Chapter 3. The Python Language

This chapter is a guide to the Python language. To learn Python from scratch, we suggest you start with the appropriate links from the **online docs** and the resources mentioned in **"Python documentation for non-programmers"**. If you already know at least one other programming language well, and just want to learn specifics about Python, this chapter is for you. However, we're not trying to teach Python: we cover a lot of ground at a pretty fast pace. We focus on the rules, and only secondarily point out best practices and style; as your Python style guide, use **PEP 8** (optionally augmented by extra guidelines such as those of **"The Hitchhiker's Guide to Python"**, **CKAN**, and **Google**).

Lexical Structure

The *lexical structure* of a programming language is the set of basic rules that govern how you write programs in that language. It is the lowest-level syntax of the language, specifying such things as what variable names look like and how to denote comments. Each Python source file, like any other text file, is a sequence of characters. You can also usefully consider it a sequence of lines, tokens, or statements. These different lexical views complement each other. Python is very particular about program layout, especially lines and indentation: pay attention to this information if you are coming to Python from another language.

Lines and Indentation

A Python program is a sequence of *logical lines*, each made up of one or more *physical lines*. Each physical line may end with a comment. A hash sign (#) that is not inside a string literal starts a comment. All characters after the #, up to but excluding the line end, are the comment: Python ig-

nores them. A line containing only whitespace, possibly with a comment, is a *blank line*: Python ignores it. In an interactive interpreter session, you must enter an empty physical line (without any whitespace or comment) to terminate a multiline statement.

In Python, the end of a physical line marks the end of most statements. Unlike in other languages, you don't normally terminate Python statements with a delimiter, such as a semicolon (`;`). When a statement is too long to fit on a physical line, you can join two adjacent physical lines into a logical line by ensuring that the first physical line does not contain a comment and ends with a backslash (`\`). More elegantly, Python also automatically joins adjacent physical lines into one logical line if an open parenthesis (`(`), bracket (`[`), or brace (`{`) has not yet been closed: take advantage of this mechanism to produce more readable code than you'd get with backslashes at line ends. Triple-quoted string literals can also span physical lines. Physical lines after the first one in a logical line are known as *continuation lines*. Indentation rules apply to the first physical line of each logical line, not to continuation lines.

Python uses indentation to express the block structure of a program. Python does not use braces, or other begin/end delimiters, around blocks of statements; indentation is the only way to denote blocks. Each logical line in a Python program is *indented* by the whitespace on its left. A *block* is a contiguous sequence of logical lines, all indented by the same amount; a logical line with less indentation ends the block. All statements in a block must have the same indentation, as must all clauses in a compound statement. The first statement in a source file must have no indentation (i.e., must not begin with any whitespace). Statements that you type at the interactive interpreter primary prompt, `>>>` (covered in **"Interactive Sessions"**), must also have no indentation.

Python treats each tab as if it was up to 8 spaces, so that the next character after the tab falls into logical column 9, 17, 25, and so on. Standard Python style is to use four spaces (*never* tabs) per indentation level.

If you must use tabs, Python does not allow mixing tabs and spaces for indentation.

USE SPACES, NOT TABS

Configure your favorite editor to expand a Tab keypress to four spaces, so that all Python source code you write contains just spaces, not tabs. This way, all tools, including Python itself, are consistent in handling indentation in your Python source files. Optimal Python style is to indent blocks by exactly four spaces; use no tab characters.

Character Sets

A Python source file can use any Unicode character, encoded by default as UTF-8. (Characters with codes between 0 and 127, the 7-bit *ASCII characters*, encode in UTF-8 into the respective single bytes, so an ASCII text file is a fine Python source file, too.)

You may choose to tell Python that a certain source file is written in a different encoding. In this case, Python uses that encoding to read the file. To let Python know that a source file is written with a nonstandard encoding, start your source file with a comment whose form must be, for example:

```
# coding: iso-8859-1
```

After `coding:`, write the name of an ASCII-compatible codec from the `codecs` module, such as `utf-8` or `iso-8859-1`. Note that this *coding directive* comment (also known as an *encoding declaration*) is taken as such only if it is at the start of a source file (possibly after the "shebang line" covered in **"Running Python Programs"**). Best practice is to use `utf-8` for all of your text files, including Python source files.

Tokens

Python breaks each logical line into a sequence of elementary lexical components known as *tokens*. Each token corresponds to a substring of the logical line. The normal token types are *identifiers*, *keywords*, *operators*, *delimiters*, and *literals*, which we cover in the following sections. You

may freely use whitespace between tokens to separate them. Some whitespace separation is necessary between logically adjacent identifiers or keywords; otherwise, Python would parse them as a single longer identifier. For example, `ifx` is a single identifier; to write the keyword **`if`** followed by the identifier `x`, you need to insert some whitespace (typically only one space character, i.e., **`if`** `x`).

Identifiers

An *identifier* is a name used to specify a variable, function, class, module, or other object. An identifier starts with a letter (that is, any character that Unicode classifies as a letter) or an underscore (_), followed by zero or more letters, underscores, digits, or other characters that Unicode classifies as letters, digits, or combining marks (as defined in **Unicode Standard Annex #31**).

For example, in the Unicode Latin-1 character range, the valid leading characters for an identifier are:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
ªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
```

After the leading character, the valid identifier body characters are just the same, plus the digits and · (Unicode MIDDLE DOT) character:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
ª·µºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
```

Case is significant: lowercase and uppercase letters are distinct. Punctuation characters such as `@`, `$`, and `!` are not allowed in identifiers.

BEWARE OF USING UNICODE CHARACTERS THAT ARE HOMOGLYPHS

Some Unicode characters look very similar to, if not indistinguishable from, other characters. Such character pairs are called *homoglyphs*. For instance, compare the capital letter A and the capital Greek letter alpha (Α). These are actually two different letters that just look very similar (or identical) in most fonts. In Python, they define two different variables:

```
>>> A = 100
>>> # this variable is GREEK CAPITAL LETTER ALPHA:
>>> A = 200
>>> print(A, A)
```

```
100 200
```

If you want to make your Python code widely usable, we recommend a policy that all identifiers, comments, and documentation are written in English, avoiding, in particular, non-English homoglyph characters. For more information, see **PEP 3131**.

Unicode normalization strategies add further complexities (Python uses **NFKC normalization** when parsing identifiers containing Unicode characters). See Jukka K. Korpela's ***Unicode Explained*** (O'Reilly) and other technical information provided on the **Unicode website** and in the **books that site references** for more information.

AVOID NORMALIZABLE UNICODE CHARACTERS IN IDENTIFIERS

Python may create unintended aliases between variables when names contain certain Unicode characters, by internally converting the name as shown in the Python script to one using normalized characters. For example, the letters ª and º normalize to the ASCII lowercase letters a and o, so variables using these letters could clash with other variables:

```
>>> a, o = 100, 101
>>> ª, º = 200, 201
>>> print(a, o, ª, º)
```

```
200 201 200 201  # expected "100 101 200 201"
```

It is best to avoid using normalizable Unicode characters in your Python identifiers.

Normal Python style is to start class names with an uppercase letter, and most[1] other identifiers with a lowercase letter. Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private. Starting an identifier with two leading underscores indicates a *strongly private* identifier; if the identifier also *ends* with two trailing underscores, however, this means that it's a language-defined special name. Identifiers composed of multiple words should be all lowercase with underscores between words, as in `login_password`. This is sometimes referred to as *snake case*.

THE SINGLE UNDERSCORE (_) IN THE INTERACTIVE INTERPRETER

The identifier _ (a single underscore) is special in interactive interpreter sessions: the interpreter binds _ to the result of the last expression statement it has evaluated interactively, if any.

Keywords

Python has 35 *keywords*, or identifiers that it reserves for special syntactic uses. Like identifiers, keywords are case-sensitive. You cannot use keywords as regular identifiers (thus, they're sometimes known as "reserved words"). Some keywords begin simple statements or clauses of compound statements, while other keywords are operators. We cover all the keywords in detail in this book, either in this chapter or in Chapters 4, 6, and 7. The keywords in Python are:

and	break	elif	from	is	pass	w
as	class	else	global	lambda	raise	y
assert	continue	except	if	nonlocal	return	F
async	def	finally	import	not	try	N
await	del	for	in	or	while	T

You can list them by importing the `keyword` module and printing `keyword.kwlist`.

3.9+ In addition, Python 3.9 introduced the concept of *soft keywords*, which are keywords that are context sensitive. That is, they are language keywords for some specific syntax constructs, but outside of those constructs they may be used as variable or function names, so they are not *reserved* words. No soft keywords were defined in Python 3.9, but Python 3.10 introduced the following soft keywords:

_	case	match

You can list them from the `keyword` module by printing `keyword.softkwlist`.

Operators

Python uses nonalphanumeric characters and character combinations as operators. Python recognizes the following operators, which are covered in detail in **"Expressions and Operators"**:

```
+   -   *   /   %    **   //   <<   >>   &    @
|   ^   ~   <   <=   >    >=   !=   ==   @=   :=
```

You can use `@` as an operator (in matrix multiplication, covered in **Chapter 16**), although (pedantically speaking!) the character is actually a delimiter.

Delimiters

Python uses the following characters and combinations as delimiters in various statements, expressions, and list, dictionary, and set literals and comprehensions, among other purposes:

```
(     )     [     ]     {     }
,     :     .     =     ;     @
+=    -=    *=    /=    //=   %=
&=    |=    ^=    >>=   <<=   **=
```

The period (`.`) can also appear in floating-point literals (e.g., `2.3`) and imaginary literals (e.g., `2.3j`). The last two rows are the augmented assignment operators, which are delimiters but also perform operations. We discuss the syntax for the various delimiters when we introduce the objects or statements using them.

The following characters have special meanings as part of other tokens:

```
'  "  #  \
```

' and " surround string literals. # outside of a string starts a comment, which ends at the end of the current line. \ at the end of a physical line joins the following physical line with it into one logical line; \ is also an escape character in strings. The characters $ and ?, and all control characters[2] except whitespace, can never be part of the text of a Python program, except in comments or string literals.

Literals

A *literal* is the direct denotation in a program of a data value (a number, string, or container). The following are number and string literals in Python:

```
42                    # Integer literal
3.14                  # Floating-point literal
1.0j                  # Imaginary literal
'hello'               # String literal
"world"               # Another string literal
"""Good
night"""              # Triple-quoted string literal, spanning 2 lines
```

Combining number and string literals with the appropriate delimiters, you can directly build many container types with those literals as values:

```
[42, 3.14, 'hello']     # List
[]                      # Empty list
100, 200, 300           # Tuple
(100, 200, 300)         # Tuple
()                      # Empty tuple
{'x':42, 'y':3.14}      # Dictionary
{}                      # Empty dictionary
{1, 2, 4, 8, 'string'}  # Set
# There is no literal form to denote an empty set; use set() instead
```

We cover the syntax for such container literals[3] in detail in **“Data Types”**, when we discuss the various data types Python supports. We refer to these expressions as literals throughout this book, as they describe literal (i.e., not requiring additional evaluation) values in the source code.

Statements

You can look at a Python source file as a sequence of simple and compound statements.

Simple statements

A *simple statement* is one that contains no other statements. A simple statement lies entirely within a logical line. As in many other languages, you may place more than one simple statement on a single logical line, with a semicolon (`;`) as the separator. However, using one statement per line is the usual and recommended Python style, and it makes programs more readable.

Any *expression* can stand on its own as a simple statement (we discuss expressions in **“Expressions and Operators”**). When you’re working interactively, the interpreter shows the result of an expression statement you enter at the prompt (`>>>`) and binds the result to a global variable named `_` (underscore). Apart from interactive sessions, expression statements are useful only to call functions (and other *callables*) that have side effects (e.g., perform output, change arguments or global variables, or raise exceptions).

An *assignment* is a simple statement that assigns values to variables, as we discuss in **“Assignment Statements”**. An assignment in Python using the `=` operator is a statement and can never be part of an expression. To perform an assignment as part of an expression, you must use the `:=` (known as the “walrus”) operator. You’ll see some examples of using `:=` in **“Assignment Expressions”**.

Compound statements

A *compound statement* contains one or more other statements and controls their execution. A compound statement has one or more *clauses*, aligned at the same indentation. Each clause has a *header* starting with a keyword and ending with a colon (`:`), followed by a *body*, which is a sequence of one or more statements. Normally, these statements, also known as a *block*, are on separate logical lines after the header line, indented four spaces rightward. The block lexically ends when the indentation returns to that of the clause header (or further left from there, to the indentation of some enclosing compound statement). Alternatively, the body can be a single simple statement following the `:` on the same logical line as the header. The body may also consist of several simple statements on the same line with semicolons between them, but, as we've already mentioned, this is not good Python style.

Data Types

The operation of a Python program hinges on the data it handles. Data values in Python are known as *objects*; each object, aka *value*, has a *type*. An object's type determines which operations the object supports (in other words, which operations you can perform on the value). The type also determines the object's *attributes* and *items* (if any) and whether the object can be altered. An object that can be altered is known as a *mutable object*, while one that cannot be altered is an *immutable object*. We cover object attributes and items in **"Object attributes and items"**.

The built-in `type(`*`obj`*`)` function accepts any object as its argument and returns the type object that is the type of *`obj`*. The built-in function `isinstance(`*`obj, type`*`)` returns **`True`** when object *`obj`* has type *`type`* (or any subclass thereof); otherwise, it returns **`False`**. The *`type`* argument of `isinstance` may also be a tuple of types (3.10+ or multiple types joined with the `|` operator), in which case it returns **`True`** if the type of *`obj`* matches any of the given types, or any subclasses of those types.

Python has built-in types for fundamental data types such as numbers, strings, tuples, lists, dictionaries, and sets, as covered in the following sections. You can also create user-defined types, known as *classes*, as discussed in **"Classes and Instances"**.

Numbers

The built-in numeric types in Python include integers, floating-point numbers, and complex numbers. The standard library also offers decimal floating-point numbers, covered in **"The decimal Module"**, and fractions, covered in **"The fractions Module"**. All numbers in Python are immutable objects; therefore, when you perform an operation on a number object, you produce a new number object. We cover operations on numbers, also known as arithmetic operations, in **"Numeric Operations"**.

Numeric literals do not include a sign: a leading `+` or `-`, if present, is a separate operator, as discussed in **"Arithmetic Operations"**.

Integer numbers

Integer literals can be decimal, binary, octal, or hexadecimal. A decimal literal is a sequence of digits in which the first digit is nonzero. A binary literal is `0b` followed by a sequence of binary digits (`0` or `1`). An octal literal is `0o` followed by a sequence of octal digits (`0` to `7`). A hexadecimal literal is `0x` followed by a sequence of hexadecimal digits (`0` to `9` and `A` to `F`, in either upper- or lowercase). For example:

```
1, 23, 3493                    # Decimal integer literals
0b010101, 0b110010, 0B01       # Binary integer literals
0o1, 0o27, 0o6645, 0O777       # Octal integer literals
0x1, 0x17, 0xDA5, 0xda5, 0Xff  # Hexadecimal integer literals
```

Integers can represent values in the range `±2**sys.maxsize`, or roughly $\pm 10^{2.8e18}$.

Table 3-1 lists the methods supported by an `int` object `i`.

Table 3-1. `int` methods

`as_integer_ratio`	`i.as_integer_ratio()` **3.8+** Returns a tuple of two `int`s, whose exact ratio is the original integer value. (Since `i` is always `int`, the tuple is always `(i, 1)`; compare with `float.as_integer_ratio`.)
`bit_count`	`i.bit_count()` **3.10+** Returns the number of ones in the binary representation of `abs(i)`.
`bit_length`	`i.bit_length()` Returns the minimum number of bits needed to represent `i`. Equivalent to the length of the binary representation of `abs(i)`, after removing `'b'` and all leading zeros. `(0).bit_length()` returns `0`.
`from_bytes`	`int.from_bytes(`*`bytes_value`*`, `*`byteorder`*`, *, `*`signed`*`=`**`False`**`)` Returns an `int` from the bytes in *`bytes_value`* following the same argument usage as in `to_bytes`. (Note that `from_bytes` is a class method of `int`.)
`to_bytes`	`i.to_bytes(`*`length`*`, `*`byteorder`*`, *, `*`signed`*`=`**`False`**`)` Returns a `bytes` value *`length`* bytes in size representing the binary value of `i`. *`byteorder`* must be the `str` value `'big'` or `'little'`, indicating whether the return value should be big-endian (most significant byte first) or little-endian (least significant byte first). For example, `(258).to_bytes(2, 'big')` returns `b'\x01\x02'`, and `(258).to_bytes(2, 'little')` returns `b'\x02\x01'`. When `i < 0` and *`signed`* is **`True`**, `to_bytes` returns the bytes of `i` represented in two's complement. When `i < 0` and *`signed`* is **`False`**, `to_bytes` raises `OverflowError`.

Floating-point numbers

A floating-point literal is a sequence of decimal digits that includes a decimal point (`.`), an exponent suffix (`e` or `E`, optionally followed by `+` or `-`, followed by one or more digits), or both. The leading character of a floating-point literal cannot be `e` or `E`; it may be any digit or a period (`.`) followed by a digit. For example:

```
0., 0.0, .0, 1., 1.0, 1e0, 1.e0, 1.0E0  # Floating-point literals
```

A Python floating-point value corresponds to a C `double` and shares its limits of range and precision: typically 53 bits—about 15 digits—of precision on modern platforms. (For the exact range and precision of floating-point values on the platform where the code is running, and many other details, see the online documentation on **`sys.float_info`**.)

Table 3-2 lists the methods supported by a `float` object *f*.

