turn to examples at the REPL prompt as usual. We've already used Table 11-1's *basic* assignment in this book, so you should be familiar with its basics. In short, it assigns a single *target* to a single value, where the target may be a name, index, slice, or attribute, and the value is any expression:

```
$ python3
>>> L = [1, 2]                  # Name target
>>> L[0] = 3                    # Index target
>>> L[-1:] = [4, 5]             # Slice target
>>> L
[3, 4, 5]
```

Attribute targets crop up for classes; we haven't studied these yet, but the assignment is straightforward:

```
object.attr = L            # Attribute target (see Part
```

Though *names* are common and dominate examples here, any of these four types of assignment targets can be used in any form of assignment, except where noted ahead (named assignment expressions, for example, allow only names).

NOTE

Name annotations: Basic assignment statements can also have an *annotation* expression introduced by a colon immediately after the target on the left. This statement form allows only a single target, and is used for *type hinting*, described near the end of Chapter 6. As noted there, because this is optional, convoluted, completely unused by Python, and fundamentally at odds with the language's core idiom, we're skipping it in this book. See Python's docs for more details if you ever stumble in the wild onto the curious case of type declarations in a dynamically typed language.

Sequence Assignments

Next up, here are a few simple and comparable examples of tuple and list assignment (a.k.a. *sequence* assignment) in action, unpacking items into individual variables:

```
>>> first  = 1                     # Basic assignment
>>> second = 2

>>> A, B = first, second           # Tuple assignment
>>> A, B                           # Similar to A = fi
(1, 2)

>>> [C, D] = [first, second]       # List assignment
>>> C, D
(1, 2)
```

Notice that we really are coding two tuples in the third line in this interaction —we've just omitted their enclosing parentheses. Python pairs the *values* in the tuple on the right side of the assignment operator with the *variables* in the tuple on the left side and assigns the values one at a time. The same goes when `=` is surrounded by lists.

Tuple assignment leads to a common coding trick in Python that was introduced in a solution to the exercises at the end of Part II. Because Python creates a temporary tuple that saves the original values of the variables on the right while the statement runs, unpacking assignments are also an easy way to *swap* two variables' values without creating a temporary variable of your own—the tuple on the right remembers the prior values of the variables automatically:

```
>>> first  = 1
>>> second = 2
>>> first, second = second, first    # Tuples: swaps vc
>>> first, second                    # Like T = first;
(2, 1)
```

As already noted, the original tuple and list assignment forms in Python were eventually generalized to accept *any* type of sequence (really, *iterable*) on the right as long as it is of the same length as the sequence on the left. You can assign a tuple of values to a list of variables, a string of characters to a tuple of variables, and so on. In all cases, Python assigns items in the sequence on the right to targets in the sequence on the left by position—from left to right:

```
>>> [a, b, c] = (1, 2, 3)           # Assign tuple of vc
>>> a, c
(1, 3)
>>> (a, b, c) = 'ABC'               # Assign string of c
>>> a, c
('A', 'C')
```

More broadly, while the left side of a sequence assignment is still a sequence (a tuple or list of targets), the right side may be any *iterable* object, not just any sequence. This is a more general category that includes collections both physical (e.g., lists) and virtual (e.g., a file's lines), which was first defined in Chapter 4 and has popped up in passing ever since. We'll firm up this term when we explore iterables in Chapters 14 and 20, and apply it to unpack a `range` iterable in the next section. For now, the "sequence" in assignment is best associated with what's on the left of the `=`.

Advanced sequence-assignment patterns

Although we can mix and match sequence types around the `=` symbol, we must generally have the *same number* of items on the right as we have targets on the left, or we'll get an error. As you'll see in the next section, Python allows us to be more general with extended-unpacking `*` syntax, but the number of items in the assignment target and subject must normally match:

```
>>> string = 'TEXT'
>>> a, b, c, d = string                                    # Sc
>>> a, b, c, d
('T', 'E', 'X', 'T')

>>> a, b, c = string                                       # Er
ValueError: too many values to unpack (expected 3)
```

To be more flexible, we can always *slice*. There are a variety of ways to employ slicing to make this last case work:

```
>>> a, b, c = string[0], string[1], string[2:]             # Ir
>>> a, b, c                                                # a,
('T', 'E', 'XT')

>>> a, b, c = list(string[:2]) + [string[2:]]              # Sl
>>> a, b, c                                                # a,
('T', 'E', 'XT')

>>> a, b = string[:2]                                      # Sl
>>> c = string[2:]                                         # a,
>>> a, b, c
('T', 'E', 'XT')

>>> (a, b), c = string[:2], string[2:]                     # Ne
>>> a, b, c                                                # (c
('T', 'E', 'XT')
```

As the last example in this interaction demonstrates, we can even assign *nested* sequences, and Python unpacks their parts according to their shape, as expected. In this case, we are assigning a tuple of two items, where the first item is a nested sequence (a string), exactly as though we had coded it this way:

```
>>> ((a, b), c) = ('TE', 'XT')                         # Pc
>>> a, b, c
('T', 'E', 'XT')
```

Python pairs the first string on the right (`'TE'`) with the first tuple on the left (`(a, b)`) and assigns one character at a time, before assigning the entire second string (`'XT'`) to the variable `c` all at once. In this event, the sequence-nesting shape of the object on the left must match that of the object on the right. Nested sequence assignment like this is somewhat rare to see, but it can be convenient for picking out the parts of data structures with known shapes.

For example, you'll see in Chapter 13 that this technique also works in `for` loops, because loop items are assigned to the target given in the loop header just as if an `=` statement had been run:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: …             # S

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: …       # N
```

Such nested sequence assignments can also be achieved with the `match` statement of the next chapter, but we'll hold back the details until then. Sequence-unpacking assignments also give rise to another common coding idiom in Python—assigning an integer series to a set of variables:

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

This code initializes the three names on the left of `=` to the integers `0` , `1` , and `2` , respectively. Because each name gets a unique value, it's Python's simplest equivalent to the *enumerated* data types you may have seen in other languages. To make sense of this, you need to recall that the `range` built-in function returns an *iterable*, which generates a list of successive integers (wrap it in a `list` if you wish to display its values at the REPL all at once like this):

```
>>> list(range(3))                            # list() requi
[0, 1, 2]
```

This call was previewed briefly in Chapter 4; because `range` is commonly used in `for` loops, we'll say more about it in Chapter 13. Another place you may see a tuple assignment at work is for splitting a sequence into its front and the rest, in loops like the following's `while`, introduced in the prior chapter:

```
>>> L = [1, 2, 3, 4]
>>> while L:                                  # Reapeat unti
...     front, L = L[0], L[1:]                # See next sec
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

The tuple assignment in the loop here could be coded as the following two lines instead, but it's often more convenient to string them together:

```
...     front = L[0]
...     L = L[1:]
```

Notice that this code is using the list as a sort of *stack* data structure, which can often also be achieved with the `append` and `pop` methods of list objects; here, `front = L.pop(0)` would have much the same effect as the tuple assignment statement, but it would be an in-place change. You'll learn more about `while` loops, and other (and often better) ways to step through a sequence with `for` loops, in Chapter 13.