Table 3-2. `float` methods

`as_integer_ratio`	*f*`.as_integer_ratio()`
	Returns a tuple of two `int`s, a numerator and a denominator, whose exact ratio is the original float value, *f*. For example:
	`>>> f=2.5`
	`>>> f.as_integer_ratio()`
	`(5, 2)`

`from_hex`	`float.from_hex(`*s*`)` Returns a `float` value from the hexadecimal `str` value *s*. *s* can be of the form returned by `f.hex()`, or

	simply a string of hexadecimal digits. When the latter is the case, `from_hex` returns `float(int(s, 16))`.
`hex`	*f*`.hex()` Returns a hexadecimal representation of *f*, with leading `0x` and trailing `p` and exponent. For example, `(99.0).hex()` returns `'0x1.8c00000000000p+6'`.
`is_integer`	*f*`.is_integer()` Returns a `bool` value indicating if *f* is an integer value. Equivalent to `int(`*f*`) == `*f*.

Complex numbers

A complex number is made up of two floating-point values, one each for the real and imaginary parts. You can access the parts of a complex object *z* as read-only attributes *z*`.real` and *z*`.imag`. You can specify an imaginary literal as any floating-point or integer decimal literal followed by a `j` or `J`:

```
0j, 0.j, 0.0j, .0j, 1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j
```

The `j` at the end of the literal indicates the square root of `-1`, as commonly used in electrical engineering (some other disciplines use `i` for this purpose, but Python uses `j`). There are no other complex literals. To denote any constant complex number, add or subtract a floating-point (or integer) literal and an imaginary one. For example, to denote the complex number that equals 1, use expressions like `1+0j` or `1.0+0.0j`. Python performs the addition at compile time, so there's no need to worry about overhead.

A `complex` object *c* supports a single method:

`conjugate`	`c.conjugate()` Returns a new complex number `complex(c.real, -c.imag)` (i.e., the return value has `c`'s `imag` attribute with a sign change).

See **"The math and cmath Modules"** for several other functions that use floats and complex numbers.

Underscores in numeric literals

To aid with visual assessment of the magnitude of a number, numeric literals can include single underscore (_) characters between digits or after any base specifier. It's not only decimal numeric constants that can benefit from this notational freedom, however, as these examples show:

```
>>> 100_000.000_0001, 0x_FF_FF, 0o7_777, 0b_1010_1010
```

```
(100000.0000001, 65535, 4095, 170)
```

There is no enforcement of location of the underscores (except that two may not occur consecutively), so `123_456` and `12_34_56` both represent the same `int` value as `123456`.

Sequences

A *sequence* is an ordered container of items, indexed by integers. Python has built-in sequence types known as strings (`bytes` or `str`), tuples, and lists. Library and extension modules provide other sequence types, and you can write others yourself (as discussed in **"Sequences"**"). You can manipulate sequences in a variety of ways, as discussed in **"Sequence Operations"**.

Iterables

A Python concept that captures in abstract the iteration behavior of sequences is that of *iterables*, covered in **"The for Statement"**. All sequences are iterable: whenever we say you can use an iterable, you can use a sequence (for example, a list).

Also, when we say that you can use an iterable we usually mean a *bounded* iterable: an iterable that eventually stops yielding items. In general, sequences are bounded. Iterables can be unbounded, but if you try to use an unbounded iterable without special precautions, you could produce a program that never terminates, or one that exhausts all available memory.

Strings

Python has two built-in string types, `str` and `bytes`.[4] A `str` object is a sequence of characters used to store and represent text-based information. A `bytes` object stores and represents arbitrary sequences of binary bytes. Strings of both types in Python are *immutable*: when you perform an operation on strings, you always produce a new string object of the same type, rather than mutating an existing string. String objects provide many methods, as discussed in detail in **"Methods of String Objects"**.

A string literal can be quoted or triple-quoted. A quoted string is a sequence of zero or more characters within matching quotes, single (') or double ("). For example:

```
'This is a literal string'
"This is another string"
```

The two different kinds of quotes function identically; having both lets you include one kind of quote inside of a string specified with the other kind, with no need to escape quote characters with the backslash character (\):

```
'I\'m a Python fanatic'     # You can escape a quote
"I'm a Python fanatic"      # This way may be more readable
```

Many (but far from all) style guides that pronounce on the subject suggest that you use single quotes when the choice is otherwise indifferent. The popular code formatter `black` prefers double quotes; this choice is controversial enough to have been the main inspiration for a "fork," `blue`, whose main difference from `black` is to prefer single quotes instead, as most of this book's authors do.

To have a string literal span multiple physical lines, you can use a `\` as the last character of a line to indicate that the next line is a continuation:

```
'A not very long string \
that spans two lines'       # Comment not allowed on previous line
```

You can also embed a newline in the string to make it contain two lines rather than just one:

```
'A not very long string\n\
that prints on two lines'   # Comment not allowed on previous line
```

A better approach, however, is to use a triple-quoted string, enclosed by matching triplets of quote characters (`'''`, or better, as mandated by **PEP 8**, `"""`). In a triple-quoted string literal, line breaks in the literal remain as newline characters in the resulting string object:

```
"""An even bigger
string that spans
three lines"""              # Comments not allowed on previous lines
```

You can start a triple-quoted literal with an escaped newline, to avoid having the first line of the literal string's content at a different indentation level from the rest. For example:

```
the_text = """\
First line
Second line
"""      # The same as "First line\nSecond line\n" but more readable
```

The only character that cannot be part of a triple-quoted string literal is an unescaped backslash, while a single-quoted string literal cannot contain unescaped backslashes, nor line ends, nor the quote character that encloses it. The backslash character starts an *escape sequence*, which lets you introduce any character in either kind of string literal. See **Table 3-3** for a list of all of Python's string escape sequences.

Table 3-3. String escape sequences

Sequence	Meaning	ASCII / ISO code
`\` *<newline>*	Ignore end of line	None
`\\`	Backslash	`0x5c`
`\'`	Single quote	`0x27`
`\"`	Double quote	`0x22`
`\a`	Bell	`0x07`
`\b`	Backspace	`0x08`
`\f`	Form feed	`0x0c`

Sequence	Meaning	ASCII / ISO code
`\n`	Newline	`0x0a`
`\r`	Carriage return	`0x0d`
`\t`	Tab	`0x09`
`\v`	Vertical tab	`0x0b`
`\` *DDD*	Octal value *DDD*	As given
`\x` *XX*	Hexadecimal value *XX*	As given
`\N{`*name*`}`	Unicode character	As given
`\` *o*	Any other character *o*: a two-character string	`0x5c` + as given

A variant of a string literal is a *raw string literal*. The syntax is the same as for quoted or triple-quoted string literals, except that an `r` or `R` immediately precedes the leading quote. In raw string literals, escape sequences are not interpreted as in **Table 3-3**, but are literally copied into the string, including backslashes and newline characters. Raw string literal syntax is handy for strings that include many backslashes, especially regular expression patterns (see **"Pattern String Syntax"**) and Windows absolute filenames (which use backslashes as directory separators). A raw string literal cannot end with an odd number of backslashes: the last one would be taken as escaping the terminating quote.

RAW AND TRIPLE-QUOTED STRING LITERALS ARE NOT DIFFERENT TYPES

Raw and triple-quoted string literals are *not* types different from other strings; they are just alternative syntaxes for literals of the usual two string types, `bytes` and `str`.

In `str` literals, you can use `\u` followed by four hex digits, or `\U` followed by eight hex digits, to denote Unicode characters; you can also include the escape sequences listed in **Table 3-3**. `str` literals can also include Unicode characters using the escape sequence `\N{`*`name`*`}`, where *name* is a standard **Unicode name**. For example, `\N{Copyright Sign}` indicates a Unicode copyright sign character (©).

Formatted string literals (commonly called *f-strings*) let you inject formatted expressions into your string "literals," which are therefore no longer constant, but rather are subject to evaluation at execution time. The formatting process is described in **"String Formatting"**. From a purely syntactic point of view, these new literals can be regarded as just another kind of string literal.

Multiple string literals of any kind—quoted, triple-quoted, raw, bytes, formatted—can be adjacent, with optional whitespace in between (as long as you do not mix strings containing text and bytes). The compiler concatenates such adjacent string literals into a single string object. Writing a long string literal in this way lets you present it readably across multiple physical lines and gives you an opportunity to insert comments about parts of the string. For example:

```
marypop = ('supercali'       # '(' begins logical line,
           'fragilistic'     # indentation is ignored
           'expialidocious') # until closing ')'
```

The string assigned to `marypop` is a single word of 34 characters.

bytes objects

A `bytes` object is an ordered sequence of `ints` from `0` to `255`. `bytes` objects are usually encountered when reading data from or writing data to a binary source (e.g, a file, a socket, or a network resource).

A `bytes` object can be initialized from a list of `ints` or from a string of characters. A `bytes` literal has the same syntax as a `str` literal, prefixed with `'b'`:

```
b'abc'
bytes([97, 98, 99])           # Same as the previous line
rb'\ = solidus'               # A raw bytes literal, containing a '\'
```

To convert a `bytes` object to a `str`, use the `bytes.decode` method. To convert a `str` object to a `bytes` object, use the `str.encode` method, as described in detail in **Chapter 9**.

bytearray objects

A `bytearray` is a *mutable* ordered sequence of `ints` from `0` to `255`; like a `bytes` object, you can construct it from a sequence of `ints` or characters. In fact, apart from mutability, it is just like a `bytes` object. As they are mutable, `bytearray` objects support methods and operators that modify elements within the array of byte values:

```
ba = bytearray([97, 98, 99])  # Like bytes, can take a sequence of ints
ba[1] = 97                    # Unlike bytes, contents can be modified
print(ba.decode())            # Prints 'aac'
```

Chapter 9 has additional material on creating and working with `bytearray` objects.

Tuples

A *tuple* is an immutable ordered sequence of items. The items of a tuple are arbitrary objects and may be of different types. You can use mutable objects (such as lists) as tuple items, but best practice is generally to avoid doing so.

To denote a tuple, use a series of expressions (the items of the tuple) separated by commas (`,`);[5] if every item is a literal, the whole construct is a *tuple literal*. You may optionally place a redundant comma after the last item. You can group tuple items within parentheses, but the parentheses are necessary only where the commas would otherwise have another meaning (e.g., in function calls), or to denote empty or nested tuples. A tuple with exactly two items is also known as a *pair*. To create a tuple of one item, add a comma to the end of the expression. To denote an empty tuple, use an empty pair of parentheses. Here are some tuple literals, the second of which uses optional parentheses:

```
100, 200, 300        # Tuple with three items
(3.14,)              # Tuple with one item, needs trailing comma
()                   # Empty tuple (parentheses NOT optional)
```

You can also call the built-in type `tuple` to create a tuple. For example:

```
tuple('wow')
```

This builds a tuple equal to that denoted by the tuple literal:

```
('w', 'o', 'w')
```

`tuple()` without arguments creates and returns an empty tuple, like `()`. When *x* is iterable, `tuple(`*x*`)` returns a tuple whose items are the same as those in *x*.

Lists

A *list* is a mutable ordered sequence of items. The items of a list are arbitrary objects and may be of different types. To denote a list, use a series of expressions (the items of the list) separated by commas (`,`), within brackets (`[]`);[6] if every item is a literal, the whole construct is a *list literal*. You may optionally place a redundant comma after the last item. To denote an empty list, use an empty pair of brackets. Here are some examples of list literals:

```
[42, 3.14, 'hello']  # List with three items
[100]                # List with one item
[]                   # Empty list
```

You can also call the built-in type `list` to create a list. For example:

```
list('wow')
```

This builds a list equal to that denoted by the list literal:

```
['w', 'o', 'w']
```

`list()` without arguments creates and returns an empty list, like `[]`. When *x* is iterable, `list(`*x*`)` returns a list whose items are the same as those in *x*.

You can also build lists with list comprehensions, covered in **"List comprehensions"**.

Sets

Python has two built-in set types, `set` and `frozenset`, to represent arbitrarily ordered collections of unique items. Items in a set may be of differ-

ent types, but they must all be *hashable* (see `hash` in **Table 8-2**). Instances of type `set` are mutable, and thus not hashable; instances of type `frozenset` are immutable and hashable. You can't have a set whose items are sets, but you can have a set (or frozenset) whose items are frozensets. Sets and frozensets are *not* ordered.

To create a set, you can call the built-in type `set` with no argument (this means an empty set) or one argument that is iterable (this means a set whose items are those of the iterable). You can similarly build a frozenset by calling `frozenset`.

Alternatively, to denote a (nonfrozen, nonempty) set, use a series of expressions (the items of the set) separated by commas (`,`) within braces (`{}`);[7] if every item is a literal, the whole assembly is a *set literal*. You may optionally place a redundant comma after the last item. Here are some example sets (two literals, one not):

```
{42, 3.14, 'hello'}  # Literal for a set with three items
{100}                # Literal for a set with one item
set()                # Empty set - no literal for empty set
                     # {} is an empty dict!
```

You can also build nonfrozen sets with set comprehensions, as discussed in **"Set comprehensions"**.

Note that two sets or frozensets (or a set and a frozenset) may compare as equal, but since they are unordered, iterating over them can return their contents in differing order.

Dictionaries

A *mapping* is an arbitrary collection of objects indexed by nearly[8] arbitrary values called *keys*. Mappings are mutable and, like sets but unlike sequences, are *not* (necessarily) ordered.

Python provides a single built-in mapping type: the dictionary type `dict`. Library and extension modules provide other mapping types, and you can write others yourself (as discussed in **"Mappings"**). Keys in a dictionary may be of different types, but they must be *hashable* (see `hash` in **Table 8-2**). Values in a dictionary are arbitrary objects and may be of any type. An item in a dictionary is a key/value pair. You can think of a dictionary as an associative array (known in some other languages as a "map," "hash table," or "hash").

To denote a dictionary, you can use a series of colon-separated pairs of expressions (the pairs are the items of the dictionary) separated by commas (`,`) within braces (`{}`);[9] if every expression is a literal, the whole construct is a *dictionary literal*. You may optionally place a redundant comma after the last item. Each item in a dictionary is written as *`key`*`:`*`value`*, where *`key`* is an expression giving the item's key and *`value`* is an expression giving the item's value. If a key's value appears more than once in a dictionary expression, only an arbitrary one of the items with that key is kept in the resulting dictionary object—dictionaries do not support duplicate keys. For example:

```
{1:2, 3:4, 1:5}  # The value of this dictionary is {1:5, 3:4}
```

To denote an empty dictionary, use an empty pair of braces.

Here are some dictionary literals:

```
{'x':42, 'y':3.14, 'z':7}    # Dictionary with three items, str keys
{1:2, 3:4}                   # Dictionary with two items, int keys
{1:'za', 'br':23}            # Dictionary with different key types
{}                           # Empty dictionary
```

You can also call the built-in type `dict` to create a dictionary in a way that, while less concise, can sometimes be more readable. For example, the `dicts` in the preceding snippet can also be written as:

```
dict(x=42, y=3.14, z=7)       # Dictionary with three items, str keys
dict([(1, 2), (3, 4)])        # Dictionary with two items, int keys
dict([(1,'za'), ('br',23)])   # Dictionary with different key types
dict()                        # Empty dictionary
```

`dict()` without arguments creates and returns an empty dictionary, like `{}`. When the argument *x* to `dict` is a mapping, `dict` returns a new dictionary object with the same keys and values as *x*. When *x* is iterable, the items in *x* must be pairs, and `dict(`*x*`)` returns a dictionary whose items (key/value pairs) are the same as the items in *x*. If a key value appears more than once in *x*, only the *last* item from *x* with that key value is kept in the resulting dictionary.

When you call `dict` in addition to or instead of the positional argument *x*, you may pass *named arguments*, each with the syntax *name=value*, where *name* is an identifier to use as an item's key and *value* is an expression giving the item's value. When you call `dict` and pass both a positional argument and one or more named arguments, if a key appears both in the positional argument and as a named argument, Python associates to that key the named argument's value (i.e., the named argument "wins").

You can unpack a dict's contents into another dict using the ** operator.

```
d1 = {'a':1, 'x': 0}
d2 = {'c': 2, 'x': 5}
d3 = {**d1, **d2}  # result is {'a':1, 'x': 5, 'c': 2}
```

3.9+ As of Python 3.9, this same operation can be performed using the | operator.

```
d4 = d1 | d2  # same result as d3
```

You can also create a dictionary by calling `dict.fromkeys`. The first argument is an iterable whose items become the keys of the dictionary; the second argument is the value that corresponds to each and every key (all keys initially map to the same value). If you omit the second argument, it defaults to **`None`**. For example:

```
dict.fromkeys('hello', 2)    # Same as {'h':2, 'e':2, 'l':2, 'o':2}
dict.fromkeys([1, 2, 3])     # Same as {1:None, 2:None, 3:None}
```

You can also build a `dict` using a dictionary comprehension, as discussed in **"Dictionary comprehensions"**.

When comparing two `dicts` for equality, they will evaluate as equal if they have the same keys and corresponding values, even if the keys are not in the same order.

None

The built-in **`None`** denotes a null object. **`None`** has no methods or other attributes. You can use **`None`** as a placeholder when you need a reference but you don't care what object you refer to, or when you need to indicate that no object is there. Functions return **`None`** as their result unless they have specific **`return`** statements coded to return other values. **`None`** is hashable and can be used as a `dict` key.

Ellipsis (...)

The `Ellipsis`, written as three periods with no intervening spaces, `...`, is a special object in Python used in numerical applications,[10] or as an alternative to **`None`** when **`None`** is a valid entry. For instance, to initialize a `dict` that may take **`None`** as a legitimate value, you can initialize it with `...` as an indicator of "no value supplied, not even **`None`**." `Ellipsis` is hashable and so can be used as a `dict` key:

```
tally = dict.fromkeys(['A', 'B', None, ...], 0)
```

Callables

In Python, callable types are those whose instances support the function call operation (see **"Calling Functions"**). Functions are callable. Python provides numerous built-in functions (see **"Built-in Functions"**) and supports user-defined functions (see **"Defining Functions: The def Statement"**). Generators are also callable (see **"Generators"**).

Types are callable too, as we saw for the `dict`, `list`, `set`, and `tuple` built-in types. (See **"Built-in Types"** for a complete list of built-in types.) As we discuss in **"Python Classes"**, **`class`** objects (user-defined types) are also callable. Calling a type usually creates and returns a new instance of that type.

Other callables include *methods*, which are functions bound as class attributes, and instances of classes that supply a special method named `__call__`.

Boolean Values

Any[11] data value in Python can be used as a truth value: true or false. Any nonzero number or nonempty container (e.g., string, tuple, list, set, or dictionary) is true. Zero (`0`, of any numeric type), **`None`**, and empty containers are false. You may see the terms "truthy" and "falsy" used to indicate values that evaluate as either true or false.

BEWARE USING A FLOAT AS A TRUTH VALUE

Be careful about using a floating-point number as a truth value: that's like comparing the number for exact equality with zero, and floating-point numbers should almost never be compared for exact equality.

The built-in type `bool` is a subclass of `int`. The only two values of type `bool` are **`True`** and **`False`**, which have string representations of **`'True'`** and **`'False'`**, but also numerical values of `1` and `0`, respectively. Several built-in functions return `bool` results, as do comparison operators.

You can call `bool(`*`x`*`)` with any[1][2] *x* as the argument. The result is **`True`** when *x* is true and **`False`** when *x* is false. Good Python style is not to use such calls when they are redundant, as they most often are: *always* write **`if`** *x*`:`, *never* any of **`if`** `bool(`*x*`):`, **`if`** *x* **`is True`**`:`, **`if`** *x* `==` **`True`**`:`, or **`if`** `bool(`*x*`) ==` **`True`**`:`. However, you *can* use `bool(`*x*`)` to count the number of true items in a sequence. For example:

```
def count_trues(seq):
    return sum(bool(x) for x in seq)
```

In this example, the `bool` call ensures each item of *`seq`* is counted as `0` (if false) or `1` (if true), so `count_trues` is more general than `sum(`*`seq`*`)` would be.

When we say "*`expression`* is true" we mean that `bool(`*`expression`*`)` would return **`True`**. As we mentioned, this is also known as "*`expression`* being *truthy*" (the other possibility is that "*`expression`* is *falsy*").

Variables and Other References

A Python program accesses data values through *references*. A reference is a "name" that refers to a value (object). References take the form of variables, attributes, and items. In Python, a variable or other reference has no intrinsic type. The object to which a reference is bound at a given time always has a type, but a given reference may be bound to objects of various types in the course of the program's execution.

Variables

In Python, there are no "declarations." The existence of a variable begins with a statement that *binds* the variable (in other words, sets a name to hold a reference to some object). You can also *unbind* a variable, resetting the name so it no longer holds a reference. Assignment statements are the usual way to bind variables and other references. The `del` statement unbinds a variable reference, although doing so is rare.

Binding a reference that was already bound is also known as *rebinding* it. Whenever we mention binding, we implicitly include rebinding (except where we explicitly exclude it). Rebinding or unbinding a reference has no effect on the object to which the reference was bound, except that an object goes away when nothing refers to it. The cleanup of objects with no references is known as *garbage collection*.

You can name a variable with any identifier except the 30-plus that are reserved as Python's keywords (see **"Keywords"**). A variable can be global or local. A *global variable* is an attribute of a module object (see **Chapter 7**). A *local variable* lives in a function's local namespace (see **"Namespaces"**).

Object attributes and items

The main distinction between the attributes and items of an object is in the syntax you use to access them. To denote an *attribute* of an object, use a reference to the object, followed by a period (`.`), followed by an identifier known as the *attribute name*. For example, *x*`.`*y* refers to one of the attributes of the object bound to name *x*; specifically, that attribute whose name is `'`*y*`'`.

To denote an *item* of an object, use a reference to the object, followed by an expression within brackets []. The expression in brackets is known as the item's *index* or *key*, and the object is known as the item's *container*. For example, *x*[*y*] refers to the item at the key or index bound to name *y*, within the container object bound to name *x*.

Attributes that are callable are also known as *methods*. Python draws no strong distinctions between callable and noncallable attributes, as some other languages do. All general rules about attributes also apply to callable attributes (methods).

Accessing nonexistent references

A common programming error is to access a reference that does not exist. For example, a variable may be unbound, or an attribute name or item index may not be valid for the object to which you apply it. The Python compiler, when it analyzes and compiles source code, diagnoses only syntax errors. Compilation does not diagnose semantic errors, such as trying to access an unbound attribute, item, or variable. Python diagnoses semantic errors only when the errant code executes—that is, *at runtime*. When an operation is a Python semantic error, attempting it raises an exception (see **Chapter 6**). Accessing a nonexistent variable, attribute, or item—just like any other semantic error—raises an exception.

Assignment Statements

Assignment statements can be plain or augmented. Plain assignment to a variable (e.g., `name = value`) is how you create a new variable or rebind an existing variable to a new value. Plain assignment to an object attribute (e.g., `x.attr = value`) is a request to object `x` to create or rebind the attribute named `'attr'`. Plain assignment to an item in a container (e.g., `x[k] = value`) is a request to container `x` to create or rebind the item with index or key `k`.

Augmented assignment (e.g., `name += value`) cannot, per se, create new references. Augmented assignment can rebind a variable, ask an object to rebind one of its existing attributes or items, or request the target object to modify itself. When you make any kind of request to an object, it is up to the object to decide whether and how to honor the request, and whether to raise an exception.

Plain assignment

A plain assignment statement in the simplest form has the syntax:

```
target = expression
```

The target is known as the lefthand side (LHS), and the expression is the righthand side (RHS). When the assignment executes, Python evaluates the RHS expression, then binds the expression's value to the LHS target. The binding never depends on the type of the value. In particular, Python draws no strong distinction between callable and noncallable objects, as some other languages do, so you can bind functions, methods, types, and other callables to variables, just as you can numbers, strings, lists, and so on. This is part of functions and other callables being *first-class objects*.

Details of the binding do depend on the kind of target. The target in an assignment may be an identifier, an attribute reference, an indexing, or a slicing, where:

An identifier
: Is a variable name. Assigning to an identifier binds the variable with this name.

An attribute reference
: Has the syntax `obj.name`. `obj` is an arbitrary expression, and `name` is an identifier, known as an *attribute name* of the object. Assigning to an attribute reference asks the object `obj` to bind its attribute named `'name'`.

An indexing
: Has the syntax `obj[expr]`. `obj` and `expr` are arbitrary expressions. Assigning to an indexing asks the container `obj` to bind its item indicated by the value of `expr`, also known as the index or key of the item in the container (an *indexing* is an index *applied to* a container).

A slicing
: Has the syntax `obj[start:stop]` or `obj[start:stop:stride]`. `obj`, `start`, `stop`, and `stride` are arbitrary expressions. `start`, `stop`, and `stride` are all optional (i.e., `obj[:stop:]` and `obj[:stop]` are also syntactically correct slicings, each being equivalent to `obj[None:stop:None]`). Assigning to a slicing asks the container `obj` to bind or unbind some of its items. Assigning to a slicing such as `obj[start:stop:stride]` is equivalent to assigning to the indexing

obj`[slice(`*start*`,` *stop*`,` *stride*`)]`. See Python's built-in type `slice` in (**Table 8-1**), whose instances represent slices (a *slicing* is a slice *applied to* a container).

We'll get back to indexing and slicing targets when we discuss operations on lists in **"Modifying a list"**, and on dictionaries in **"Indexing a Dictionary"**.

When the target of the assignment is an identifier, the assignment statement specifies the binding of a variable. This is *never* disallowed: when you request it, it takes place. In all other cases, the assignment statement denotes a request to an object to bind one or more of its attributes or items. An object may refuse to create or rebind some (or all) attributes or items, raising an exception if you attempt a disallowed creation or rebinding (see also `__setattr__` in **Table 4-1** and `__setitem__` in **"Container methods"**).

A plain assignment can use multiple targets and equals signs (`=`). For example:

```
a = b = c = 0
```

binds variables `a`, `b`, and `c` to the same value, `0`. Each time the statement executes, the RHS expression evaluates just once, no matter how many targets are in the statement. Each target, left to right, is bound to the one object returned by the expression, just as if several simple assignments executed one after the other.

The target in a plain assignment can list two or more references separated by commas, optionally enclosed in parentheses or brackets. For example:

```
a, b, c = x
```

This statement requires `x` to be an iterable with exactly three items, and binds `a` to the first item, `b` to the second, and `c` to the third. This kind of assignment is known as an *unpacking assignment*. The RHS expression must be an iterable with exactly as many items as there are references in the target; otherwise, Python raises an exception. Python binds each reference in the target to the corresponding item in the RHS. You can use an unpacking assignment, for example, to swap references:

```
a, b = b, a
```

This assignment statement rebinds name `a` to what name `b` was bound to, and vice versa. Exactly one of the multiple targets of an unpacking assignment may be preceded by `*`. That *starred* target, if present, is bound to a list of all items, if any, that were not assigned to other targets. For example, when `x` is a list, this:

```
first, *middle, last = x
```

is the same as (but more concise, clearer, more general, and faster than) this:

```
first, middle, last = x[0], x[1:-1], x[-1]
```

Each of these forms requires `x` to have at least two items. This feature is known as *extended unpacking*.

Augmented assignment

An *augmented assignment* (sometimes called an *in-place assignment*) differs from a plain assignment in that, instead of an equals sign (`=`) between the target and the expression, it uses an *augmented operator*, which is a binary operator followed by `=`. The augmented operators are `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `|=`, `>>=`, `<<=`, `&=`, `^=`, and `@=`. An augmented assignment can

have only one target on the LHS; augmented assignment does not support multiple targets.

In an augmented assignment, like in a plain one, Python first evaluates the RHS expression. Then, when the LHS refers to an object that has a special method for the appropriate *in-place* version of the operator, Python calls the method with the RHS value as its argument (it is up to the method to modify the LHS object appropriately and return the modified object; **"Special Methods"** covers special methods). When the LHS object has no applicable in-place special method, Python uses the corresponding binary operator on the LHS and RHS objects, then rebinds the target to the result. For example, *x* += *y* is like *x* = *x*.__iadd_(*y*) when *x* has the special method `__iadd__` for "in-place addition"; otherwise, *x* += *y* is like *x* = *x* + *y*.

Augmented assignment never creates its target reference; the target must already be bound when augmented assignment executes. Augmented assignment can rebind the target reference to a new object, or modify the same object to which the target reference was already bound. Plain assignment, in contrast, can create or rebind the LHS target reference, but it never modifies the object, if any, to which the target reference was previously bound. The distinction between objects and references to objects is crucial here. For example, *x* = *x* + *y* never modifies the object to which *x* was originally bound, if any. Rather, it rebinds *x* to refer to a new object. *x* += *y*, in contrast, modifies the object to which the name *x* is bound, when that object has the special method `__iadd__`; otherwise, *x* += *y* rebinds *x* to a new object, just like *x* = *x* + *y*.

del Statements

Despite its name, a **`del`** statement *unbinds references*—it does *not*, per se, *delete* objects. Object deletion may automatically follow, by garbage collection, when no more references to an object exist.

A **`del`** statement consists of the keyword **`del`**, followed by one or more target references separated by commas (`,`). Each target can be a variable, attribute reference, indexing, or slicing, just like for assignment statements,

and must be bound at the time **`del`** executes. When a **`del`** target is an identifier, the **`del`** statement means to unbind the variable. If the identifier was bound, unbinding it is never disallowed; when requested, it takes place.

In all other cases, the **`del`** statement specifies a request to an object to unbind one or more of its attributes or items. An object may refuse to unbind some (or all) attributes or items, raising an exception if you attempt a disallowed unbinding (see also `__delattr__` in **"General-Purpose Special Methods"** and `__delitem__` in **"Container methods"**). Unbinding a slicing normally has the same effect as assigning an empty sequence to that slicing, but it is up to the container object to implement this equivalence.

Containers are also allowed to have **`del`** cause side effects. For example, assuming **`del`** *`C`*`[2]` succeeds, when *`C`* is a dictionary, this makes future references to *`C`*`[2]` invalid (raising `KeyError`) until and unless you assign to *`C`*`[2]` again; but when *`C`* is a list, **`del`** *`C`*`[2]` implies that every following item of *`C`* "shifts left by one"—so, if *`C`* is long enough, future references to *`C`*`[2]` are still valid, but denote a different item than they did before the **`del`** (generally, what you'd have used *`C`*`[3]` to refer to, before the **`del`** statement).

Expressions and Operators

An expression is a "phrase" of code, which Python evaluates to produce a value. The simplest expressions are literals and identifiers. You build other expressions by joining subexpressions with the operators and/or delimiters listed in **Table 3-4**. This table lists operators in decreasing order of precedence, higher precedence before lower. Operators listed together have the same precedence. The third column lists the associativity of the operator: L (left-to-right), R (right-to-left), or NA (nonassociative).

Table 3-4. Operator precedence in expressions

Operator	Description	Associativity

Operator	Description	Associativity
`{ key : expr, ... }`	Dictionary creation	NA
`{ expr, ... }`	Set creation	NA
`[ expr, ... ]`	List creation	NA
`( expr, ... )`	Tuple creation (parentheses recommended, but not always required; at least one comma required), or just parentheses	NA
`f ( expr, ... )`	Function call	L
`x [ index: index: step ]`	Slicing	L
`x [ index ]`	Indexing	L
`x . attr`	Attribute reference	L
`x ** y`	Exponentiation (*x* to the *y*th power)	R
`~ x`, `+ x`, `- x`	Bitwise NOT, unary plus and minus	NA
`x * y`, `x @ y`, `x / y`, `x // y`, `x % y`	Multiplication, matrix multiplication, division, floor division, remainder	L

Operator	Description	Associativity
x + *y*, *x* - *y*	Addition, subtraction	L
x << *y*, *x* >> *y*	Left-shift, right-shift	L
x & *y*	Bitwise AND	L
x ^ *y*	Bitwise XOR	L
x | *y*	Bitwise OR	L
x < *y*, *x* <= *y*, *x* > *y*, *x* >= *y*, *x* != *y*, *x* == *y*	Comparisons (less than, less than or equal, greater than, greater than or equal, inequality, equality)	NA
x **is** *y*, *x* **is not** *y*	Identity tests	NA
x **in** *y*, *x* **not in** *y*	Membership tests	NA
not *x*	Boolean NOT	NA
x **and** *y*	Boolean AND	L
x **or** *y*	Boolean OR	L
x **if** *expr* **else** *y*	Conditional expression (or ternary operator)	NA
lambda *arg*, ...: *expr*	Anonymous simple function	NA

Operator	Description	Associativity
`( ident := expr )`	Assignment expression (parentheses recommended, but not always required)	NA

In this table, *expr*, *key*, *f*, *index*, *x*, and *y* mean any expression, while *attr*, *arg*, and *ident* mean any identifier. The notation `, ...` means commas join zero or more repetitions; in such cases, a trailing comma is optional and innocuous.

Comparison Chaining

You can chain comparisons, implying a logical **and**. For example:

```
a < b <= c < d
```

where *a*, *b*, *c*, and *d* are arbitrary expressions, has (in the absence of evaluation side effects) the same value as:

```
a < b and b <= c and c < d
```

The chained form is more readable, and evaluates each subexpression at most once.

Short-Circuiting Operators

The **and** and **or** operators *short-circuit* their operands' evaluation: the righthand operand evaluates only when its value is needed to get the truth value of the entire **and** or **or** operation.

In other words, *x* **and** *y* first evaluates *x*. When *x* is false, the result is *x*; otherwise, the result is *y*. Similarly, *x* **or** *y* first evaluates *x*. When *x* is true,

the result is *x*; otherwise, the result is *y*.

`and` and **`or`** don't force their results to be **`True`** or **`False`**, but rather return one or the other of their operands. This lets you use these operators more generally, not just in Boolean contexts. **`and`** and **`or`**, because of their short-circuiting semantics, differ from other operators, which fully evaluate all operands before performing the operation. **`and`** and **`or`** let the left operand act as a *guard* for the right operand.

The conditional operator

Another short-circuiting operator is the conditional[13] operator **`if`**/**`else`**:

```
when_true if condition else when_false
```

Each of *`when_true`*, *`when_false`*, and *`condition`* is an arbitrary expression. *`condition`* evaluates first. When *`condition`* is true, the result is *`when_true`*; otherwise, the result is *`when_false`*. Only one of the subexpressions *`when_true`* and *`when_false`* evaluates, depending on the truth value of *`condition`*.

The order of the subexpressions in this conditional operator may be a bit confusing. The recommended style is to always place parentheses around the whole expression.

Assignment Expressions

3.8+ You can combine evaluation of an expression and the assignment of its result using the `:=` operator. There are several common cases where this is useful.

:= in an if/elif statement

Code that assigns a value and then checks it can be collapsed using `:=`:

```
re_match = re.match(r'Name: (\S)', input_string)
if re_match:
    print(re_match.groups(1))

# collapsed version using :=
if (re_match := re.match(r'Name: (\S)', input_string)):
    print(re_match.groups(1))
```

This is especially helpful when writing a sequence of **`if/elif`** blocks (you'll find a more extended example in **Chapter 10**).