Extended-Unpacking Assignments

The prior section demonstrated how to use manual slicing to make sequence assignments more general. In later Pythons, sequence assignment was further extended to make this easier. In short, a *starred target*, `*X`, can be used on the left of `=` in order to specify a more general matching against the iterable on the right—the starred target is assigned a list, which collects all items in the iterable not assigned to other targets. This is especially handy for common coding patterns such as splitting a sequence into its “front” and “rest,” as in the preceding example.

Extended unpacking in action

Let’s jump right into an example to demo how this works. As already shown, sequence assignments normally require exactly as many targets on the left as there are items in the object on the right. We get an error if the lengths disagree, unless we manually slice on the right, as shown in the prior section:

```
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> a, d
(1, 4)

>>> a, b = seq
ValueError: too many values to unpack (expected 2)
```

We can, however, use a single starred target in this example to match more generally. In the following continuation of our interactive session, `a` matches the first item in the sequence, and `b` matches the rest:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

When a starred target is used (like name `b` here), the number of items on the left need not match the length of the iterable on the right (like sequence `seq` here). In fact, the starred target can appear *anywhere* on the left. For instance, in the next interaction `b` matches the last item in the sequence, and `a` matches everything before the last:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
```

```
>>> b
4
```

When the starred target appears in the *middle*, it collects everything between the other targets listed. Thus, in the following interaction, `a` and `c` are assigned the first and last items, and `b` gets everything in between them:

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

More generally, wherever the starred target shows up, it will be assigned a list that collects every unassigned item at that position:

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Naturally, like normal sequence assignment, extended-unpacking syntax works for any sequence types (really, again, any *iterable*), not just lists. Here it is unpacking characters in a string and an iterable `range`—which generates values 0…3 on demand:

```
>>> a, *b = 'hack'
>>> a, b
('h', ['a', 'c', 'k'])

>>> a, *b, c = 'hack'
>>> a, b, c
(('h', ['a', 'c'], 'k')

>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

This is similar in spirit to *slicing*, but not exactly the same—a sequence unpacking assignment always returns a *list* for matched items, whereas slicing returns a sequence of the same type as the object sliced:

```
>>> S = 'hack'

>>> a, b, c = S[0], S[1:-1], S[-1]       # Slices are ty
>>> a, b, c
('h', 'ac', 'k')

>>> a, *b, c = S                         # But * always
>>> a, b, c
('h', ['a', 'c'], 'k')
```

Finally, the closing example of the prior section becomes even simpler with this extension, since we don't have to manually slice to get the first and rest of the items (though the "rest" `L` always becomes a list after the first `*`):

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, *L = L                    # Get first, re
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Boundary cases

Although extended unpacking is flexible, some boundary (atypical) cases are worth noting. First, the starred target may match just a single item, but is always assigned a list:

```
>>> seq = [1, 2, 3, 4]

>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]
```

Second, if there is nothing left to match the starred target, it is assigned an empty list, regardless of where it appears. In the following, names `a`, `b`, `c`, and `d` have matched every item in the sequence, but Python assigns `e` an empty list instead of treating this as an error case:

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

Errors can still be triggered, though, if there is more than one starred target, if there are too few values and no star (as before), and if the starred target is not itself coded inside a sequence:

```
>>> a, *b, c, *d = seq
SyntaxError: multiple starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack (expected 2)

>>> *a = seq
SyntaxError: starred assignment target must be in a lis

>>> *a, = seq          # A one-item tuple sans parenthes
>>> a                  # Same as a = seq, but makes a ne
[1, 2, 3, 4]
```

Technically, the single-appearance rule for starred targets on the left of `=` really applies only to *each* sequence when there is *nesting* on both sides—though you still can't have more than one per nested sequence:

```
>>> [(a, *b), (c, *d)] = [(1, 2, 3, 4), (5, 6, 7, 8)]
>>> a, b, c, d
(1, [2, 3, 4], 5, [6, 7, 8])

>>> [(a, *b, *x), (c, *d)] = [(1, 2, 3, 4), (5, 6, 7, 8
SyntaxError: multiple starred expressions in assignment
```

And finally, any assignment *target* can be starred—though you may be hard-pressed to find some in real code:

```
>>> a, *L = [1, 2, 3]           # Name target
>>> a, L
(1, [2, 3])
>>> a, *L[0] = [4, 5, 6]        # Index target
>>> a, L
(4, [[5, 6], 3])
>>> a, *L[1:] = [7, 8, 9]       # Slice target
>>> a, L
(7, [[5, 6], 8, 9])

>>> class C: …code…             # See Part VI
>>> a, *C.attr = 'yikes'        # Attribute target
>>> a, C.attr
('y', ['i', 'k', 'e', 's'])
```

A useful convenience

Keep in mind that extended-unpacking assignment is just a convenience (well, in addition to a mouthful). We can usually achieve the same effects with explicit indexing and slicing, but extended unpacking is simpler to code. The common "first, rest" splitting coding pattern, for example, can be coded either way, but slicing is extra work:

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq                         # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:]              # First, rest sc
>>> a, b
(1, [2, 3, 4])
```

The also-common "rest, last" splitting pattern can similarly be coded either way, but extended-unpacking syntax again requires noticeably fewer keystrokes (and notably fewer neurons):

```
>>> *a, b = seq                     # Rest, last
>>> a, b
([1, 2, 3], 4)

>>> a, b = seq[:-1], seq[-1]        # Rest, last sar
>>> a, b
([1, 2, 3], 4)
```

Being both simpler and arguably more natural, extended-unpacking syntax is also common in Python code.

Application to for loops

Because the loop variable in the `for` loop statement can be any assignment target, extended unpacking works here too. We used the `for` loop iteration tool briefly in Chapter 4 and will study it formally in Chapter 13, but its tie-in here is straightforward: extended-unpacking assignments may show up after the word `for`, where a simple variable name is more commonly used:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]: …
```

When used in this context, on each iteration Python simply assigns the next tuple of values to the tuple of targets. On the first loop, for example, it's as if we'd run the following assignment statement:

```
a, *b, c = (1, 2, 3, 4)                            # b
```

The names `a`, `b`, and `c` can be used within the loop's code to reference the extracted components. In fact, this is really not a special case at all, but just an instance of general assignment at work. As we saw earlier in this chapter, for example, we can do similar in loops with simple sequence (tuple) assignment:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: …          # c
```

And we can always emulate extended-unpacking assignment behavior by manually slicing:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]: …
    a, b, c = all[0], all[1:-1], all[-1]
```

Since we haven’t learned enough to get more detailed about the syntax of `for` loops, we’ll put this topic on the back burner until its reprise in Chapter 13.