:= in a while statement

Use `:=` to simplify code that uses a variable as its **`while`** condition. Consider this code that works with a sequence of values returned by some function `get_next_value`, which returns **`None`** when there are no more values to process:

```
current_value = get_next_value()
while current_value is not None:
    if not filter_condition(current_value):
        continue   # BUG! Current_value is not advanced to next
    # ... do some work with current_value ...
    current_value = get_next_value()
```

This code has a couple of problems. First, there is the duplicated call to `get_next_value`, which carries extra maintenance costs when `get_next_value` changes. But more seriously, there is a bug when an early exiting filter is added: the **`continue`** statement jumps directly back to the **`while`** statement without advancing to the next value, creating an infinite loop.

When we use `:=` to incorporate the assignment into the **`while`** statement itself, we fix the duplication problem, and calling **`continue`** does not cause an infinite loop:

```
while (current_value := get_next_value()) is not None:
    if not filter_condition(current_value):
        continue   # no bug, current_value advances in while statement
    # ... do some work with current_value ...
```

:= in a list comprehension filter

A list comprehension that converts an input item but must filter out some items based on their converted values can use `:=` to do the conversion only once. In this example, a function to convert `str`s to `int`s returns **`None`** for invalid values. Without `:=`, the list comprehension must call `safe_int` twice for valid values, once to check for **`None`** and then again to add the actual `int` value to the list:

```
def safe_int(s):
    try:
        return int(s)
    except Exception:
        return None

input_strings = ['1','2','a','11']

valid_int_strings = [safe_int(s) for s in input_strings
                     if safe_int(s) is not None]
```

If we use an assignment expression in the condition part of the list comprehension, `safe_int` only gets called once for each value in `input_strings`:

```
valid_int_strings = [int_s for s in input_strings
                     if (int_s := safe_int(s)) is not None]
```

You can find more examples in the original PEP for this feature, **PEP 572**.

Numeric Operations

Python offers the usual numeric operations, as we've just seen in **Table 3-4**. Numbers are immutable objects: when you perform operations on number objects, you always produce new objects and never modify existing ones. You can access the parts of a complex object *z* as read-only attributes `z.real` and `z.imag`. Trying to rebind these attributes raises an exception.

A number's optional `+` or `-` sign, and the `+` or `-` that joins a floating-point literal to an imaginary one to make a complex number, are not part of the literals' syntax. They are ordinary operators, subject to normal operator precedence rules (see **Table 3-4**). For example, `-2 ** 2` evaluates to `-4`: exponentiation has higher precedence than unary minus, so the whole expression parses as `-(2 ** 2)`, not as `(-2) ** 2`. (Again, parentheses are recommended, to avoid confusing a reader of the code.)

Numeric Conversions

You can perform arithmetic operations and comparisons between any two numbers of Python built-in types (integers, floating-point numbers, and complex numbers). If the operands' types differ, Python converts the operand with the "narrower" type to the "wider" type.[14] The built-in numeric types, in order from narrowest to widest, are `int`, `float`, and `complex`. You can request an explicit conversion by passing a noncomplex numeric argument to any of these types. `int` drops its argument's fractional part, if any (e.g., `int(9.8)` is `9`). You can also call `complex` with two numeric arguments, giving real and imaginary parts. You cannot convert a `complex` to another numeric type in this way, because there is no single unambiguous way to convert a complex number into, for example, a `float`.

You can also call each built-in numeric type with a string argument with the syntax of an appropriate numeric literal, with small extensions: the argument string may have leading and/or trailing whitespace, may start with a sign, and—for complex numbers—may sum or subtract a real part

and an imaginary one. `int` can also be called with two arguments: the first one a string to convert, and the second the *radix*, an integer between 2 and 36 to use as the base for the conversion (e.g., `int('101', 2)` returns `5`, the value of `'101'` in base 2). For radices larger than 10, the appropriate subset of ASCII letters from the start of the alphabet (in either lower- or uppercase) are the extra needed "digits."[15]

Arithmetic Operations

Arithmetic operations in Python behave in rather obvious ways, with the possible exception of division and exponentiation.

Division

When the right operand of `/`, `//`, or `%` is `0`, Python raises an exception at runtime. Otherwise, the `/` operator performs *true* division, returning the floating-point result of division of the two operands (or a complex result if either operand is a complex number). In contrast, the `//` operator performs *floor* division, which means it returns an integer result (converted to the same type as the wider operand) that's the largest integer less than or equal to the true division result (ignoring the remainder, if any); e.g., `5.0 // 2 = 2.0` (not `2`). The `%` operator returns the remainder of the (floor) division, i.e., the integer such that `(x // y) * y + (x % y) == x`.

–X // Y IS NOT THE SAME AS INT(–X / Y)

Take care not to think of `//` as a truncating or integer form of division; this is only the case for operands of the same sign. When operands are of different signs, the largest integer less than or equal to the true division result will actually be a more negative value than the result from true division (for example, `-5 / 2` returns `-2.5`, so `-5 // 2` returns `-3`, not `-2`).

The built-in `divmod` function takes two numeric arguments and returns a pair whose items are the quotient and remainder, so you don't have to use both `//` for the quotient and `%` for the remainder.[16]

Exponentiation

The exponentiation ("raise to power") operation, when *a* is less than zero and *b* is a floating-point value with a nonzero fractional part, returns a complex number. The built-in `pow(a, b)` function returns the same result as *a* ** *b*. With three arguments, `pow(a, b, c)` returns the same result as `(a ** b) % c` but may sometimes be faster. Note that, unlike other arithmetic operations, exponentiation evaluates right to left: in other words, *a* ** *b* ** *c* evaluates as *a* ** (*b* ** *c*).

Comparisons

All objects, including numbers, can be compared for equality (==) and inequality (!=). Comparisons requiring order (<, <=, >, >=) may be used between any two numbers unless either operand is complex, in which case they raise exceptions at runtime. All these operators return Boolean values (**`True`** or **`False`**). Be careful when comparing floating-point numbers for equality, however, as discussed in **Chapter 16** and the **online tutorial on floating-point arithmetic**.

Bitwise Operations on Integers

`int`s can be interpreted as strings of bits and used with the bitwise operations shown in **Table 3-4**. Bitwise operators have lower priority than arithmetic operators. Positive `int`s are conceptually extended by an unbounded string of bits on the left, each bit being `0`. Negative `int`s, as they're held in two's complement representation, are conceptually extended by an unbounded string of bits on the left, each bit being `1`.

Sequence Operations

Python supports a variety of operations applicable to all sequences, including strings, lists, and tuples. Some sequence operations apply to all containers (including sets and dictionaries, which are not sequences); some apply to all iterables (meaning "any object over which you can loop"—all containers, be they sequences or not, are iterable, and so are

many objects that are not containers, such as files, covered in **"The io Module"**, and generators, covered in **"Generators"**). In the following we use the terms *sequence*, *container*, and *iterable* quite precisely, to indicate exactly which operations apply to each category.

Sequences in General

Sequences are ordered containers with items that are accessible by indexing and slicing.

The built-in `len` function takes any container as an argument and returns the number of items in the container.

The built-in `min` and `max` functions take one argument, an iterable whose items are comparable, and return the smallest and largest items, respectively. You can also call `min` and `max` with multiple arguments, in which case they return the smallest and largest arguments, respectively.

`min` and `max` also accept two keyword-only optional arguments: `key`, a callable to apply to each item (the comparisons are then performed on the callable's results rather than on the items themselves); and `default`, the value to return when the iterable is empty (when the iterable is empty and you supply no default argument, the function raises `ValueError`). For example, `max('who', 'why', 'what', key=len)` returns `'what'`.

The built-in `sum` function takes one argument, an iterable whose items are numbers, and returns the sum of the numbers.

Sequence conversions

There is no implicit conversion between different sequence types. You can call the built-ins `tuple` and `list` with a single argument (any iterable) to get a new instance of the type you're calling, with the same items, in the same order, as in the argument.

Concatenation and repetition

You can concatenate sequences of the same type with the + operator. You can multiply a sequence *S* by an integer *n* with the * operator. *S***n* is the concatenation of *n* copies of *S*. When *n* <= `0`, *S* * *n* is an empty sequence of the same type as *S*.

Membership testing

The *x* **`in`** *S* operator tests to check whether the object *x* equals any item in the sequence (or other kind of container or iterable) *S*. It returns **`True`** when it does and **`False`** when it doesn't. The *x* **`not in`** *S* operator is equivalent to **`not`** (*x* **`in`** *S*). For dictionaries, *x* **`in`** *S* tests for the presence of *x* as a key. In the specific case of strings, *x* **`in`** *S* may match more than expected; in this case, *x* **`in`** *S* tests whether *x* equals any *substring* of *S*, not just any single character.

Indexing a sequence

To denote the *n*th item of a sequence *S*, use indexing: *S*`[`*n*`]`. Indexing is zero-based: *S*'s first item is *S*`[0]`. If *S* has *L* items, the index *n* may be `0`, `1`... up to and including *L*`-1`, but no larger. *n* may also be `-1`, `-2`... down to and including `-`*L*, but no smaller. A negative *n* (e.g., `-1`) denotes the same item in *S* as *L*`+`*n* (e.g., *L*`-1`) does. In other words, *S*`[-1]`, like *S*`[`*L*`-1]`, is the last element of *S*, *S*`[-2]` is the next-to-last one, and so on. For example:

```
x = [10, 20, 30, 40]
x[1]                    # 20
x[-1]                   # 40
```

Using an index >=*L* or <`-`*L* raises an exception. Assigning to an item with an invalid index also raises an exception. You can add elements to a list, but to do so you assign to a slice, not to an item, as we'll discuss shortly.

Slicing a sequence

To indicate a subsequence of `S`, you can use slicing, with the syntax `S[i:j]`, where `i` and `j` are integers. `S[i:j]` is the subsequence of `S` from the `i`th item, included, to the `j`th item, excluded (in Python, ranges always include the lower bound and exclude the upper bound). A slice is an empty subsequence when `j` is less than or equal to `i`, or when `i` is greater than or equal to `L`, the length of `S`. You can omit `i` when it is equal to `0`, so that the slice begins from the start of `S`. You can omit `j` when it is greater than or equal to `L`, so that the slice extends all the way to the end of `S`. You can even omit both indices, to mean a shallow copy of the entire sequence: `S[:]`. Either or both indices may be less than zero. Here are some examples:

```
x = [10, 20, 30, 40]
x[1:3]                  # [20, 30]
x[1:]                   # [20, 30, 40]
x[:2]                   # [10, 20]
```

A negative index *n* in slicing indicates the same spot in `S` as `L+n`, just like it does in indexing. An index greater than or equal to `L` means the end of `S`, while a negative index less than or equal to `-L` means the start of `S`.

Slicing can use the extended syntax `S[i:j:k]`. `k` is the *stride* of the slice, meaning the distance between successive indices. `S[i:j]` is equivalent to `S[i:j:1]`, `S[::2]` is the subsequence of `S` that includes all items that have an even index in `S`, and `S[::-1]` is a slicing, also whimsically known as "the Martian smiley," with the same items as `S` but in reverse order. With a negative stride, in order to have a nonempty slice, the second ("stop") index needs to be *smaller* than the first ("start") one—the reverse of the condition that must hold when the stride is positive. A stride of `0` raises an exception. Here are some examples:

```
>>> y = list(range(10))  # values from 0-9
```

```
>>> y[-5:]              # last five items
```

```
[5, 6, 7, 8, 9]
```

```
>>> y[::2]              # every other item
```

```
[0, 2, 4, 6, 8]
```

```
>>> y[10:0:-2]          # every other item, in reverse order
```

```
[9, 7, 5, 3, 1]
```

```
>>> y[:0:-2]            # every other item, in reverse order (simpler)
```

```
[9, 7, 5, 3, 1]
```

```
>>> y[::-2]             # every other item, in reverse order (best)
```

```
[9, 7, 5, 3, 1]
```

Strings

String objects (both `str` and `bytes`) are immutable: attempting to rebind or delete an item or slice of a string raises an exception. (Python also has

a built-in type that is mutable but otherwise equivalent to `bytes`: `bytearray` (see **"bytearray objects"**). The items of a text string (each of the characters in the string) are themselves text strings, each of length 1—Python has no special data type for "single characters" (the items of a `bytes` or `bytearray` object are `int`s). All slices of a string are strings of the same kind. String objects have many methods, covered in **"Methods of String Objects"**.

Tuples

Tuple objects are immutable: therefore, attempting to rebind or delete an item or slice of a tuple raises an exception. The items of a tuple are arbitrary objects and may be of different types; tuple items may be mutable, but we recommend not mutating them, as doing so can be confusing. The slices of a tuple are also tuples. Tuples have no normal (nonspecial) methods, except `count` and `index`, with the same meanings as for lists; they do have many of the special methods covered in **"Special Methods"**.

Lists

List objects are mutable: you may rebind or delete items and slices of a list. Items of a list are arbitrary objects and may be of different types. Slices of a list are lists.

Modifying a list

You can modify (rebind) a single item in a list by assigning to an indexing. For instance:

```
x = [1, 2, 3, 4]
x[1] = 42                # x is now [1, 42, 3, 4]
```

Another way to modify a list object *L* is to use a slice of *L* as the target (LHS) of an assignment statement. The RHS of the assignment must be an iterable. When the LHS slice is in extended form (i.e., the slicing specifies

a stride other than 1), then the RHS must have just as many items as the number of items in the LHS slice. When the LHS slicing does not specify a stride, or explicitly specifies a stride of 1, the LHS slice and the RHS may each be of any length; assigning to such a slice of a list can make the list longer or shorter. For example:

```
x = [10, 20, 30, 40, 50]
# replace items 1 and 2
x[1:3] = [22, 33, 44]    # x is now [10, 22, 33, 44, 40, 50]
# replace items 1-3
x[1:4] = [88, 99]        # x is now [10, 88, 99, 40, 50]
```

There are some important special cases of assignment to slices:

- Using the empty list `[]` as the RHS expression removes the target slice from *L*. In other words, `L[i:j] = []` has the same effect as **`del`** `L[i:j]` (or the peculiar statement `L[i:j] *= 0`).
- Using an empty slice of *L* as the LHS target inserts the items of the RHS at the appropriate spot in *L*. For example, `L[i:i] = ['a', 'b']` inserts `'a'` and `'b'` before the item that was at index *i* in *L* prior to the assignment.
- Using a slice that covers the entire list object, `L[:]`, as the LHS target totally replaces the contents of *L*.

You can delete an item or a slice from a list with **`del`**. For instance:

```
x = [1, 2, 3, 4, 5]
del x[1]                 # x is now [1, 3, 4, 5]
del x[::2]               # x is now [3, 5]
```

In-place operations on a list

List objects define in-place versions of the + and * operators, which you can use via augmented assignment statements. The augmented assignment statement *L* += *L1* has the effect of adding the items of the iterable

L1 to the end of *L*, just like `L.extend(L1)`. `L *= n` has the effect of adding *n*-1 copies of *L* to the end of *L*; if *n* <= 0, `L *= n` makes *L* empty, like `L[:] = []` or **`del`** `L[:]`.

List methods

List objects provide several methods, as shown in **Table 3-5**. Nonmutating methods return a result without altering the object to which they apply, while mutating methods may alter the object to which they apply. Many of a list's mutating methods behave like assignments to appropriate slices of the list. In this table, *L* indicates any list object, *i* any valid index in *L*, *s* any iterable, and *x* any object.

Table 3-5. List object methods

Nonmutating	
`count`	`L.count(x)` Returns the number of items of *L* that are equal to *x*.
`index`	`L.index(x)` Returns the index of the first occurrence of an item in *L* that is equal to *x*, or raises an exception if *L* has no such item.
Mutating	
`append`	`L.append(x)` Appends item *x* to the end of *L* ; like `L[len(L):] = [x]`.
`clear`	`L.clear()` Removes all items from *L*, leaving *L* empty.
`extend`	`L.extend(s)` Appends all the items of iterable *s* to the end of *L*; like `L[len(L):] = s` or `L += s`.

`insert`	`L.insert(i, x)` Inserts item *x* in `L` before the item at index `i`, moving following items of `L` (if any) "rightward" to make space (increases `len(L)` by one, does not replace any item, does not raise exceptions; acts just like `L[i:i]=[x]`).
`pop`	`L.pop(i=-1)` Returns the value of the item at index `i` and removes it from `L`; when you omit `i`, removes and returns the last item; raises an exception when `L` is empty or `i` is an invalid index in `L`.
`remove`	`L.remove(x)` Removes from `L` the first occurrence of an item in `L` that is equal to *x*, or raises an exception when `L` has no such item.
`reverse`	`L.reverse()` Reverses, in place, the items of `L`.
`sort`	`L.sort(key=None, reverse=False)` Sorts, in place, the items of `L` (in ascending order, by default; in descending order, if the argument `reverse` is **`True`**). When the argument `key` is not **`None`**, what gets compared for each item *x* is `key(x)`, not *x* itself. For more details, see the following section.

All mutating methods of list objects, except `pop`, return **`None`**.

Sorting a list

A list's `sort` method causes the list to be sorted in place (reordering items to place them in increasing order) in a way that is guaranteed to be stable

(elements that compare equal are not exchanged). In practice, `sort` is extremely fast—often *preternaturally* fast, as it can exploit any order or reverse order that may be present in any sublist (the advanced algorithm `sort` uses, known as *timsort*[17] to honor its inventor, great Pythonista **Tim Peters**, is a "non-recursive adaptive stable natural mergesort/binary insertion sort hybrid"—now *there's* a mouthful for you!).

The `sort` method takes two optional arguments, which may be passed with either positional or named-argument syntax. The argument `key`, if not **`None`**, must be a function that can be called with any list item as its only argument. In this case, to compare any two items *x* and *y*, Python compares `key(`*x*`)` and `key(`*y*`)` rather than *x* and *y* (internally, Python implements this in the same way as the decorate–sort–undecorate idiom presented in **"Searching and sorting"**, but it's much faster). The argument `reverse`, if **`True`**, causes the result of each comparison to be reversed; this is not exactly the same thing as reversing *L* after sorting, because the sort is stable (elements that compare equal are never exchanged) whether the argument `reverse` is **`True`** or **`False`**. In other words, Python sorts the list in ascending order by default, or in descending order if `reverse` is **`True`**:

```
mylist = ['alpha', 'Beta', 'GAMMA']
mylist.sort()                   # ['Beta', 'GAMMA', 'alpha']
mylist.sort(key=str.lower)      # ['alpha', 'Beta', 'GAMMA']
```

Python also provides the built-in function `sorted` (covered in **Table 8-2**) to produce a sorted list from any input iterable. `sorted`, after the first argument (which is the iterable supplying the items), accepts the same two optional arguments as a list's `sort` method.

The standard library module `operator` (covered in **"The operator Module"**) supplies higher-order functions `attrgetter`, `itemgetter`, and `methodcaller`, which produce functions particularly suitable for the optional `key` argument of the list's `sort` method and the built-in function `sorted`. This optional argument also exists, with exactly the same mean-

ing, for the built-in functions `min` and `max`, as well as for the functions `nsmallest`, `nlargest`, and `merge` in the standard library module `heapq` (covered in **"The heapq Module"**) and the class `groupby` in the standard library module `itertools` (covered in **"The itertools Module"**).

Set Operations

Python provides a variety of operations applicable to sets (both plain and frozen). Since sets are containers, the built-in `len` function can take a set as its single argument and return the number of items in the set. A set is iterable, so you can pass it to any function or method that takes an iterable argument. In this case, iteration yields the items of the set in some arbitrary order. For example, for any set *`S`*, `min(`*`S`*`)` returns the smallest item in *`S`*, since `min` with a single argument iterates on that argument (the order does not matter, because the implied comparisons are transitive).

Set Membership

The *`k`* **`in`** *`S`* operator checks whether the object *`k`* equals one of the items in the set *`S`*. It returns **`True`** when the set contains *`k`*, and **`False`** when it doesn't. *`k`* **`not in`** *`S`* is like **`not`** `(`*`k`* **`in`** *`S`*`)`.

Set Methods

Set objects provide several methods, as shown in **Table 3-6**. Nonmutating methods return a result without altering the object to which they apply, and can also be called on instances of `frozenset`; mutating methods may alter the object to which they apply, and can be called only on instances of `set`. In this table, *`s`* denotes any set object, *`s1`* any iterable with hashable items (often but not necessarily a `set` or `frozenset`), and *`x`* any hashable object.

Table 3-6. Set object methods

Nonmutating

`copy`	`s.copy()` Returns a shallow copy of *`s`* (a copy whose items are the same objects as *`s`*'s, not copies thereof); like `set(s)`
`difference`	`s.difference(s1)` Returns the set of all items of *`s`* that aren't in *`s1`*; can be written as *`s`* `-` *`s1`*
`intersection`	`s.intersection(s1)` Returns the set of all items of *`s`* that are also in *`s1`*; can be written as *`s`* `&` *`s1`*
`isdisjoint`	`s.isdisjoint(s1)` Returns **`True`** if the intersection of *`s`* and *`s1`* is the empty set (they have no items in common), and otherwise returns **`False`**
`issubset`	`s.issubset(s1)` Returns **`True`** when all items of *`s`* are also in *`s1`*, and otherwise returns **`False`**; can be written as *`s`* `<=` *`s1`*
`issuperset`	`s.issuperset(s1)` Returns **`True`** when all items of *`s1`* are also in *`s`*, and otherwise returns **`False`** (like *`s1`*`.issubset(`*`s`*`)`); can be written as *`s`* `>=` *`s1`*
`symmetric_difference`	`s.symmetric_difference(s1)` Returns the set of all items that are in either *`s`* or *`s1`*, but not both; can be written *`s`* `^` *`s1`*
`union`	`s.union(s1)` Returns the set of all items that are in *`s`*, *`s1`*, or both; can be written as *`s`* `|` *`s1`*

Mutating	
`add`	`s.add(x)` Adds *x* as an item to *s*; no effect if *x* was already an item in *s*
`clear`	`s.clear()` Removes all items from *s*, leaving *s* empty
`discard`	`s.discard(x)` Removes *x* as an item of *s*; no effect when *x* was not an item of *s*
`pop`	`s.pop()` Removes and returns an arbitrary item of *s*
`remove`	`s.remove(x)` Removes *x* as an item of *s*; raises a `KeyError` exception when *x* was not an item of *s*

All mutating methods of set objects, except `pop`, return **`None`**.

The `pop` method can be used for destructive iteration on a set, consuming little extra memory. The memory savings make `pop` usable for a loop on a huge set, when what you want is to "consume" the set in the course of the loop. Besides saving memory, a potential advantage of a destructive loop such as this:

```
while S:
    item = S.pop()
    # ...handle item...
```

in comparison to a nondestructive loop such as this:

```
for item in S:
    # ...handle item...
```

is that in the body of the destructive loop you're allowed to modify *S* (adding and/or removing items), which is not allowed in the nondestructive loop.

Sets also have mutating methods named `difference_update`, `intersection_update`, `symmetric_difference_update`, and `update` (corresponding to the nonmutating method `union`). Each such mutating method performs the same operation as the corresponding nonmutating method, but it performs the operation in place, altering the set on which you call it, and returns **`None`**.

The four corresponding nonmutating methods are also accessible with operator syntax (where *S2* is a `set` or `frozenset`, respectively, *S* `-` *S2*, *S* `&` *S2*, *S* `^` *S2*, and *S* `|` *S2*) and the mutating methods are accessible with augmented assignment syntax (respectively, *S* `-=` *S2*, *S* `&=` *S2*, *S* `^=` *S2*, and *S* `|=` *S2*). In addition, `sets` and `frozensets` also support comparison operators: `==` (the sets have the same items; that is, they're "equal" sets), `!=` (the reverse of `==`), `>=` (`issuperset`), `<=` (`issubset`), `<` (`issubset` and not equal), and `>` (`issuperset` and not equal).

When you use operator or augmented assignment syntax, both operands must be `sets` or `frozensets`; however, when you call named methods, argument *S1* can be any iterable with hashable items, and it works just as if the argument you passed was `set(`*S1*`)`.

Dictionary Operations

Python provides a variety of operations applicable to dictionaries. Since dictionaries are containers, the built-in `len` function can take a dictionary as its argument and return the number of items (key/value pairs) in the dictionary. A dictionary is iterable, so you can pass it to any function that takes an iterable argument. In this case, iteration yields only the keys of

the dictionary, in insertion order. For example, for any dictionary *D*, `min(D)` returns the smallest key in *D* (the order of keys in the iteration doesn't matter here).

Dictionary Membership

The *k* **in** *D* operator checks whether the object *k* is a key in the dictionary *D*. It returns **True** if the key is present, and **False** otherwise. *k* **not in** *D* is like **not** (*k* **in** *D*).

Indexing a Dictionary

To denote the value in a dictionary *D* currently associated with the key *k*, use an indexing: *D*[*k*]. Indexing with a key that is not present in the dictionary raises an exception. For example:

```
d = {'x':42, 'h':3.14, 'z':7}
d['x']                          # 42
d['z']                          # 7
d['a']                          # raises KeyError exception
```

Plain assignment to a dictionary indexed with a key that is not yet in the dictionary (e.g., *D*[*newkey*]=*value*) is a valid operation and adds the key and value as a new item in the dictionary. For instance:

```
d = {'x':42, 'h':3.14}
d['a'] = 16                     # d is now {'x':42, 'h':3.14, 'a':16}
```

The **del** statement, in the form **del** *D*[*k*], removes from the dictionary the item whose key is *k*. When *k* is not a key in dictionary *D*, **del** *D*[*k*] raises a `KeyError` exception.

Dictionary Methods

Dictionary objects provide several methods, as shown in **Table 3-7**. Nonmutating methods return a result without altering the object to which they apply, while mutating methods may alter the object to which they apply. In this table, *d* and *d1* indicate any dictionary objects, *k* any hashable object, and *x* any object.

Table 3-7. Dictionary object methods

Nonmutating	
`copy`	*d*`.copy()` Returns a shallow copy of the dictionary (a copy whose items are the same objects as *D*'s, not copies thereof, just like `dict(`*d*`)`)
`get`	*d*`.get(`*k*`[, `*x*`])` Returns *d*`[`*k*`]` when *k* is a key in *d*; otherwise, returns *x* (or **None**, when you don't pass *x*)
`items`	*d*`.items()` Returns an iterable view object whose items are all current items (key/value pairs) in *d*
`keys`	*d*`.keys()` Returns an iterable view object whose items are all current keys in *d*
`values`	*d*`.values()` Returns an iterable view object whose items are all current values in *d*
Mutating	

`clear`	`d.clear()` Removes all items from *d*, leaving *d* empty
`pop`	`d.pop(k[, x])` Removes and returns `d[k]` when *k* is a key in *d*; otherwise, returns *x* (or raises a `KeyError` exception when you don't pass *x*)
`popitem`	`d.popitem()` Removes and returns the items from *d* in last-in, first-out order
`setdefault`	`d.setdefault(k, x)` Returns `d[k]` when *k* is a key in *d*; otherwise, sets `d[k]` equal to *x* (or **`None`**, when you don't pass *x*), then returns `d[k]`
`update`	`d.update(d1)` For each *k* in mapping *d1*, sets `d[k]`equal to `d1[k]`

The `items`, `keys`, and `values` methods return values known as *view objects*. If the underlying `dict` changes, the retrieved view also changes; Python doesn't allow you to alter the set of keys in the underlying `dict` while using a **`for`** loop on any of its view objects.

Iterating on any of the view objects yields values in insertion order. In particular, when you call more than one of these methods without any intervening change to the `dict`, the order of the results is the same for all of them.

Dictionaries also support the class method `fromkeys(seq, value)`, which returns a dictionary containing all the keys of the given iterable *seq*, each identically initialized with *value*.

NEVER MODIFY A DICT'S KEYS WHILE ITERATING ON IT

Don't ever modify the set of keys in a `dict` (i.e., add or remove keys) while iterating over that `dict` or any of the iterable views returned by its methods. If you need to avoid such constraints against mutation during iteration, iterate instead on a list explicitly built from the `dict` or view (i.e., on list(*D*)). Iterating directly on a dict *D* is exactly like iterating on *D*`.keys()`.

The return values of the `items` and `keys` methods also implement set nonmutating methods and behave much like `frozensets`; the return value of the method `values` doesn't, since, in contrast to the others (and to `sets`), it may contain duplicates.

The `popitem` method can be used for destructive iteration on a dictionary. Both `items` and `popitem` return dictionary items as key/value pairs. `popitem` is usable for a loop on a huge dictionary, when what you want is to "consume" the dictionary in the course of the loop.

D`.setdefault(`*k*`,` *x*`)` returns the same result as *D*`.get(`*k*`,` *x*`)`; but, when *k* is not a key in *D*, `setdefault` also has the side effect of binding *D*`[`*k*`]` to the value *x*. (In modern Python, `setdefault` is not often used, since the type `collections.defaultdict`, covered in **"defaultdict"**, often offers similar, faster, clearer functionality.)

The `pop` method returns the same result as `get`, but when *k* is a key in *D*, `pop` also has the side effect of removing *D*`[`*k*`]` (when *x* is not specified, and *k* is not a key in *D*, `get` returns **None**, but `pop` raises an exception). *d*`.pop(`*key*`,` **None**`)` is a useful shortcut for removing a key from a `dict` without having to first check if the key is present, much like *s*`.discard(`*x*`)` (as opposed to *s*`.remove(`*x*`)`) when *s* is a set.

3.9+ The `update` method is accessible with augmented assignment syntax: where *D2* is a `dict`, *D* `|=` *D2* is the same as *D*`.update(`*D2*`)`. Operator syntax, *D* `|` *D2*, mutates neither dictionary: rather, it returns a new dictionary result, such that *D3* `=` *D* `|` *D2* is equivalent to *D3* `=` *D*`.copy();` *D3*`.update(`*D2*`)`.

The `update` method (but not the `|` and `|=` operators) can also accept an iterable of key/value pairs as an alternative argument instead of a mapping, and can accept named arguments instead of—or in addition to—its positional argument; the semantics are the same as for passing such arguments when calling the built-in `dict` type, as covered in **"Dictionaries"**.

Control Flow Statements

A program's *control flow* regulates the order in which the program's code executes. The control flow of a Python program mostly depends on conditional statements, loops, and function calls. (This section covers the **`if`** and **`match`** conditional statements, and **`for`** and **`while`** loops; we cover functions in the following section.) Raising and handling exceptions also affects control flow (via the **`try`** and **`with`** statements); we cover exceptions in **Chapter 6**.

The if Statement

Often, you'll need to execute some statements only when some condition holds, or choose statements to execute depending on mutually exclusive conditions. The compound statement **`if`**—comprising **`if`**, **`elif`**, and **`else`** clauses—lets you conditionally execute blocks of statements. The syntax for the **`if`** statement is:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

The **`elif`** and **`else`** clauses are optional. Before the introduction of the **`match`** construct, which we'll look at next, using **`if`**, **`elif`**, and **`else`** was the

most common approach for all conditional processing (although at times a `dict` with callables as values might provide a good alternative).

Here's a typical **`if`** statement with all three kinds of clauses:

```
if x < 0:
    print('x is negative')
elif x % 2:
    print('x is positive and odd')
else:
    print('x is even and nonnegative')
```

Each clause controls one or more statements (known as a block): place the block's statements on separate logical lines after the line containing the clause's keyword (known as the *header line* of the clause), indented four spaces past the header line. The block terminates when the indentation returns to the level of the clause header, or further left from there (this is the style mandated by **PEP 8**).

You can use any Python expression[18] as the condition in an **`if`** or **`elif`** clause. Using an expression this way is known as using it *in a Boolean context*. In this context, any value is taken as being either true or false. As mentioned earlier, any nonzero number or nonempty container (string, tuple, list, dictionary, set, etc.) evaluates as true, while zero (`0`, of any numeric type), **`None`**, and empty containers evaluate as false. To test a value *x* in a Boolean context, use the following coding style:

```
if x:
```

This is the clearest and most Pythonic form.

Do *not* use any of the following:

```
if x is True:
```

```
if x == True:
if bool(x):
```

There is a crucial difference between saying that an expression *returns* `True` (meaning the expression returns the value `1` with the `bool` type) and saying that an expression *evaluates as* true (meaning the expression returns any result that is true in a Boolean context). When testing an expression, for example in an `if` clause, you only care about what it *evaluates as*, not what, precisely, it *returns*. As we previously mentioned, "evaluates as true" is often expressed informally as "is truthy," and "evaluated as false" as "is falsy."

When the `if` clause's condition evaluates as true, the statements within the `if` clause execute, then the entire `if` statement ends. Otherwise, Python evaluates each `elif` clause's condition, in order. The statements within the first `elif` clause whose condition evaluates as true, if any, execute, and the entire `if` statement ends. Otherwise, when an `else` clause exists, it executes. In any case, statements following the entire `if` construct, at the same level, execute next.

The match Statement

3.10+ The `match` statement brings *structural pattern matching* to the Python language. You might think of this as doing for other Python types something similar to what the `re` module (see **"Regular Expressions and the re Module"**) does for strings: it allows easy testing of the structure and contents of Python objects.[19] Resist the temptation to use `match` unless there is a need to analyze the *structure* of an object.

The overall syntactic structure of the statement is the new (soft) keyword `match` followed by an expression whose value becomes the *matching subject*. This is followed by one or more indented `case` clauses, each of which controls the execution of the indented code block it contains:

```
match expression:
    case pattern [if guard]:
```

```
            statement(s)
        # ...
```

In execution, Python first evaluates the *expression*, then tests the resulting subject value against the *pattern* in each **case** in turn, in order from first to last, until one matches: then, the block indented within the matching **case** clause evaluates. A pattern can do two things:

- Verify that the subject is an object with a particular structure.
- Bind matched components to names for further use (usually within the associated **case** clause).

When a pattern matches the subject, the *guard* allows a final check before selection of the case for execution. All the pattern's name bindings have occurred, and you can use them in the guard. When there is no guard, or when the guard evaluates as true, the case's indented code block executes, after which the **match** statement's execution is complete and no further cases are checked.

The **match** statement, per se, provides no default action. If one is needed, the last **case** clause must specify a *wildcard* pattern—one whose syntax ensures it matches any subject value. It is a `SyntaxError` to follow a **case** clause having such a wildcard pattern with any further **case** clauses.

Pattern elements cannot be created in advance, bound to variables, and (for example) reused in multiple places. Pattern syntax is only valid immediately following the (soft) keyword **case**, so there is no way to perform such an assignment. For each execution of a **match** statement, the interpreter is free to cache pattern expressions that repeat inside the cases, but the cache starts empty for each new execution.

We'll first describe the various types of pattern expressions, before discussing guards and providing some more complex examples.

PATTERN EXPRESSIONS HAVE THEIR OWN SEMANTICS

The syntax of pattern expressions might seem familiar, but their *interpretation* is sometimes quite different from that of nonpattern expressions, which could mislead readers unaware of those differences. Specific syntactic forms are used in the `case` clause to indicate matching of particular structures. A complete summary of this syntax would require more than the simplified notation we use in this book;[20] we therefore prefer to explain this new feature in plain language, with examples. For more detailed examples, refer to the Python **documentation**.