THE MANY STARS OF PYTHON

No, this sidebar is not about personalities (which already permeate software culture more than they should). It's a reference to all the special-case appearances of `*` that have cropped up in Python over the years, many of which you've already seen. Prior to Python 3.5, the special starred-item syntax forms can appear in:

- *Assignments*, where a single `*X` starred assignment target of any kind (name, index, slice, attribute), and repeatable in nested parts, collects unmatched items in a new list, as covered in this chapter
- *Function headers*, where single `*X` and `**X` starred names collect unmatched positional and keyword arguments in a tuple and dictionary, respectively
- *Function calls*, where single `*X` and `**X` starred expressions unpack iterables and mappings into individual positional and keyword arguments, respectively

As of Python 3.5, *function calls* (the latter listed item) support any number of `*X` and `**X` unpacking expressions, and any number of `*X` and `**X` starred expressions can also appear in *object literals*, where they unpack collections of types implied by the literal in which they appear:

```
[x, *iter]         # List:  unpack items in iterables
(x, *iter, y)      # Tuple: ditto, parentheses or not
{*iter, x}         # Set:   ditto, though values are unor
{x: y, **map}      # Dict:  unpack keys/values in mapping
```

As of Python 3.10, both `*X` and `**X` starred names can also show up in the patterns of the `match` statement covered in the next chapter, where they serve roles similar to that in sequence assignment, though only `*X` overlaps between the two, and its assignment in `match` is really a side effect of a true-or-false test.

And as of Python 3.11, an `except*` can appear in `try` statements, where it allows multiple handlers to be run to process multiple exceptions wrapped in an exception *group*—an addition that bifurcates the already-convoluted `try` for narrow roles, which we won't meet until Chapter 35.

All that being said, the avenue of stars has been a meandering walk in Python, and it's not impossible that other parts of the language will rise to stardom in the future. Consult the stars for more info.

Multiple-Target Assignments

Simpler than sequence assignment on first glance, a *multiple-target assignment* simply assigns all the given targets to the same object all the way to the right. The following, for example, assigns the three names (variables) `a`, `b`, and `c` to the string `'code'`:

```
>>> a = b = c = 'code'
>>> a, b, c
('code', 'code', 'code')
```

This form is equivalent to—but easier to code and lighter on line count than—these three assignments:

```
>>> c = 'code'
>>> b = c
>>> a = b
```

Multiple-target assignment and shared references

While this seems simple, keep in mind that there is just one object here, shared by all three variables: they all wind up referencing the *same* object in memory. This behavior is worry-free for *immutable* types—for example, when initializing a set of counters to zero (recall that variables must be assigned before they can be used in Python, so you must initialize counters to zero before you can start adding to them):

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

Here, changing `b` changes only `b` because numbers do not support in-place changes. As long as the object assigned is immutable, it's irrelevant if more than one name references it.

As usual, though, we have to be more cautious when initializing variables to an empty *mutable* object such as a list or dictionary:

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

This time, because `a` and `b` reference the *same* object, appending to it in place through `b` will impact what we see through `a` as well. This is really just another example of the shared reference phenomenon we first met in Chapter 6. To avoid the issue, initialize mutable objects in separate statements instead, so that each creates a distinct empty object, by running a distinct literal expression:

```
>>> a = []
>>> b = []                # a and b do not share the s
>>> b.append(42)
>>> a, b
([], [42])
```

A *tuple* assignment like the following has the same effect with more brevity—by running two list-literal expressions, it creates two distinct objects:

```
>>> a, b = [], []         # a and b do not share the s
```

Augmented Assignments

In addition to the basic, sequence (original and starred), and multiple assignment forms already covered, Python gained all the assignment statement formats listed in Table 11-2 relatively early in its career. Known as *augmented assignments*, and borrowed from the C language, these formats are mostly just shorthand: they imply the combination of a binary (two-operand) expression and an assignment. For instance, the following two formats are functionally equivalent, though the latter may use in-place options to change `X` directly, as you'll see in a moment:

```
X = X + Y                       # Basic form
X += Y                          # Augmented form
```

Table 11-2. Augmented assignment statements

`X += Y`	`X -= Y`	`X *= Y`	`X /= Y`	`X @= Y`
`X //= Y`	`X %= Y`	`X **= Y`	`X >>= Y`	
`X <<= Y`	`X &= Y`	`X |= Y`	`X ^= Y`	

Augmented assignment works on any type that supports the implied binary expression. For example, here are two ways to add 1 to a name; really, they both change a name to reference different values:

```
>>> x = 1
>>> x = x + 1                   # Basic
>>> x
2
>>> x += 1                      # Augmented
>>> x
3
```

When applied to a *sequence* such as a string, the augmented form performs concatenation instead, because that's what `+` means for such objects. Thus, the second line here is equivalent to typing the longer `S = S + 'HACK'`:

```
>>> S = 'hack'
>>> S += 'HACK'                 # Implied concatenatior
>>> S
'hackHACK'
```

As shown in Table 11-2, there are analogous augmented assignment forms for other Python binary expression operators (i.e., operators with values on their left and right sides). For instance, `X *= Y` multiplies and assigns, `X >>= Y` shifts right and assigns, and so on (though per Chapter 5, the form `X @= Y` is unused and unimplemented by Python itself today). All told, augmented assignments have three noteworthy advantages:[1]

- There's less for you to type. (Need this book say more?)

- The left side has to be evaluated only once. In `X += Y`, `X` may be a complicated object expression. In the augmented form, its code must be run only once. However, in the long form, `X = X + Y`, `X` appears twice and must be run twice. Because of this, augmented assignments usually run faster.
- The optimal technique is automatically chosen. That is, for objects that support *in-place* changes, the augmented forms automatically perform in-place change operations instead of slower copies.

The last point here requires a bit more explanation. For augmented assignments, in-place operations may be applied for mutable objects as an optimization. Recall that lists can be extended in a variety of ways. To add a single item to the end of a list, we can concatenate or call `append`:

```
>>> L = [1, 2]
>>> L = L + [3]                     # Concatenate one: slow
>>> L
[1, 2, 3]
>>> L.append(4)                     # Faster, but in place
>>> L
[1, 2, 3, 4]
```

And to add a set of items to the end, we can either concatenate again or call the list `extend` method:

```
>>> L = L + [5, 6]                  # Concatenate many: slc
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])                # Faster, but in place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

In both cases, concatenation is less prone to the side effects of shared object references but will generally run slower than the in-place equivalent. Concatenation operations must create a new object, copy in the list on the left, and then copy in the list on the right. By contrast, in-place method calls simply add items at the end of a memory block (it can be a bit more complicated than that internally, but this description suffices).

When we use augmented assignment to extend a list, we can largely forget these details—Python automatically calls the quicker `extend` method (or its equivalent) instead of using the slower concatenation operation implied by `+`:

```
>>> L += [9, 10]                # Mapped to L.extend([9
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

As suggested in Chapter 6, we can also use slice assignment (e.g., `L[len(L):] = [11,12,13]`), but this works roughly the same as the simpler and more mnemonic list `extend` method or the `+=` statement. Note, however, that because of this equivalence `+=` for a list is not exactly the same as a `+` and `=` in all cases—for lists `+=` allows arbitrary sequences (just like `extend`), but concatenation normally does not:

```
>>> L = []
>>> L += 'hack'                 # += and extend allow a
>>> L
['h', 'a', 'c', 'k']
>>> L = L + 'code'
TypeError: can only concatenate list (not "str") to lis
```

Moreover, using *index* and *slice* targets in augmented assignment may change mutables in other ways:

```
>>> L = [1, 2]
>>> L[0] += 10                 # Change an item of a li
>>> L
[11, 2]

>>> L[-1:] += [3, 4]           # Change a section of a
>>> L
[11, 2, 3, 4]
```

Augmented assignment and shared references

More subtly, the *in-place* change implicit in `+=` for lists is very different from the *new* object created by `+` concatenation. As for all shared-reference cases, this difference may matter when objects are shared by many names:

```
>>> L = [1, 2]
>>> M = L                          # L and M reference the
>>> L = L + [3, 4]                 # Concatenation makes a
>>> L, M                           # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]                    # But += really means e
>>> L, M                           # M sees the in-place c
([1, 2, 3, 4], [1, 2, 3, 4])
```

This only matters for mutables like lists and dictionaries, and it is a fairly obscure case (at least, until it impacts your code!). As always, make copies of your mutable objects if you need to break the shared reference structure.

Named Assignment Expressions

For most of Python's three-decade tenure, it resisted emulating the assignment-as-expression idiom of the C language, on the grounds that it was too subtle and error-prone, and fostered code that was hard to read. While these concerns still apply, Python 3.8 gained a flavor of this, known as *named assignment*.

Importantly, this flavor does *not* make normal `=` assignment statements nestable expressions, as they are in C. Instead, it adds a new expression operator to Python, `:=`, on the grounds that its different and limited syntax may neutralize pitfalls of other languages. You still can't accidentally type `=` when you mean `==`, and `:=` is visually distinct.

We explored this expression briefly in a spoiler note in the previous chapter. In short, the expression `name := value` first evaluates expression `value`, and then both:

- *Assigns* the result to the provided variable `name`
- *Returns* the result as the value of the overall `:=` expression

There's no reason to use this syntax for the first part alone, because all of Python's `=` assignment statements already assign values to names (in fact, you'll get a syntax error if you try using `:=` as a statement sans parentheses).

Because `:=` is an *expression* that returns the value assigned, though, it can be nested in contexts where statements aren't allowed and may be used in roles that both test and then use the assigned name.

As an artificial first example, the following nested `:=` both assigns `2` to `b`, and returns it to be used in the `*` string-repetition result assigned to `a`:

```
>>> a = 'hack' * (b := 2)
>>> a
'hackhack'
>>> b
2
```

Among its deliberate limitations, named assignment allows only a simple, single *name* (variable) on the left, so neither other assignment-statement forms nor references to data-structure components work here:

```
>>> (python := 3.12) + 0.01         # Just a name on tł
3.13
>>> python
3.12

>>> (python, Python := 3.12, 3.13) + 0.01
TypeError: can only concatenate tuple (not "float") to
>>> (python[0] := 3.12) + 0.01
SyntaxError: cannot use assignment expressions with suŁ
>>> (python.attr := 3.12) + 0.01
SyntaxError: cannot use assignment expressions with att
```

The first of these failers, for example, is taken to be a three-item tuple with `:=` in the middle, not a two-target named sequence assignment. Moreover, named assignment does not support any of the *augmented* assignment forms we met earlier: there is no `:+=`, for instance, and coding statement `X += 1` may actually take less work than expression `(X := X + 1)` (subtly, a `:=+` runs, but simply applies the `+` identity operator to the expression on the right).

When to use named assignment

While more useful contexts for `:=` are limited, its most common use cases arise in concert with the `if` and `while` statements, which were both introduced in the preceding chapter and earlier. Without `:=`, it was—and still is—common to assign a variable before the statement, test it in the statement header, and then use it in the statement body.

For example, code that reads lines from files and must detect the end of the file (which, you'll recall from Chapter 9, means an empty string that is logically false) may look like these partial snippets (to run code in this section live, open a text file for input and assign it to `file` before each snippet):

```
line = file.readline()                # Sans the := expres
if line:
    print(line)

line = file.readline()                # Ditto, in while lo
while line:
    print(line)
    line = file.readline()
```

To avoid redundant calls, it's also common to code the latter like this in Python—as in the preceding chapter:

```
while True:                           # Sans both := and r
   line = file.readline()
   if not line: break
   print(line)
```

These forms still work well, and the `:=` is never required. With `:=`, however, we can sometimes collapse a fetch, assignment, and test into a one-liner within statement headers themselves:

```
if line := file.readline():           # The := alternative
    print(line)

while line := file.readline():        # Read all lines (vs
    print(line)
```

In both cases, because `:=` *returns* the results of the `readline` call, its logical value can be tested in the statement header itself. And because `:=` also *assigns* the result to `line`, it can be used in the statement's body if it is run. The upshot is that `:=` provides a sort of *shortcut* that allows multiple operations to be coded in a compact way.

If you opt to use this expression, bear in mind that it often requires enclosing *parentheses* to avoid interacting with surrounding code. Even comparing its result to an explicit value, for instance, mandates parentheses that are not required in the equivalent statements:

```
if (line := file.readline()) != ignore:           # Pare
    print(line)

while (line := file.readline()) != stop:          # And
    print(line)
```

As a rule of thumb, if you opt to use `:=`, wrapping it in parentheses both sets it off visually and avoids sticky issues that may arise if the code surrounding it changes its meaning unexpectedly. In the preceding, for example, names are assigned the results of the `!=` tests if parentheses are omitted, because it binds tighter than `:=` (see Chapter 9). Moreover, `:=` has special syntax rules that we'll omit here, but they encourage parenthesized usage by design.

Besides file reads and similar roles, the `:=` can be leveraged to reuse results in literals and get last results in comprehensions and other iteration tools, and can even be nested in f-strings and itself—with requisite parentheses:

```
>>> [val := 'Py!', val * 2, val * 3]                 # Re
['Py!', 'Py!Py!', 'Py!Py!Py!']

>>> list(pow := 2 ** num for num in [2, 4, 8])      # Co
[4, 16, 256]
>>> pow
256

>>> f'Hello {(name := input('Who are you? '))}'     # Ne
Who are you? Pat
'Hello Pat'
>>> name
'Pat'
```

```
>>> (x := (y := (z := 1) + 1) + 1)                    # Ne
3
>>> x, y, z
(3, 2, 1)
```

As another rule of thumb, bear in mind that a deeply nested and obscure `:=` expression might seem clever but will be a lot harder to read—and even *notice*—than a simple standalone assignment. You'll be able to judge this for yourself in examples in Chapter 20. Generally speaking, though, for the sake of others (including your future self!), use `:=`, like so many nestable tools, sparingly and wisely. Clarity is usually worth the extra line or two.

Variable Name Rules

Now that we've explored assignment statements and expressions, it's time to get more formal about the use of variable *names*. In Python, names come into existence when you assign values to them, but there are a few rules to follow when choosing names for the subjects of your programs:

Syntax: (underscore or letter) + (any number of letters, digits, or underscores)

> Variable names must start with an underscore or letter, which can be followed by any number of letters, digits, or underscores. `_hack`, `hack`, and `Hack_1` are legal names, but `1_hack`, `hack$`, and `@#!` are not.
>
> Python also allows non-ASCII *Unicode* characters to appear in variable names, but such characters may make your code difficult to use in some contexts and are subject to a handful of rules that require deep Unicode knowledge and are too arcane to cover here. See Chapter 37's note "Unicode in variable names" for expanded coverage of this topic, and Python's language manuals for full details.