Building patterns

Patterns are expressions, though with a syntax specific to the `case` clause, so familiar grammatical rules apply even though certain features are interpreted differently. They can be enclosed in parentheses to let elements of a pattern be treated as a single expression unit. Like other expressions, patterns have a recursive syntax and can be combined to form more complex patterns. Let's start with the simplest patterns first.

Literal patterns

Most literal values are valid patterns. Integer, float, complex number, and string literals (but *not* formatted string literals) are all permissible,[21] and all succeed in matching subjects of the same type and value:

```
>>> for subject in (42, 42.0, 42.1, 1+1j, b'abc', 'abc'):
...     print(subject, end=': ')
...     match subject:
...         case 42: print('integer')  # note this matches 42.0, too!
...         case 42.1: print('float')
...         case 1+1j: print('complex')
...         case b'abc': print('bytestring')
...         case 'abc': print('string')
```

```
42: integer
42.0: integer
```

```
42.1: float
(1+1j): complex
b'abc': bytestring
abc: string
```

For most matches, the interpreter checks for equality without type checking, which is why `42.0` matches integer `42`. If the distinction is important, consider using class matching (see **"Class patterns"**) rather than literal matching. **`True`**, **`False`**, and **`None`** being singleton objects, each matches itself.

The wildcard pattern

In pattern syntax, the underscore (_) plays the role of a wildcard expression. As the simplest wildcard pattern, _ matches any value at all:

```
>>> for subject in 42, 'string', ('tu', 'ple'), ['list'], object:
...     match subject:
...         case _: print('matched', subject)
...
```

```
matched 42
matched string
matched ('tu', 'ple')
matched ['list']
matched <class 'object'>
```

Capture patterns

The use of unqualified names (names with no dots in them) is so different in patterns that we feel it necessary to begin this section with a warning.

SIMPLE NAMES BIND TO MATCHED ELEMENTS INSIDE PATTERNS

Unqualified names—simple identifiers (e.g., *color*) rather than attribute references (e.g., *name.attr*)—do not necessarily have their usual meaning in pattern expressions. Some names, rather than being references to values, are instead bound to elements of the subject value during pattern matching.

Unqualified names, except _, are *capture patterns*. They're wildcards, matching anything, but with a side effect: the name, in the current local namespace, gets bound to the object matched by the pattern. Bindings created by matching remain after the statement has executed, allowing the statements in the **`case`** clause and subsequent code to process extracted portions of the subject value.

The following example is similar to the preceding one, except that the name x, instead of the underscore, matches the subject. The absence of exceptions shows that the name captures the whole subject in each case:

```
>>> for subject in 42, 'string', ('tu', 'ple'), ['list'], object:
...     match subject:
...         case x: assert x == subject
...
```

Value patterns

This section, too, begins with a reminder to readers that simple names can't be used to inject their bindings into pattern values to be matched.

REPRESENT VARIABLE VALUES IN PATTERNS WITH QUALIFIED NAMES

Because simple names capture values during pattern matching, you *must* use attribute references (qualified names like *name.attr*) to express values that may change between different executions of the same **`match`** statement.

Though this feature is useful, it means you can't reference values directly with simple names. Therefore, in patterns, values must be represented by qualified names, which are known as *value patterns*—they *represent* values, rather than *capturing* them as simple names do. While slightly inconvenient, to use qualified names you can just set attribute values on an otherwise empty class.[22] For example:

```
>>> class m: v1 = "one"; v2 = 2; v3 = 2.56
...
>>> match ('one', 2, 2.56):
...     case (m.v1, m.v2, m.v3):  print('matched')
...
```

```
matched
```

It is easy to give yourself access to the current module's "global" namespace, like this:

```
>>> import sys
>>> g = sys.modules[__name__]
>>> v1 = "one"; v2 = 2; v3 = 2.56
>>> match ('one', 2, 2.56):
...     case (g.v1, g.v2, g.v3):  print('matched')
...
```

```
matched
```

OR patterns

When *P1* and *P2* are patterns, the expression *P1* | *P2* is an *OR pattern*, matching anything that matches either *P1* or *P2*, as shown in the following example. Any number of alternate patterns can be used, and matches are attempted from left to right:

```
>>> for subject in range(5):
...     match subject:
...         case 1 | 3: print('odd')
...         case 0 | 2 | 4: print('even')
...
```

```
even
odd
even
odd
even
```

It is a syntax error to follow a wildcard pattern with further alternatives, however, since they can never be activated. While our initial examples are simple, remember that the syntax is recursive, so patterns of arbitrary complexity can replace any of the subpatterns in these examples.

Group patterns

If *P1* is a pattern, then `(`*P1*`)` is also a pattern that matches the same values. This addition of "grouping" parentheses is useful when patterns become complicated, just as it is with standard expressions. Like in other expressions, take care to distinguish between `(`*P1*`)`, a simple grouped pattern matching *P1*, and `(`*P1*`,)`, a sequence pattern (described next) matching a sequence with a single element matching *P1*.

Sequence patterns

A list or tuple of patterns, optionally with a single starred wildcard (`*_`) or starred capture pattern (`*`*name*), is a *sequence pattern*. When the starred pattern is absent, the pattern matches a fixed-length sequence of values of the same length as the pattern. Elements of the sequence are matched one at a time, until all elements have matched (then matching succeeds) or an element fails to match (then matching fails).

When the sequence pattern includes a starred pattern, that subpattern matches a sequence of elements sufficiently long to allow the remaining unstarred patterns to match the final elements of the sequence. When the starred pattern is of the form **name*, *name* is bound to the (possibly empty) list of the elements in the middle that don't correspond to individual patterns at the beginning or end.

You can match a sequence with patterns that look like tuples or lists—it makes no difference to the matching process. The next example shows an unnecessarily complicated way to extract the first, middle, and last elements of a sequence:

```
>>> for sequence in (["one", "two", "three"], range(2), range(6)):
...     match sequence:
...         case  (first, *vars, last): print(first, vars, last)
...
```

```
one ['two'] three
0 [] 1
0 [1, 2, 3, 4] 5
```

as patterns

You can use so-called ***as*** *patterns* to capture values matched by more complex patterns, or components of a pattern, that simple capture patterns (see **"Capture patterns"**) cannot.

When *P1* is a pattern, then *P1* **as** *name* is also a pattern; when *P1* succeeds, Python binds the matched value to the name *name* in the local namespace. The interpreter tries to ensure that, even with complicated patterns, the same bindings always take place when a match occurs. Therefore, each of the next two examples raises `SyntaxError`, because the constraint cannot be guaranteed:

```
>>> match subject:
...     case ((0 | 1) as x) | 2: print(x)
...
```

```
SyntaxError: alternative patterns bind different names
```

```
>>> match subject:
...     case (2 | x): print(x)
...
```

```
SyntaxError: alternative patterns bind different names
```

But this one works:

```
>>> match 42:
...     case (1 | 2 | 42) as x: print(x)
...
```

```
42
```

Mapping patterns

Mapping patterns match mapping objects, usually dictionaries, that associate keys with values. The syntax of mapping patterns uses *`key`*`:` *`pattern`* pairs. The keys must be either literal or value patterns.

The interpreter iterates over the keys in the mapping pattern, processing each as follows:

- Python looks up the key in the subject mapping; a lookup failure causes an immediate match failure.
- Python then matches the extracted value against the pattern associated with the key; if the value fails to match the pattern, then the whole match fails.

When all keys (and associated values) in the mapping pattern match, the whole match succeeds:

```
>>> match {1: "two", "two": 1}:
...     case {1: v1, "two": v2}: print(v1, v2)
...
```

```
two 1
```

You can also use a mapping pattern together with an **as** clause:

```
>>> match {1: "two", "two": 1}:
...     case {1: v1} as v2: print(v1, v2)
...
```

```
two {1: 'two', 'two': 1}
```

The **as** pattern in the second example binds *v2* to the whole subject dictionary, not just the matched keys.

The final element of the pattern may optionally be a double-starred capture pattern such as ***name*. When that is the case, Python binds *name* to a possibly empty dictionary whose items are the (*key*, *value*) pairs from the subject mapping whose keys were *not* present in the pattern:

```
>>> match {1: 'one', 2: 'two', 3: 'three'}:
...     case {2: middle, **others}: print(middle, others)
...
```

```
two {1: 'one', 3: 'three'}
```

Class patterns

The final and maybe the most versatile kind of pattern is the *class pattern*, offering the ability to match instances of particular classes and their attributes.

A class pattern is of the general form:

```
name_or_attr(patterns)
```

where *`name_or_attr`* is a simple or qualified name bound to a class—specifically, an instance of the built-in type `type` (or of a subclass thereof, but no super-fancy metaclasses need apply!)—and *`patterns`* is a (possibly empty) comma-separated list of pattern specifications. When no pattern specifications are present in a class pattern, the match succeeds whenever the subject is an instance of the given class, so for example the pattern `int()` matches *any* integer.

Like function arguments and parameters, the pattern specifications can be positional (like *`pattern`*) or named (like *`name=pattern`*). If a class pattern has positional pattern specifications, they must all precede the first named pattern specification. User-defined classes cannot use positional patterns without setting the class's *`__match_args__`* attribute (see **"Configuring classes for positional matching"**).

The built-in types `bool`, `bytearray`, `bytes`, `dict`,[23] `float`, `frozenset`, `int`, `list`, `set`, `str`, and `tuple`, as well as any `namedtuple` and any `dataclass`,

are all configured to take a single positional pattern, which is matched against the instance value. For example, the pattern `str(x)` matches any string and binds its value to *x* by matching the string's value against the capture pattern—as does `str() as x`.

You may remember a literal pattern example we presented earlier, showing that literal matching could not discriminate between the integer `42` and the float `42.0` because `42 == 42.0`. You can use class matching to overcome that issue:

```
>>> for subject in 42, 42.0:
...     match subject:
...         case int(x): print('integer', x)
...         case float(x): print('float', x)
...
```

```
integer 42
float 42.0
```

Once the type of the subject value has matched, for each of the named patterns *name=pattern*, Python retrieves the attribute *name* from the instance and matches its value against *pattern*. If all named pattern matches succeed, the whole match succeeds. Python handles positional patterns by converting them to named patterns, as you'll see momentarily.

Guards

When a **case** clause's pattern succeeds, it is often convenient to determine on the basis of values extracted from the match whether this **case** should execute. When a guard is present, it executes after a successful match. If the guard expression evaluates as false, Python abandons the current **case**, despite the match, and moves on to consider the next case. This example uses a guard to exclude odd integers by checking the value bound in the match:

```
>>> for subject in range(5):
...     match subject:
...         case int(i) if i % 2 == 0: print(i, "is even")
...
```

```
0 is even
2 is even
4 is even
```

Configuring classes for positional matching

When you want your own classes to handle positional patterns in matching, you have to tell the interpreter which *attribute of the instance* (not which *argument to* `__init__`) each positional pattern corresponds to. You do this by setting the class's `__match_args__` attribute to a sequence of names. The interpreter raises a `TypeError` exception if you attempt to use more positional patterns than you defined:

```
>>> class Color:
...     __match_args__ = ('red', 'green', 'blue')
...     def __init__(self, r, g, b, name='anonymous'):
...         self.name = name
...         self.red, self.green, self.blue = r, g, b
...
>>> color_red = Color(255, 0, 0, 'red')
>>> color_blue = Color(0, 0, 255)
>>> for subject in (42.0, color_red, color_blue):
...     match subject:
...         case float(x):
...             print('float', x)
...         case Color(red, green, blue, name='red'):
...             print(type(subject).__name__, subject.name,
...                   red, green, blue)
...         case Color(red, green, 255) as color:
...             print(type(subject).__name__, color.name,
...                   red, green, color.blue)
```

```
...         case _: print(type(subject), subject)
...
```

```
float 42.0
Color red 255 0 0
Color anonymous 0 0 255
```

```
>>> match color_red:
...     case Color(red, green, blue, name):
...         print("matched")
...
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Color() accepts 3 positional sub-patterns (4 given)
```

The while Statement

The **`while`** statement repeats execution of a statement or block of statements for as long as a conditional expression evaluates as true. Here's the syntax of the **`while`** statement:

```
while expression:
    statement(s)
```

A **`while`** statement can also include an **`else`** clause and **`break`** and **`continue`** statements, all of which we'll discuss after we look at the **`for`** statement.

Here's a typical **`while`** statement:

```
count = 0
while x > 0:
    x //= 2             # floor division
    count += 1
print('The approximate log2 is', count)
```

First Python evaluates *expression*, which is known as the *loop condition*, in a Boolean context. When the condition evaluates as false, the **while** statement ends. When the loop condition evaluates as true, the statement or block of statements that make up the *loop body* executes. Once the loop body finishes executing, Python evaluates the loop condition again, to check whether another iteration should execute. This process continues until the loop condition evaluates as false, at which point the **while** statement ends.

The loop body should contain code that eventually makes the loop condition false, since otherwise the loop never ends (unless the body raises an exception or executes a **break** statement). A loop within a function's body also ends if the loop body executes a **return** statement, since in this case the whole function ends.

The for Statement

The **for** statement repeats execution of a statement or block of statements controlled by an iterable expression. Here's the syntax:

```
for target in iterable:
    statement(s)
```

The **in** keyword is part of the syntax of the **for** statement; its purpose here is distinct from the **in** operator, which tests membership.

Here's a rather typical **for** statement:

```
for letter in 'ciao':
    print(f'give me a {letter}...')
```

A **`for`** statement can also include an **`else`** clause and **`break`** and **`continue`** statements; we'll discuss all of these shortly, starting with **"The else Clause on Loop Statements"**. As mentioned previously, *`iterable`* may be any iterable Python expression. In particular, any sequence is iterable. The interpreter implicitly calls the built-in `iter` on the iterable, producing an *iterator* (discussed in the following subsection), which it then iterates over.

`target` is normally an identifier naming the *control variable* of the loop; the **`for`** statement successively rebinds this variable to each item of the iterator, in order. The statement or statements that make up the *loop body* execute once for each item in *`iterable`* (unless the loop ends because of an exception or a **`break`** or **`return`** statement). Since the loop body may terminate before the iterator is exhausted, this is one case in which you may use an *unbounded* iterable—one that, per se, would never cease yielding items.

You can also use a target with multiple identifiers, as in an unpacking assignment. In this case, the iterator's items must themselves be iterables, each with exactly as many items as there are identifiers in the target. For example, when *d* is a dictionary, this is a typical way to loop on the items (key/value pairs) in *d*:

```
for key, value in d.items():
    if key and value:         # print only truthy keys and values
        print(key, value)
```

The `items` method returns another kind of iterable (a *view*), whose items are key/value pairs; so, we use a **`for`** loop with two identifiers in the target to unpack each item into `key` and `value`.

Precisely *one* of the identifiers may be preceded by a star, in which case the starred identifier is bound to a list of all items not assigned to other targets. Although components of a target are commonly identifiers, values can be bound to any acceptable LHS expression, as covered in **"Assignment Statements"**—so, the following is correct, although not the most readable style:

```
prototype = [1, 'placeholder', 3]
for prototype[1] in 'xyz':
    print(prototype)
# prints [1, 'x', 3], then [1, 'y', 3], then [1, 'z', 3]
```

DON'T ALTER MUTABLE OBJECTS WHILE LOOPING ON THEM

When an iterator has a mutable underlying iterable, don't alter that underlying object during the execution of a **`for`** loop on the iterable. For example, the preceding key/value printing example cannot alter *d*. The `items` method returns a "view" iterable whose underlying object is *d*, so the loop body cannot mutate the set of keys in *d* (e.g., by executing **`del`** *d*[*key*]). To ensure that *d* is not the underlying object of the iterable, you may, for example, iterate over `list(`*d*`.items())` to allow the loop body to mutate *d*. Specifically:

- When looping on a list, do not insert, append, or delete items (rebinding an item at an existing index is OK).
- When looping on a dictionary, do not add or delete items (rebinding the value for an existing key is OK).
- When looping on a set, do not add or delete items (no alteration is permitted).

The loop body may rebind the control target variable(s), but the next iteration of the loop always rebinds them again. If the iterator yields no items, the loop body does not execute at all. In this case, the **`for`** statement does not bind or rebind its control variable in any way. If the iterator yields at least one item, however, then when the loop statement ends, the control variable remains bound to the last value to which the loop state-

ment bound it. The following code is therefore correct *only* when `someseq` is not empty:

```
for x in someseq:
    process(x)
# potential NameError if someseq is empty
print(f'Last item processed was {x}')
```

Iterators

An *iterator* is an object `i` such that you can call `next(i)`, which returns the next item of iterator `i` or, when exhausted, raises a `StopIteration` exception. Alternatively, you can call `next(i, default)`, in which case, when iterator `i` has no more items, the call returns `default`.

When you write a class (see **"Classes and Instances"**), you can let instances of the class be iterators by defining a special method `__next__` that takes no argument except `self`, and returns the next item or raises `StopIteration`. Most iterators are built by implicit or explicit calls to the built-in function `iter`, covered in **Table 8-2**. Calling a generator also returns an iterator, as we discuss in **"Generators"**.

As pointed out previously, the **`for`** statement implicitly calls `iter` on its iterable to get an iterator. The statement:

```
for x in c:
    statement(s)
```

is exactly equivalent to:

```
_temporary_iterator = iter(c)
while True:
    try:
        x = next(_temporary_iterator)
    except StopIteration:
```

```
        break
    statement(s)
```

where *_temporary_iterator* is an arbitrary name not used elsewhere in the current scope.