Case matters: `HACK` is not the same as `hack`

> Python always pays attention to case in programs, both in names you create and in reserved words. For instance, the names `X` and `x` refer to two different variables. For portability, case also matters in the names of imported module files, even on platforms where the filesystems are case-insensitive (as is common in Windows). That way,

your imports still work after programs are copied to differing platforms.

Reserved words are off-limits

Names you define cannot be the same as words that mean special things in the Python language. For instance, Python will raise a syntax error if you try to use `class` as a variable name, but `klass` and `Class` work fine. Table 11-3 lists the words that are currently *reserved*—and hence off-limits for names of your own—in Python. Note: Python's docs call these "keywords" today, despite the longstanding and differing use of "keywords" for pass-by-name function arguments; to avoid confusion, this book uses "reserved words" for, well, reserved words.

Table 11-3. Python reserved words

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

In addition to Table 11-3 (and as partly leaked in Chapter 10), the words `match`, `case`, `_`, and `type` are *soft* reserved words: they are reserved only in the context of the statement to which they belong and can be used as variables anywhere else. As you'll learn in the next chapter, the first three are part of a multiple-choice `match` statement; the latter is used by a `type` statement that creates type aliases, used for the type hinting discussed at the end of Chapter 6.

As you can see, most of Python's reserved words are all lowercase. They are also all truly reserved—unlike names in the *built-in scope* that you will meet in the next part of this book, you cannot redefine reserved words by assignment. The statement `and = 1`, for instance, results in a syntax error.

Besides being of mixed case, the first three entries in Table 11-3— `False`, `None`, and `True` —are somewhat unusual in meaning: they also appear in the built-in scope of Python described in Chapter 17, and they are technically names assigned to objects. They are truly reserved in all other senses, though, and cannot be used for any other purpose in your script other than that of the objects they represent. All the other reserved words are hardwired into Python's syntax and can appear only in the specific contexts for which they are intended.[2]

Furthermore, because module names in `import` statements become variables in your scripts, variable name constraints extend to your *module filenames* too. For instance, you can code files called *and.py* and *my-code.py* and can run them as top-level scripts, but you cannot import them: their names without their *.py* extensions become *variables* in your code on imports, and so must follow all the variable rules just outlined. Hence, reserved words are off-limits, and dashes won't work, though underscores will. This module idea will be revisited in Part V of this book, where you'll find that this constraint also applies to names of "package" folders by extension.

PYTHON'S DEPRECATION POLICIES

Because making new words reserved has the potential to break code widely, such changes are generally phased into the language gradually. When a change might impact existing code, Python often makes it an option and begins issuing *deprecation warnings* one or more releases before the mod is officially enabled. The idea is that you should have time to notice the warnings and update your code before upgrading your Python. More recently, the "soft" reserved-word category has arisen to allow new reserved words to be added with neither warnings nor breakages. This seems a tacit recognition that deprecations only go so far.

Deprecation policies are not always followed, especially for major releases like 3.0 (which broke existing code wantonly), but also for newer changes deemed to be justifiable or innocuous by their promoters. In theory, changes are overseen by a *steering committee* charged with enforcing deprecation policies. While this helps in some cases, it is largely ineffectual in stemming the flood of opinionated morph and bloat in Python, which makes engineering reliable and durable software more challenging than it should be.

Even when deprecation warnings *are* issued, though, programs that cannot be easily changed will break if they have been distributed in source code form to users who innocently upgrade their local Python. Those on the receiving end of such flux may legitimately see deprecation warnings less as a courtesy than a bandage on a gaping wound. Warning that you're going to be rude doesn't make it OK to be rude!

Naming conventions

Besides these rules, there is also a set of naming *conventions*—rules that are not required but are followed in normal practice. For instance, because names with two leading and trailing underscores (e.g., `__name__`) generally have special meaning to the Python interpreter, you should avoid this pattern for your own names except in contexts where it is expected. Here is a list of the conventions Python follows:

- Names that begin with a single underscore (`_X`) are not imported by a `from module import *` statement, described in Chapter 23.

- Names that have two leading and trailing underscores (`__X__`) are system-defined names that have special meaning to the interpreter and provide implementation details in the user-defined OOP classes of Part VI.
- Names that begin with two underscores and do not *end* with two more (`__X`) are localized ("mangled") to enclosing classes, per the discussion of pseudoprivate attributes in Chapter 31.
- The name that is just a single underscore (`_`) retains the result of the last expression when you are working interactively at some REPLs and is a soft reserved word for a wildcard in `match` , as covered in Chapter 12.

In addition to these Python interpreter conventions, there are various other conventions that Python programmers usually follow. For instance, some programmers distinguish parts of long names using "camelCase" (`aLongName`), and others use underscores (`a_long_name`); either is completely valid according to both Python and this book.

Later in the book, you'll also see that *class* names commonly start with an uppercase letter and *module* names with a lowercase letter, and that the name `self` , though not reserved, usually has a special role in classes. Moreover, in Chapter 17 we'll study another, larger category of names known as the *built-ins*, which are predefined but not reserved (and so can be reassigned: `open = 99` silently works, though you might occasionally wish it didn't!).

Names have no type, but objects do

This is mostly review, but remember that it's crucial to keep Python's distinction between names and objects clear. As described in Chapter 6, objects have a type (e.g., integer, list) and may be mutable or not. Names (a.k.a. variables), on the other hand, are always just references to objects; they have no notion of mutability and have no associated type information, apart from the type of the object they happen to reference at a given point in time.

Thus, it's OK to assign the same name to different kinds of objects at different times:

```
>>> x = 0                 # x bound to an integer object
>>> x = 'Hello'           # Now it's a string
>>> x = [1, 2, 3]         # And now it's a list
```

Especially when we step up to functions and classes later in this book, you'll see that this generic nature of names can be a decided advantage in Python programming. In Chapter 17, you'll also learn that names live in something called a *scope*, which defines where they can be used; the place where you assign a name determines where it is visible.[3]

Expression Statements

In Python, you can use any *expression* as a statement, too—which usually means on a line by itself. Because the result of the expression won't be saved, though, it usually makes sense to do so in scripts only if the expression does something useful as a side effect. Expressions are commonly used as statements in two situations:

For calls to functions and methods

> Some functions do their work without returning a value. Such functions are sometimes called *procedures* in other languages. Because they don't return values that you might be interested in retaining, you can call these functions with expression statements. This also applies to methods, which are just functions with an implied subject.

For printing values at the interactive prompt

> As you certainly know by now, Python echoes back the results of expressions typed at the interactive command line. Technically, these are expression statements, too; they serve as a shorthand for typing `print` statements.

Table 11-4 lists some common expression statement forms in Python. Calls to functions and methods are coded with zero or more argument objects (really, expressions that evaluate to objects) in parentheses, after the function or method name.

Table 11-4. Common Python expression statements

Operation	Interpretation
`hack('Py', 3.12)`	Function calls
`code.hack('Py')`	Method calls
`hack`	Printing results in the interactive interpreter (REPL)
`print(a, b, c, sep='')`	Printing operations (a special function call)
`yield x ** 2`	Yielding expression statements (generators)
`await producer()`	Pausing for steps to finish (coroutines)

The last three entries in Table 11-4 are somewhat special cases. As you'll see later in this chapter, printing in Python is a function call usually coded as a statement by itself, and important enough to callout here. The `yield` and `await` operations on generator and coroutine functions (discussed in Chapter 20) are regularly coded as statements as well. All three, though, are really just instances of expressions masquerading as statements.

For instance, though you normally run a `print` call on a line by itself as an expression statement, it actually returns a value like any other function call—the return value is `None`, the default return value for functions that don't return anything meaningful (it requires another `print` to reveal in output):

```
>>> x = print('code')          # print is a function-cal
code
>>> print(x)                   # But is usually coded as
None
```

Also keep in mind that although expressions can appear as statements in Python, the *converse* is not true: statements cannot be used as expressions. A statement that is not also an expression must generally appear on a line all by itself, not nested in a larger syntactic structure.

For example, Python doesn't allow you to embed basic assignment statements (`=`) in other expressions. The rationale for this is that it avoids common coding mistakes; you can't accidentally change a variable by typing `=` when you really mean to use the `==` equality test. If you really miss this coding pattern from other languages, though, the newer and visually distinct `:=` named-assignment expression we met earlier works the same way with less chance of being confused with `==`, and a variety of coding alternatives achieve similar goals (e.g., see `while` coverage in Chapter 13).

Expression Statements and In-Place Changes

This brings up a common mistake in Python work, which we've encountered before, but is so pervasive that it merits another quick nag here. Expression statements are often used to run list methods that change a list in place:

```
>>> L = [1, 2]
>>> L.append(3)               # Append is an in-place c
>>> L
[1, 2, 3]
```

It's not unusual, though, for Python newcomers to code this as an assignment statement instead:

```
>>> L = L.append(4)           # But append returns None
>>> print(L)                  # So we lose our list!
None
```

But this doesn't work: in-place change methods like `append`, `sort`, and `reverse` always change the list in place, but do not return the list they have changed; instead, they return the `None` object. Assigning such an operation's result back to the variable name loses your reference to the list (and it's probably garbage-collected in the process).

So don't do that—call in-place change operations without assigning their results. We'll revisit this phenomenon in "Common Coding Gotchas", because it can also appear in the context of some looping statements we'll explore in the chapters ahead.

Print Operations

In Python, `print` prints things—it's simply a programmer-friendly interface to the standard output stream. It's a function-call expression that we're calling out here because it's so pervasive and is usually coded as a statement.

Technically, printing converts one or more objects to their textual representations, adds some minor formatting, and sends the resulting text to either standard output or another file-like stream. In a bit more detail, `print` is strongly bound up with the notions of *files and streams* in Python:

File object methods

> In Chapter 9, we explored file object methods that write text (e.g., `file.write(str)`). Printing operations are similar, but more focused—whereas file write methods write strings to arbitrary files, `print` writes objects to the `stdout` stream by default, with some automatic conversion and formatting added. Unlike with file methods, there is no need to convert objects to strings when using print operations.

Standard output stream

> The standard output stream (often known as `stdout`) is simply a default place to send a program's text output. Along with the standard input and error streams, it's one of three data connections created when your script starts. The standard output stream is usually mapped to the window where you started your Python program or REPL, unless it's been redirected to a file or pipe in your operating system's shell or rerouted by your coding GUI.
>
> Because the standard output stream is available in Python as the `stdout` file object in the built-in `sys` module (i.e., `sys.stdout`), it's possible to emulate `print` with file write method calls. However, `print` is noticeably simpler to use in most roles and makes it easy to print text to other files and streams.

NOTE

Blast from the past: Printing is also one of the most visible places where Python 3.X and 2.X diverged: it morphed from statement in 2.X to built-in function in 3.X—and broke nearly every existing Python program in the process. This book no longer covers 2.X's statement, but the legacy of 3.X's backward incompatibilities lives on as a natural brake on language changes (and in the muscle memory of Python coders who still type `print x` unconsciously!).

The print Function

Strictly speaking, printing is not a separate statement form. Instead, it is simply an instance of the *expression statement* we studied in the preceding section. The `print` built-in function is normally called on a line of its own, because it doesn't return any value we care about (technically, it returns `None`, per the preceding section). Because it is a normal function, though, it can use standard function-call syntax (including keyword arguments for special operation modes) and may be both passed around as an object and reassigned to a different implementation.

Call format

Syntactically, calls to the `print` function have the following form:

```
print([object, …][, sep=' '][, end='\n'][, file=sys.stc
```

In this notation, items in square brackets are optional and may be omitted in a given call, and values after `=` give keyword-argument defaults. In English, this built-in function prints the textual representation of one or more `object`s, separated by the string `sep` and followed by the string `end`, to the stream `file`, flushing buffered output or not per `flush`.

The `sep`, `end`, `file`, and `flush` parts, if present, must be given as *keyword arguments*—that is, you must use a *`name=value`* syntax to pass the arguments by name instead of position. Keyword arguments are covered in depth in Chapter 18, but they're straightforward to use. The keyword arguments sent to this call may appear in any left-to-right order following the objects to be printed, and they control the `print` operation:

- `sep` is a string inserted between each object's text, which defaults to a single space if not passed. Passing an empty string suppresses separators altogether.
- `end` is a string added at the end of the printed text, which defaults to a `\n` newline character if not passed. Passing an empty string avoids dropping down to the next output line at the end of the printed text—the next `print` will keep adding to the end of the current output line.
- `file` specifies the file, standard stream, or other file-like object to which the text will be sent. It defaults to the `sys.stdout` standard output stream if not passed, but any object with a file-like `write(`*`string`*`)` method may be passed, including real file objects that have already been opened for output to external files.
- `flush` defaults to `False`. It allows prints to mandate that their text be flushed through the output stream immediately to any waiting recipients. Normally, whether printed output is buffered in memory or not is determined by the `file` object; passing a true value to `flush` forcibly flushes the stream.

The textual representation of each *`object`* to be printed is obtained by passing the object to the `str` built-in call (or its equivalent inside Python); as we've seen, this built-in returns a "user friendly" display string for any object.[4] With no arguments at all, the `print` function simply prints a newline character to the standard output stream, which usually displays a blank line.

The print function in action

Printing is probably simpler than its full details may imply. To illustrate, let's run some quick examples in the REPL. The following prints a variety of object types to the default standard output stream, with the default separator and end-of-line formatting added (these are the defaults because they are the most common use case):

```
>>> print()                                              # Disp

>>> x = 'python'
>>> y = 3.12
>>> z = ['lp6e']
>>> print(x, y, z)                                          # Prir
python 3.12 ['lp6e']
```

There's no need to convert objects to strings here, as would be required for file write methods. By default, `print` calls add a space between the objects printed. To suppress this, send an empty string to the `sep` keyword argument, or send an alternative separator of your choosing:

```
>>> print(x, y, z, sep='')                                  # Supp
python3.12['lp6e']
>>>
>>> print(x, y, z, sep=', ')                                # Cust
python, 3.12, ['lp6e']
```