Thus, when `iter(c)` returns an iterator *i* such that `next(i)` never raises `StopIteration` (an *unbounded iterator*), the loop **`for`** *x* **`in`** *c* continues indefinitely unless the loop body includes suitable **`break`** or **`return`** statements, or raises or propagates exceptions. `iter(c)`, in turn, calls the special method *c*`.__iter__()` to obtain and return an iterator on *c*. We'll talk more about `__iter__` in the following subsection and in **"Container methods"**.

Many of the best ways to build and manipulate iterators are found in the standard library module `itertools`, covered in **"The itertools Module"**.

Iterables versus iterators

Python's built-in sequences, like all iterables, implement an `__iter__` method, which the interpreter calls to produce an iterator over the iterable. Because each call to the built-in's `__iter__` method produces a new iterator, it is possible to nest multiple iterations over the same iterable:

```
>>> iterable = [1, 2]
>>> for i in iterable:
...     for j in iterable:
...         print(i, j)
...
```

```
1 1
1 2
2 1
2 2
```

Iterators also implement an `__iter__` method, but it always returns `self`, so nesting iterations over an iterator doesn't work as you might expect:

```
>>> iterator = iter([1, 2])
>>> for i in iterator:
...     for j in iterator:
...         print(i, j)
...
```

```
1 2
```

Here, both the inner and outer loops are iterating over the same iterator. By the time the inner loop first gets control, the first value from the iterator has already been consumed. The first iteration of the inner loop then exhausts the iterator, making both loops end upon attempting the next iteration.

range

Looping over a sequence of integers is a common task, so Python provides the built-in function `range` to generate an iterable over integers. The simplest way to loop *n* times in Python is:

```
for i in range(n):
    statement(s)
```

`range(x)` generates the consecutive integers from `0` (included) up to *x* (excluded). `range(x, y)` generates a list whose items are consecutive integers from *x* (included) up to *y* (excluded). `range(x, y, stride)` generates a list of integers from *x* (included) up to *y* (excluded), such that the difference between each two adjacent items is *stride*. If *stride* `< 0`, `range` counts down from *x* to *y*.

`range` generates an empty iterator when *x* `>=` *y* and *stride* `>0`, or when *x* `<=` *y* and *stride* `<0`. When *stride* `==0`, `range` raises an exception.

`range` returns a special-purpose object, intended just for use in iterations like the **`for`** statement shown previously. Note that `range` returns an iterable, not an iterator; you can easily obtain such an iterator, should you need one, by calling `iter(range(...))`. The special-purpose object returned by `range` consumes less memory (for wide ranges, *much* less memory) than the equivalent `list` object would. If you really need a `list` that's an arithmetic progression of `int`s, call `list(range(...))`. You will most often find that you don't, in fact, need such a complete list to be fully built in memory.

List comprehensions

A common use of a **`for`** loop is to inspect each item in an iterable and build a new list by appending the results of an expression computed on some or all of the items. The expression form known as a *list comprehension* or *listcomp* lets you code this common idiom concisely and directly. Since a list comprehension is an expression (rather than a block of statements), you can use it wherever you need an expression (e.g., as an argument in a function call, in a **`return`** statement, or as a subexpression of some other expression).

A list comprehension has the following syntax:

```
[ expression for target in iterable lc-clauses ]
```

`target` and *`iterable`* in each **`for`** clause of a list comprehension have the same syntax and meaning as those in a regular **`for`** statement, and the *`expression`* in each **`if`** clause of a list comprehension has the same syntax and meaning as the *`expression`* in a regular **`if`** statement. When *`expression`* denotes a tuple, you must enclose it in parentheses.

`lc-clauses` is a series of zero or more clauses, each with either this form:

```
for target in iterable
```

or this form:

```
if expression
```

A list comprehension is equivalent to a **`for`** loop that builds the same list by repeated calls to the resulting list's `append` method.[24] For example (assigning the list comprehension result to a variable for clarity), this:

```
result = [x+1 for x in some_sequence]
```

is just the same as the **`for`** loop:

```
result = []
for x in some_sequence:
    result.append(x+1)
```

DON'T BUILD A LIST UNLESS YOU NEED TO

If you are just going to loop once over the items, you don't need an actual indexable list: use a generator expression instead (covered in **"Generator expressions"**). This avoids list creation and uses less memory. In particular, resist the temptation to use a list comprehension as a not particularly readable "single-line loop," like this:

```
[fn(x) for x in seq]
```

and then ignore the resulting list—just use a normal **for** loop instead!

Here's a list comprehension that uses an **`if`** clause:

```
result = [x+1 for x in some_sequence if x>23]
```

This list comprehension is the same as a **`for`** loop that contains an **`if`** statement:

```
result = []
for x in some_sequence:
    if x>23:
        result.append(x+1)
```

Here's a list comprehension using a nested **`for`** clause to flatten a "list of lists" into a single list of items:

```
result = [x for sublist in listoflists for x in sublist]
```

This is the same as a **`for`** loop with another **`for`** loop nested inside:

```
result = []
for sublist in listoflists:
    for x in sublist:
        result.append(x)I
```

As these examples show, the order of **`for`** and **`if`** in a list comprehension, is the same as in the equivalent loop, but, in the list comprehension the nesting remains implicit. If you remember "order **`for`** clauses as in a nested loop," that can help you get the ordering of the list comprehension's clauses right.

LIST COMPREHENSIONS AND VARIABLE SCOPE

A list comprehension expression evaluates in its own scope (as do set and dictionary comprehensions, described in the following sections, and generator expressions, discussed toward the end of this chapter). When a *target* component in the **for** statement is a name, the name is defined solely within the expression scope and is not available outside it.

Set comprehensions

A *set comprehension* has exactly the same syntax and semantics as a list comprehension, except that you enclose it in braces ({}) rather than in brackets ([]). The result is a `set`; for example:

```
s = {n//2 for n in range(10)}
print(sorted(s))  # prints: [0, 1, 2, 3, 4]
```

A similar list comprehension would have each item repeated twice, but building a `set` removes duplicates.

Dictionary comprehensions

A *dictionary comprehension* has the same syntax as a set comprehension, except that instead of a single expression before the **`for`** clause, you use two expressions with a colon (`:`) between them: *`key:value`*. The result is a `dict`, which retains insertion order. For example:

```
d = {s: i for (i, s) in enumerate(['zero', 'one', 'two'])}
print(d)          # prints: {'zero': 0, 'one': 1, 'two': 2}
```

The break Statement

You can use a **`break`** statement *only* within a loop body. When **`break`** executes, the loop terminates *without executing any **`else`** clause on the loop*.

When loops are nested, a **`break`** terminates only the innermost nested loop. In practice, a **`break`** is typically within a clause of an **`if`** (or, occasionally, **`match`**) statement in the loop body, so that **`break`** executes conditionally.

One common use of **`break`** is to implement a loop that decides whether it should keep looping only in the middle of each loop iteration (what Donald Knuth called the "loop and a half" structure in his great 1974 paper **"Structured Programming with go to Statements"**[25]). For example:

```
while True:          # this loop can never terminate "naturally"
    x = get_next()
    y = preprocess(x)
    if not keep_looping(x, y):
        break
    process(x, y)
```

The continue Statement

Like **`break`**, the **`continue`** statement can exist only within a loop body. It causes the current iteration of the loop body to terminate, and execution continues with the next iteration of the loop. In practice, a **`continue`** is usually within a clause of an **`if`** (or, occasionally, a **`match`**) statement in the loop body, so that **`continue`** executes conditionally.

Sometimes, a **`continue`** statement can take the place of nested **`if`** statements within a loop. For example, here each `x` has to pass multiple tests before being completely processed:

```
for x in some_container:
    if seems_ok(x):
        lowbound, highbound = bounds_to_test()
        if lowbound <= x < highbound:
            pre_process(x)
```

```
            if final_check(x):
                do_processing(x)
```

Nesting increases with the number of conditions. Equivalent code with **`continue`** "flattens" the logic:

```
for x in some_container:
    if not seems_ok(x):
        continue
    lowbound, highbound = bounds_to_test()
    if x < lowbound or x >= highbound:
        continue
    pre_process(x)
    if final_check(x):
        do_processing(x)
```

FLAT IS BETTER THAN NESTED

Both versions work the same way, so which one you use is a matter of personal preference and style. One of the principles of **The Zen of Python** (which you can see at any time by typing **`import this`** at an interactive Python interpreter prompt) is "Flat is better than nested." The **`continue`** statement is just one way Python helps you move toward the goal of a less-nested structure in a loop, when you choose to follow this tip.

The else Clause on Loop Statements

`while` and **`for`** statements may optionally have a trailing **`else`** clause. The statement or block under that clause executes when the loop terminates *naturally* (at the end of the **`for`** iterator, or when the **`while`** loop condition becomes false), but not when the loop terminates *prematurely* (via **`break`**, **`return`**, or an exception). When a loop contains one or more **`break`** statements, you'll often want to check whether it terminates naturally or prematurely. You can use an **`else`** clause on the loop for this purpose:

```
for x in some_container:
    if is_ok(x):
        break  # item x is satisfactory, terminate loop
else:
    print('Beware: no satisfactory item was found in container')
    x = None
```

The pass Statement

The body of a Python compound statement cannot be empty; it must always contain at least one statement. You can use a **pass** statement, which performs no action, as an explicit placeholder when a statement is syntactically required but you have nothing to do. Here's an example of using **pass** in a conditional statement as a part of some rather convoluted logic to test mutually exclusive conditions:

```
if condition1(x):
    process1(x)
elif x>23 or (x<5 and condition2(x)):
    pass       # nothing to be done in this case
elif condition3(x):
    process3(x)
else:
    process_default(x)
```

EMPTY DEF OR CLASS STATEMENTS: USE A DOCSTRING, NOT PASS

You can also use a docstring, covered in **"Docstrings"**, as the body of an otherwise empty **def** or **class** statement. When you do this, you do not need to also add a **pass** statement (you can do so if you wish, but it's not optimal Python style).

The try and raise Statements

Python supports exception handling with the **try** statement, which includes **try**, **except**, **finally**, and **else** clauses. Your code can also explic-

itly raise an exception with the `raise` statement. When code raises an exception, normal control flow of the program stops and Python looks for a suitable exception handler. We discuss all of this in detail in **"Exception Propagation"**.

The with Statement

A `with` statement can often be a more readable, useful alternative to the `try/finally` statement. We discuss it in detail in **"The with Statement and Context Managers"**. A good grasp of context managers can often help you structure your code more clearly without compromising efficiency.

Functions

Most statements in a typical Python program are part of some function. Code in a function body may be faster than at a module's top level, as covered in **"Avoid exec and from ... import *"**, so there are excellent practical reasons to put most of your code into functions—and there are no disadvantages: clarity, readability and code reusability all improve when you avoid having any substantial chunks of module-level code.

A *function* is a group of statements that execute upon request. Python provides many built-in functions and lets programmers define their own functions. A request to execute a function is known as a *function call*. When you call a function, you can pass arguments that specify data upon which the function performs its computation. In Python, a function always returns a result: either `None` or a value, the result of the computation. Functions defined within `class` statements are also known as *methods*. We cover issues specific to methods in **"Bound and Unbound Methods"**; the general coverage of functions in this section, however, also applies to methods.

Python is somewhat unusual in the flexibility it affords the programmer in defining and calling functions. This flexibility does mean that some constraints are not adequately expressed solely by the syntax. In Python,

functions are objects (values), handled just like other objects. Thus, you can pass a function as an argument in a call to another function, and a function can return another function as the result of a call. A function, just like any other object, can be bound to a variable, can be an item in a container, and can be an attribute of an object. Functions can also be keys in a dictionary. The fact that functions are ordinary objects in Python is often expressed by saying that functions are *first-class* objects.

For example, given a `dict` keyed by functions, with the values being each function's inverse, you could make the dictionary bidirectional by adding the inverse values as keys, with their corresponding keys as values. Here's a small example of this idea, using some functions from the module `math` (covered in **"The math and cmath Modules"**), that takes a one-way mapping of inverse pairs and then adds the inverse of each entry to complete the mapping:

```
def add_inverses(i_dict):
    for f in list(i_dict):  # iterates over keys while mutating i_dict
        i_dict[i_dict[f]] = f
math_map = {sin:asin, cos:acos, tan:atan, log:exp}
add_inverses(math_map)
```

Note that in this case the function mutates its argument (whence its need to use a `list` call for looping). In Python, the usual convention is for such functions not to return a value (see **"The return Statement"**).

Defining Functions: The def Statement

The **`def`** statement is the usual way to create a function. **`def`** is a single-clause compound statement with the following syntax:

```
def function_name(parameters):
   statement(s)
```

function_name is an identifier, and the nonempty indented *statement(s)* are the *function body*. When the interpreter encounters a **def** statement, it compiles the function body, creating a function object, and binds (or rebinds, if there was an existing binding) *function_name* to the compiled function object in the containing namespace (typically the module namespace, or a class namespace when defining methods).

parameters is an optional list specifying the identifiers that will be bound to values that each function call provides. We distinguish between those identifiers, and the values provided for them in calls, as usual in computer science by referring to the former as *parameters* and the latter as *arguments*.

In the simplest case, a function defines no parameters, meaning the function won't accept any arguments when you call it. In this case, the **def** statement has empty parentheses after *function_name*, as must all calls. Otherwise, *parameters* will be a list of specifications (see **"Parameters"**). The function body does not execute when the **def** statement executes; rather, Python compiles it into bytecode, saves it as the function object's `__code__` attribute, and executes it later on each call to the function. The function body can contain zero or more occurrences of the **return** statement, as we'll discuss shortly.

Each call to the function supplies argument expressions corresponding to parameters in the function definition. The interpreter evaluates the argument expressions from left to right and creates a new namespace in which it binds the argument values to the parameter names as local variables of the function call (as we discuss in **"Calling Functions"**). Then Python executes the function body, with the function call namespace as the local namespace.

Here's a simple function that returns a value that is twice the value passed to it each time it's called:

```
def twice(x):
    return x*2
```

The argument can be anything that you can multiply by two, so you could call the function with a number, string, list, or tuple as an argument. Each call returns a new value of the same type as the argument: `twice('ciao')`, for example, returns `'ciaociao'`.

The number of parameters of a function, together with the parameters' names, the number of mandatory parameters, and the information on whether and where unmatched arguments should be collected, are a specification known as the function's *signature*. A signature defines how you can call the function.

Parameters

Parameters (pedantically, *formal parameters*) name the values passed into a function call, and may specify default values for them. Each time you call the function, the call binds each parameter name to the corresponding argument value in a new local namespace, which Python later destroys on function exit.

Besides letting you name individual arguments, Python also lets you collect argument values not matched by individual parameters, and lets you specifically require that some arguments be positional, or be named.

Positional parameters

A positional parameter is an identifier, *`name`*, which names the parameter. You use these names inside the function body to access the argument values to the call. Callers can normally provide values for these parameters with either positional or named arguments (see **"Matching arguments to parameters"**).

Named parameters

Named parameters normally take the form *`name=expression`* (3.8+ or come after a positional argument collector, often just `*`, as discussed shortly). They are also often known (when written in the traditional *`name=expression`* form) as *default*, *optional*, and even—confusingly, since

they do not involve any Python keywords—*keyword* parameters. When it executes a **`def`** statement, the interpreter evaluates each such *`expression`* and saves the resulting value, known as the *default value* for the parameter, among the attributes of the function object. A function call thus need not provide an argument value for a named parameter written in the traditional form: the call uses the default value given as the *`expression`*. You may provide positional arguments as values for some named parameters (3.8+ except for parameters that are identified as named ones by appearing after a positional argument collector; see also **"Matching arguments to parameters"**).

Python computes each default value *exactly once*, when the **`def`** statement executes, *not* each time you call the resulting function. In particular, this means that Python binds exactly the *same* object, the default value, to the named parameter whenever the caller does not supply a corresponding argument.

AVOID MUTABLE DEFAULT VALUES

A function can alter a mutable default value, such as a list, each time you call the function without an argument corresponding to the respective parameter. This is usually not the behavior you want; for details, see **"Mutable default parameter values"**.

Positional-only marker

3.8+ A function's signature may contain a single *positional-only marker* (`/`) as a dummy parameter. The parameters preceding the marker are known as *positional-only parameters*, and *must* be provided as positional arguments, *not* named arguments, when calling the function; using named arguments for these parameters raises a `TypeError` exception.

The built-in `int` type, for example, has the following signature:

```
int(x, /, base=10)
```

When calling `int`, you must pass a value for `x` and you must pass it positionally. `base` (used when `x` is a string to be converted to `int`) is optional, and you may pass it either positionally or as a named argument (it's an error to pass `x` as a number and also pass `base`, but the notation cannot capture that quirk).

Positional argument collector

The positional argument collector can take one of two forms, either **name* or 3.8+ just `*`. In the former case, *name* is bound at call time to a tuple of unmatched positional arguments (when all positional arguments are matched, the tuple is empty). In the latter case (the `*` is a dummy parameter), a call with unmatched positional arguments raises a `TypeError` exception.

When a function's signature has either kind of positional argument collector, no call can provide a positional argument for a named parameter coming after the collector: the collector prohibits (in the `*` form) or gives a destination for (in the **name* form) positional arguments that do not correspond to parameters coming before it.

For example, consider this function from the `random` module:

```
def sample(population, k, *, counts=None):
```

When calling `sample`, values for `population` and `k` are required, and may be passed positionally or by name. `counts` is optional; if you do pass it, you must pass it as a named argument.