Also by default, `print` adds an end-of-line character to terminate the output line. You can suppress this and avoid the line break altogether by passing an empty string to the `end` keyword argument, or you can pass a different terminator of your own including a `\n` character to break the line manually if desired (the second of the following is two statements on one line, separated by a semicolon to demo the effect of custom terminators in the REPL):

```
>>> print(x, y, z, end='')                                   # Sup
python 3.12 ['lp6e']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)                   # Two
python 3.12 ['lp6e']python 3.12 ['lp6e']
>>> print(x, y, z, end='...\n')                              # Cus
python 3.12 ['lp6e']...
>>>
```

You can also combine keyword arguments to specify both separators and end-of-line strings—they may appear in any order but must appear after all the objects being printed:

```
>>> print(x, y, z, sep='...', end='!\n')                     # Mul
python...3.12...['lp6e']!
>>> print(x, y, z, end='!\n', sep='...')                     # Orc
python...3.12...['lp6e']!
```

Here is how the `file` keyword argument is used—it directs the printed text to an open output file or other compatible object for the duration of the single `print` (this is really a form of stream redirection, a topic we will revisit later in this section):

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'
>>> print(x, y, z)
python 3.12 ['lp6e']
>>> print(open('data.txt').read())
python...3.12...['lp6e']
```

Finally, keep in mind that the separator and end-of-line options provided by print operations are just conveniences. If you need to display more specific formatting, don't print this way. Instead, build up a more custom string either ahead of time or within the `print` itself, using the string tools we mastered in Chapter 7—and print the string all at once. String *formatting* expressions and f-strings, for example, were designed for this sort of job:

```
>>> text = '%s: %-.4f, %05d' % (x, y, int(z[0][-2]))
>>> print(text)
python: 3.1200, 00006

>>> print(f'{x}: {y:-0.4f}, {int(z[0][-2]):05d}')
python: 3.1200, 00006
```

On the other hand, there seems to be a misconception that f-strings are somehow required for `print`, and they crop up needlessly. In truth, f-strings are overkill if you're simply trying to print objects separated by spaces:

```
>>> a, b, c, = 11, 3.14, 'hack'
>>> print(f'{a} {b} {c}')
11 3.14 hack
>>> print(a, b, c)
11 3.14 hack
```

As usual, don't use a sledgehammer to drive every nail!

Print Stream Redirection

As we’ve seen, `print` sends text to the standard output stream by default. However, it’s often useful to send it elsewhere—to a text file, for example, to save results for later use or testing purposes. Although such redirection can usually be accomplished in system shells outside Python itself (with syntax like `> file`, per “Command-Line Usage Variations”), this is not a Python tool, and it’s just as easy to redirect a script’s streams from Python code.

The Python “hello world” program

Let’s start off with the usual (and largely pointless) language benchmark—the “hello world” program. To print a “hello world” message in Python, simply print the string with print:

```
>>> print('hello world')                  # Print a string
hello world
```

Really, because expression results are echoed on the interactive command line, you often don’t even need to use a `print` statement there—simply type the expressions you’d like to have printed, and their results are echoed back:

```
>>> 'hello world'                         # Interactive ec
'hello world'
```

This code isn’t exactly an earth-shattering feat of software mastery, but it serves to illustrate printing behavior. Technically, though, the `print` operation is just an *ergonomic* feature of Python—it provides a simple interface to the `sys.stdout` object, with a bit of default formatting. In fact, if you enjoy working harder than you must (or are pining for your Java coding days) you can also code print operations this way, which omits the `write` call’s return value for space, as in Chapter 9:

```
>>> import sys                            # Printing the h
>>> sys.stdout.write('hello world\n')
hello world
```

This code explicitly calls the `write` method of `sys.stdout` —an attribute preset when Python starts up to an open file object connected to the output stream. The `print` operation hides most of those details, providing a simple tool for simple printing tasks.

Manual stream redirection

So, why bother learning the hard way to print? The `sys.stdout` equivalent to `print` turns out to be the basis of a common technique in Python. In general, `print` and `sys.stdout` are directly related as follows. This statement:

```
print(X, Y)
```

is equivalent to the longer:

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

which manually performs a string conversion with `str`, adds a separator and newline with `+`, and calls the output stream's `write` method. That is, this is what the `print` does. Which would you rather code?

Obviously, the long form isn't all that useful for printing by itself. However, it is useful to know that this is exactly what `print` operations do because it is possible to *reassign* `sys.stdout` to something different from the standard output stream. In other words, this equivalence provides a way of making your `print` operations send their text to other places. For example:

```
import sys
sys.stdout = open('log.txt', 'a')       # Redirects pri
…
print(x, y, x)                          # Now printed t
```

Here, we reset `sys.stdout` to a manually opened file named *log.txt*, located in the script's working directory (CWD) and opened in `'a'` append mode (so we add to its current content). After the reset, every `print` operation anywhere in the program will write its text to the end of the file *log.txt* instead of to the original output stream. The `print` operations are happy to keep calling `sys.stdout`'s `write` method, no matter what `sys.stdout` happens to refer to. And because there is just one `sys` module in your program (technically, process), assigning `sys.stdout` this way will redirect every `print` anywhere in your program.

In fact, as the sidebar "Why You Will Care: print and stdout" will explain, you can even reset `sys.stdout` to an object that isn't a file at all, as long as it has the expected interface: a method named `write` to receive the printed text-string argument. When that object is a *class*, printed text can be routed and processed arbitrarily per a `write` method you code yourself.

This trick of resetting the output stream might be more useful for programs originally coded with `print` statements. If you know that output should go to a file to begin with, you can always call file write methods instead. To redirect the output of a `print`-based program, though, resetting `sys.stdout` provides a convenient alternative to changing every `print` statement or using system shell-based redirection syntax, which may be above users' pay grades.

In other roles, streams may be reset to objects that display them in pop-up windows in GUIs, colorize them in IDEs like IDLE, and so on. It's a general technique.

Automatic stream redirection

Although redirecting printed text by assigning `sys.stdout` is a useful tool, a potential problem with the last section's code is that there is no direct way to restore the original output stream should you need to switch back after printing to a file. Because `sys.stdout` is just a normal file object, though, you can always save it and restore it if needed:[5]

```
>>> import sys
>>> temp = sys.stdout                      # Save for rest
>>> sys.stdout = open('log.txt', 'a')      # Redirect prir

>>> print('lp6e was here')                 # Prints go to
>>> print(1, 2, 3)
>>> sys.stdout.close()                     # Flush output

>>> sys.stdout = temp                      # Restore origi
>>> print('back in the REPL')                 # Prints show u
back in the REPL
>>> print(open('log.txt').read())             # Result of ear
lp6e was here
1 2 3
```

As you can see, though, manual saving and restoring of the original output stream like this involves extra juggling work. Because this crops up fairly often, a `print` extension is available to make it unnecessary.

As introduced earlier, the `file` keyword allows a single `print` call to send its text to the `write` method of a file (or file-like object), without actually resetting `sys.stdout`. Because the redirection is temporary, normal `print` calls keep printing to the original output stream. For example, the following abstract snippet again sends printed text to a file named *log.txt*:

```
log = open('log.txt', 'a')
print(x, y, z, file=log)                      # Print to a fi
print(a, b, c)                                # Print to orig
```

These redirected forms of `print` are handy if you need to print to *both* files and the standard output stream in the same program. If you use these forms, however, be sure to give them a file object (or an object that has the same `write` method as a file object), not a file's name string. Here is the technique in action live:

```
>>> log = open('log2.txt', 'w')
>>> print(1, 2, 3, file=log)
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)
7 8 9
>>> print(open('log2.txt').read())
1 2 3
4 5 6
```

These extended forms of `print` are also commonly used to print error messages to the standard *error* stream, available to your script as the preopened file object `sys.stderr`. You can either use its file `write` methods and format the output manually, or print with redirection syntax:

```
>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

>>> print('Bad!' * 8, file=sys.stderr)
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
```

Now that you know all about print redirections, the equivalence between printing and file `write` methods should be clear. The following demo prints both ways, then redirects the output to an external file to verify that the same text is printed (on Unix, you won't get the `\r` added to newlines here on Windows, and `write` results are back here):

```
>>> import sys
>>> X, Y = 1, 2

>>> print(X, Y)
1 2
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
1 2
4

>>> print(X, Y, file=open('temp1', 'w'))
>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\
4

>>> print(open('temp1', 'rb').read())
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'
```

As you can see, unless you happen to enjoy typing, print operations are often the best option for displaying text. For another example of the equivalence between prints and file writes, watch for a `print` function emulation example in Chapter 18; it uses the coding patterns here to provide a `print` equivalent that can be customized.

Chapter Summary

In this chapter, we began our in-depth look at Python statements by exploring assignments, expressions, and print operations. Although these are generally simple to use, they have some alternative forms that, while optional, are often convenient in practice—for example, augmented and named assignments, as well as the redirection form of `print` operations, allow us to avoid some manual coding work. Along the way, we also studied the syntax of variable names, stream redirection techniques, and a variety of common mistakes to avoid, such as assigning the result of an `append` method call back to a variable.

In the next chapter, we'll continue our statement tour by filling in details about the `if` statement, Python's main selection tool; there, we'll also revisit Python's syntax model in more depth and look at the behavior of Boolean expressions, as well as the `match` multiple-choice statement. Before we move on, though, the end-of-chapter quiz will test your knowledge of what you've learned here.

Test Your Knowledge: Quiz

1. Name three ways that you can assign three variables to the same value.
2. What's dangerous about assigning three variables to a mutable object?
3. What's wrong with saying `L = L.sort()` ?
4. How might you use the `print` operation to send text to an external file?

Test Your Knowledge: Answers

1. You can use multiple-target assignments (`A = B = C = 0`), sequence assignment (`A, B, C = 0, 0, 0`), or multiple assignment statements on three separate lines (`A = 0` , `B = 0` , and `C = 0`). With the latter technique, as introduced in Chapter 10, you can also string the three separate statements together on the same line by separating them with semicolons (`A = 0; B = 0; C = 0`).
2. If you assign them this way: `A = B = C = []` , then all three names reference the same object, so changing it in place from one (e.g., `A.append(99)`) will affect the others. This is true only for in-place

changes to mutable objects like lists and dictionaries; for immutable objects such as numbers and strings, this issue is irrelevant because they can never be changed in place.

3. The list `sort` method is like `append` in that it makes an in-place change to the subject list—it returns `None`, not the list it changes. The assignment back to `L` sets `L` to `None`, not to the sorted list. As discussed both earlier and later in this book (e.g., Chapter 8), a newer built-in function, `sorted`, sorts any sequence and returns a new list with the sorting result; because this is not an in-place change, its result can be safely and meaningfully assigned to a name.
4. To print to a file for a single `print` operation, you can use the `print(X, file=F)` call form, or assign `sys.stdout` to a manually opened file before the `print` and restore the original after if needed. You can also redirect all of a program's printed text to a file with special syntax in the system shell like `> file`, but this is outside Python's scope.

WHY YOU WILL CARE: PRINT AND STDOUT

The equivalence between the `print` operation and writing to `sys.stdout` makes it possible to reassign `sys.stdout` to any user-defined object that provides the same `write` method as files. Because the `print` statement just sends text to the `sys.stdout.write` method, you can capture printed text in your programs by assigning `sys.stdout` to an object whose `write` method processes the text in arbitrary ways.

For instance, you can send printed text to a GUI window, or tee it off to multiple destinations, by defining an object with a `write` method that does the required routing. You'll see an example of this trick when we study classes in Part VI of this book, but as an abstract preview, it looks like this:

```
class FileFaker:
    def write(self, string):
        # Do something with the printed text in string

import sys
sys.stdout = FileFaker()
print(someObjects)              # Sends text to class w
```

This works because `print` is what we will call in the next part of this book a *polymorphic* operation—it doesn't care what `sys.stdout` is, only that it has a method (i.e., interface) called `write`. This redirection to objects is made even simpler with the `file` keyword argument of `print`, because we don't need to reset `sys.stdout` explicitly—normal prints will still be routed to the `stdout` stream:

```
myobj = FileFaker()             # Redirect to object fo
print(someObjects, file=myobj)  # Does not reset sys.st
```

Python's built-in `input` function *reads* from the `sys.stdin` file, so you can intercept read requests in a similar way, using classes that implement file-like `read` methods instead. See the `input` and `while` loop example in Chapter 10 for more background on this function.

Notice that because printed text goes to the `stdout` stream, it's also the way to print HTML reply pages in server-side scripts used on the web, and enables you to redirect Python script input and output at the operating system's shell command line as usual:

```
python script.py < inputfile > outputfile
python script.py | filterProgram
```

Python's print operation redirection tools are essentially pure-Python alternatives to some of these shell syntax forms. See other resources for more on web scripts and shell syntax.

---

1 C/C++ programmers also take note: although Python now supports statements like `X += Y`, it still does not have C's auto-increment/decrement operators (e.g., `X++`, `--X`). These don't quite map to the Python object model because Python has no notion of *in-place* changes to immutable objects like numbers. As a preview of the next section, there also are no augmented-named expression forms today; `:+= 1` would be close to `++` but thankfully is fiction. That said, we eventually got `+=` and `:=`, so…

2 Exception: Alternative implementations of Python such as Jython might allow reserved words to appear as identifiers in some contexts. See Chapter 2 for more on alternative Pythons.

3 If you've used a more restrictive language like C++, you may be interested to know that there is no notion of C++'s `const` declaration in Python; certain objects may be *immutable*, but names can always be assigned. Python also has ways to hide names in classes and modules, but they're not the same as C++'s declarations (if hiding attributes matters to you, see the coverage of `_X` module names in Chapter 25, `__X` class names in Chapter 31, and the `Private` and `Public` class decorators example in Chapter 39).

4 Technically, printing uses the *equivalent* of `str` in the internal implementation of Python, but the effect is the same. Besides this to-string conversion role, `str` is also the name of the string data type and can be used to decode Unicode strings from raw bytes with an extra encoding argument, as you'll learn in Chapter 37; this latter role is an advanced usage that you can safely ignore here.

5 You may also be able to use the `__stdout__` attribute in the `sys` module, which refers to the original value `sys.stdout` had at program startup time. You still need to restore `sys.stdout` to `sys.__stdout__` to go back to this original stream value, though. See the `sys` module documentation for more details. Also note that `sys.stdout` and its cohorts may be `None` in some GUI programs with no console on which to display tests; be sure to check this where it matters.