Named argument collector

This final, optional parameter specification has the form ***name*. When the function is called, *name* is bound to a dictionary whose items are the (*key*, *value*) pairs of any unmatched named arguments, or an empty dictionary if there are no such arguments.

Parameter sequence

Generally speaking, positional parameters are followed by named parameters, with the positional and named argument collectors (if present) last. The positional-only marker, however, may appear at any position in the list of parameters.

Mutable default parameter values

When a named parameter's default value is a mutable object, and the function body alters the parameter, things get tricky. For example:

```
def f(x, y=[]):
    y.append(x)
    return id(y), y
print(f(23))              # prints: (4302354376, [23])
print(f(42))              # prints: (4302354376, [23, 42])
```

The second `print` prints `[23, 42]` because the first call to `f` altered the default value of `y`, originally an empty list `[]`, by appending `23` to it. The `id` values (always equal to each other, although otherwise arbitrary) confirm that both calls return the same object. If you want `y` to be a new, empty list object, each time you call `f` with a single argument (a far more frequent need!), use the following idiom instead:

```
def f(x, y=None):
    if y is None:
        y = []
    y.append(x)
    return id(y), y
print(f(23))              # prints: (4302354376, [23])
print(f(42))              # prints: (4302180040, [42])
```

There may be cases in which you explicitly want a mutable default parameter value that is preserved across multiple function calls, most often for caching purposes, as in the following example:

```
def cached_compute(x, _cache={}):
    if x not in _cache:
        _cache[x] = costly_computation(x)
    return _cache[x]
```

Such caching behavior (also known as *memoization*) is usually best obtained by decorating the underlying `costly_computation` function with `functools.lru_cache`, covered in **Table 8-7** and discussed in detail in **Chapter 17**.

Argument collector parameters

The presence of argument collectors (the special forms `*`, `*`*name*, or `**`*name*) in a function's signature allows a function to prohibit (`*`) or collect positional (`*`*name*) or named (`**`*name*) arguments that do not match any parameters (see **"Matching arguments to parameters"**). There is no requirement to use specific names—you can use any identifier you want in each special form. `*args` and `**kwds` or `**kwargs`, as well as `*a` and `**k`, are popular choices.

The presence of the special form `*` causes calls with unmatched positional arguments to raise a `TypeError` exception.

`*args` specifies that any extra positional arguments to a call (i.e., positional arguments not matching positional parameters in the function signature) get collected into a (possibly empty) tuple, bound in the call's local namespace to the name `args`. Without a positional argument collector, unmatched positional arguments raise a `TypeError` exception.

For example, here's a function that accepts any number of positional arguments and returns their sum (and demonstrates the use of an identifier other than `*args`):

```
def sum_sequence(*numbers):
```

```
    return sum(numbers)
print(sum_sequence(23, 42))        # prints: 65
```

Similarly, ***kwds_* specifies that any extra named arguments (i.e., those named arguments not explicitly specified in the signature) get collected into a (possibly empty) dictionary whose items are the names and values of those arguments, bound to the name *kwds* in the function call namespace.

For example, the following function takes a dictionary whose keys are strings and whose values are numeric, and adds given amounts to certain values:

```
def inc_dict(d, **values):
    for key, value in values.items():
        if key in d:
            d[key] += value
        else:
            d[key] = value

my_dict = {'one': 1, 'two': 2}

inc_dict(my_dict, one=3, new=42)
print(my_dict)                # prints: {'one': 4, 'two': 2, 'new':42}
```

Without a named argument collector, unmatched named arguments raise a `TypeError` exception.

Attributes of Function Objects

The **`def`** statement sets some attributes of a function object *f*. The string attribute *f*`.__name__` is the identifier that **`def`** uses as the function's name. You may rebind `__name__` to any string value, but trying to unbind it raises a `TypeError` exception. *f*`.__defaults__`, which you may freely rebind or unbind, is the tuple of default values for named parameters (empty, if the function has no named parameters).

Docstrings

Another function attribute is the *documentation string*, or *docstring*. You may use or rebind a function *f*'s docstring attribute as `f.__doc__`. When the first statement in the function body is a string literal, the compiler binds that string as the function's docstring attribute. A similar rule applies to classes and modules (see **"Class documentation strings"** and **"Module documentation strings"**). Docstrings can span multiple physical lines, so it's best to specify them in triple-quoted string literal form. For example:

```
def sum_sequence(*numbers):
    """Return the sum of multiple numerical arguments.

       The arguments are zero or more numbers.
       The result is their sum.
    """
    return sum(numbers)
```

Documentation strings should be part of any Python code you write. They play a role similar to that of comments, but they are even more useful, since they remain available at runtime (unless you run your program with **`python -OO`**, as covered in **"Command-Line Syntax and Options"**). Python's `help` function (see **"Interactive Sessions"**), development environments, and other tools can use the docstrings from function, class, and module objects to remind the programmer how to use those objects. The `doctest` module (covered in **"The doctest Module"**) makes it easy to check that sample code present in docstrings, if any, is accurate and correct, and remains so as the code and docs get edited and maintained.

To make your docstrings as useful as possible, respect a few simple conventions, as detailed in **PEP 257**. The first line of a docstring should be a concise summary of the function's purpose, starting with an uppercase letter and ending with a period. It should not mention the function's name, unless the name happens to be a natural-language word that comes naturally as part of a good, concise summary of the function's operation. Use imperative rather than descriptive form: e.g., say "Return

xyz…" rather than "Returns xyz…" If the docstring is multiline, the second line should be empty, and the following lines should form one or more paragraphs, separated by empty lines, describing the function's parameters, preconditions, return value, and side effects (if any). Further explanations, bibliographical references, and usage examples—which you should always check with `doctest`—can optionally (and often very usefully!) follow toward the end of the docstring.

Other attributes of function objects

In addition to its predefined attributes, a function object may have other arbitrary attributes. To create an attribute of a function object, bind a value to the appropriate attribute reference in an assignment statement after the **`def`** statement executes. For example, a function could count how many times it gets called:

```
def counter():
    counter.count += 1
    return counter.count
counter.count = 0
```

Note that this is *not* common usage. More often, when you want to group together some state (data) and some behavior (code), you should use the object-oriented mechanisms covered in **Chapter 4**. However, the ability to associate arbitrary attributes with a function can sometimes come in handy.

Function Annotations

Every parameter in a **`def`** clause can be *annotated* with an arbitrary expression—that is, wherever within the **`def`**'s parameter list you can use an identifier, you can alternatively use the form *`identifier:expression`*, and the expression's value becomes the *annotation* for that parameter.

You can also annotate the return value of the function, using the form `->`*`expression`* between the `)` of the **`def`** clause and the `:` that ends the **`def`**

clause; the expression's value becomes the annotation for the name `'return'`. For example:

```
>>> def f(a:'foo', b)->'bar': pass
...
>>> f.__annotations__
```

```
{'a': 'foo', 'return': 'bar'}
```

As shown in this example, the `__annotations__` attribute of the function object is a `dict` mapping each annotated identifier to the respective annotation.

You can currently, in theory, use annotations for whatever purpose you wish: Python itself does nothing with them, except construct the `__annotations__` attribute. For detailed information about annotations used for type hinting, which is now normally considered their key use, see **Chapter 5**.

The return Statement

You can use the **`return`** keyword in Python only inside a function body, and you can optionally follow it with an expression. When **`return`** executes, the function terminates, and the value of the expression is the function's result. A function returns **`None`** when it terminates by reaching the end of its body, or by executing a **`return`** statement with no expression (or by explicitly executing **`return None`**).

GOOD STYLE IN RETURN STATEMENTS

As a matter of good style, when some `return` statements in a function have an expression, then *all* `return` statements in the function should have an expression. `return None` should only ever be written explicitly to meet this style requirement. *Never* write a `return` statement without an expression at the end of a function body. Python does not enforce these stylistic conventions, but your code is clearer and more readable when you follow them.

Calling Functions

A function call is an expression with the following syntax:

```
function_object(arguments)
```

`function_object` may be any reference to a function (or other callable) object; most often, it's just the function's name. The parentheses denote the function call operation itself. *`arguments`*, in the simplest case, is a series of zero or more expressions separated by commas (`,`), giving values for the function's corresponding parameters. When the function call executes, the parameters are bound to the argument values in a new namespace, the function body executes, and the value of the function call expression is whatever the function returns. Objects created inside and returned by the function are liable to garbage collection unless the caller retains a reference to them.

DON'T FORGET THE TRAILING () TO CALL A FUNCTION

Just *mentioning* a function (or other callable object) does *not*, per se, call it. To *call* a function (or other object) without arguments, you *must* use `()` after the function's name (or other reference to the callable object).

Positional and named arguments

Arguments can be of two types. *Positional* arguments are simple expressions; *named* (also known, alas, as *keyword*[26]) arguments take the form:

```
identifier=expression
```

It is a syntax error for named arguments to precede positional ones in a function call. Zero or more positional arguments may be followed by zero or more named arguments. Each positional argument supplies the value for the parameter that corresponds to it by position (order) in the function definition. There is no requirement for positional arguments to match positional parameters, or vice versa—if there are more positional arguments than positional parameters, the additional arguments are bound by position to named parameters, if any, for all parameters preceding an argument collector in the signature. For example:

```
def f(a, b, c=23, d=42, *x):
    print(a, b, c, d, x)
f(1,2,3,4,5,6)  # prints: 1 2 3 4 (5, 6)
```

Note that it matters where in the function signature the argument collector appears (see **"Matching arguments to parameters"** for all the gory details):

```
def f(a, b, *x, c=23, d=42):
    print(a, b, x, c, d)
f(1,2,3,4,5,6)  # prints: 1 2 (3, 4, 5, 6) 23 42
```

In the absence of any named argument collector (`**name`) parameter, each argument's name must be one of the parameter names used in the function's signature.[27] The *expression* supplies the value for the parameter of that name. Many built-in functions do not accept named arguments: you must call such functions with positional arguments only. However,

functions coded in Python usually accept named as well as positional arguments, so you may call them in different ways. Positional parameters can be matched by named arguments, in the absence of matching positional arguments.

A function call must supply, via a positional or a named argument, exactly one value for each mandatory parameter, and zero or one value for each optional parameter.[28] For example:

```
def divide(divisor, dividend=94):
    return dividend // divisor
print(divide(12))                           # prints: 7
print(divide(12, 94))                       # prints: 7
print(divide(dividend=94, divisor=12))      # prints: 7
print(divide(divisor=12))                   # prints: 7
```

As you can see, the four calls to `divide` here are equivalent. You can pass named arguments for readability purposes whenever you think that identifying the role of each argument and controlling the order of arguments enhances your code's clarity.

A common use of named arguments is to bind some optional parameters to specific values, while letting other optional parameters take default values:

```
def f(middle, begin='init', end='finis'):
    return begin+middle+end
print(f('tini', end=''))                    # prints: inittini
```

With the named argument `end=''`, the caller specifies a value (the empty string `''`) for `f`'s third parameter, `end`, and still lets *f*'s second parameter, `begin`, use its default value, the string `'init'`. You may pass the arguments as positional even when parameters are named; for example, with the preceding function:

```
print(f('a','c','t'))                    # prints: cat
```

At the end of the arguments in a function call, you may optionally use either or both of the special forms *`*seq`* and *`**dct`*. If both forms are present, the form with two asterisks must be last. *`*seq`* passes the items of iterable *`seq`* to the function as positional arguments (after the normal positional arguments, if any, that the call gives with the usual syntax). *`seq`* may be any iterable. *`**dct`* passes the items of *`dct`* to the function as named arguments, where *`dct`* must be a mapping whose keys are all strings. Each item's key is a parameter name, and the item's value is the argument's value.

You may want to pass an argument of the form *`*seq`* or *`**dct`* when the parameters use similar forms, as discussed in **"Parameters"**. For example, using the function `sum_sequence` defined in that section (and shown again here), you may want to print the sum of all the values in the dictionary *`d`*. This is easy with *`*seq`*:

```
def sum_sequence(*numbers):
    return sum(numbers)

d = {'a': 1, 'b': 100, 'c': 1000}
print(sum_sequence(*d.values()))
```

(Of course, `print(sum(d.values()))` would be simpler and more direct.)

A function call may have zero or more occurrences of *`*seq`* and/or *`**dct`*, as specified in **PEP 448**. You may even pass *`*seq`* or *`**dct`* when calling a function that does not use the corresponding form in its signature. In that case, you must ensure that the iterable *`seq`* has the right number of items, or that the dictionary *`dct`* uses the right identifier strings as keys; otherwise, the call raises an exception. As noted in the following section, a positional argument *cannot* match a "keyword-only" parameter; only a named argument, explicit or passed via *`**kwargs`*, can do that.

“Keyword-only” parameters

Parameters after a positional argument collector (**name* or 3.8+ *) in the function’s signature are known as *keyword-only parameters*: the corresponding arguments, if any, *must* be named arguments. In the absence of any match by name, such a parameter is bound to its default value, as set at function definition time.

Keyword-only parameters can be either positional or named. However, you *must* pass them as named arguments, not as positional ones. It’s more usual and readable to have simple identifiers, if any, at the start of the keyword-only parameter specifications, and *`identifier=default`* forms, if any, following them, though this is not a requirement of the Python language.

Functions requiring keyword-only parameter specifications *without* collecting “surplus” positional arguments indicate the start of the keyword-only parameter specifications with a dummy parameter consisting solely of an asterisk (*), to which no argument corresponds. For example:

```
def f(a, *, b, c=56):  # b and c are keyword only
    return a, b, c
f(12, b=34)  # Returns (12, 34, 56) c is optional, it has a default
f(12)        # Raises a TypeError exception, since you didn't pass b
# Error message is: missing 1 required keyword-only argument: 'b'
```

If you also specify the special form ***kwds*, it must come at the end of the parameter list (after the keyword-only parameter specifications, if any). For example:

```
def g(x, *a, b=23, **k):  # b is keyword only
    return x, a, b, k
g(1, 2, 3, c=99)          # Returns (1, (2, 3), 23, {'c': 99})
```

Matching arguments to parameters

A call *must* provide an argument for all positional parameters, and *may* do so for named parameters that are not keyword only.

The matching proceeds as follows:

1. Arguments of the form `*expression` are internally replaced by a sequence of positional arguments obtained by iterating over `expression`.
2. Arguments of the form `**expression` are internally replaced by a sequence of keyword arguments whose names and values are obtained by iterating over `expression`'s `items()`.
3. Say that the function has *N* positional parameters and the call has *M* positional arguments:
    1. When $M \leq N$, bind all the positional arguments to the first *M* positional parameter names; the remaining positional parameters, if any, *must* be matched by named arguments.
    2. When $M > N$, bind the remaining positional arguments to named parameters *in the order in which they appear in the signature*. This process terminates in one of three ways:
        a. All positional arguments have been bound.
        b. The next item in the signature is a `*` argument collector: the interpreter raises a `TypeError` exception.
        c. The next item in the signature is a `*name` argument collector: the remaining positional arguments are collected in a tuple that is then bound to `name` in the function call namespace.
4. The named arguments are then matched, in the order of the arguments' occurrences in the call, by name with the parameters—both positional and named. Attempts to rebind an already bound parameter name raise a `TypeError` exception.
5. If unmatched named arguments remain at this stage:
    1. When the function signature includes a `**name` collector, the interpreter creates a dictionary with key/value pairs (`argument's_name, argument's_value`), and binds it to `name` in the function call namespace.

2. In the absence of such an argument collector, Python raises a `TypeError` exception.

6. Any remaining unmatched named parameters are bound to their default values.

At this point, the function call namespace is fully populated, and the interpreter executes the function's body using that "call namespace" as the local namespace for the function.

The semantics of argument passing

In traditional terms, all argument passing in Python is *by value* (although, in modern terminology, to say that argument passing is *by object reference* is more precise and accurate; check out the synonym ***call by sharing***). When you pass a variable as an argument, Python passes to the function the object (value) to which the variable currently refers (not "the variable itself"!), binding this object to the parameter name in the function call namespace. Thus, a function *cannot* rebind the caller's variables. Passing a mutable object as an argument, however, allows the function to make changes to that object, because Python passes a reference to the object itself, not a copy. Rebinding a variable and mutating an object are totally disjoint concepts. For example:

```
def f(x, y):
    x = 23
    y.append(42)
a = 77
b = [99]
f(a, b)
print(a, b)             # prints: 77 [99, 42]
```

`print` shows that `a` is still bound to `77`. Function `f`'s rebinding of its parameter `x` to `23` has no effect on `f`'s caller, nor, in particular, on the binding of the caller's variable that happened to be used to pass `77` as the parameter's value. However, `print` also shows that `b` is now bound to `[99, 42]`. `b` is still bound to the same list object as before the call, but `f`

has appended `42` to that list object, mutating it. In neither case has `f` altered the caller's bindings, nor can `f` alter the number `77`, since numbers are immutable. `f` can alter a list object, though, since list objects are mutable.

Namespaces

A function's parameters, plus any names that are bound (by assignment or by other binding statements, such as **`def`**) in the function body, make up the function's *local namespace*, also known as its *local scope*. Each of these variables is known as a *local variable* of the function.

Variables that are not local are known as *global variables* (in the absence of nested function definitions, which we'll discuss shortly). Global variables are attributes of the module object, as covered in **"Attributes of module objects"**. Whenever a function's local variable has the same name as a global variable, that name, within the function body, refers to the local variable, not the global one. We express this by saying that the local variable *hides* the global variable of the same name throughout the function body.

The global statement

By default, any variable bound in a function body is local to the function. If a function needs to bind or rebind some global variables (*not* best practice!), the first statement of the function's body must be:

```
global identifiers
```

where *`identifiers`* is one or more identifiers separated by commas (`,`). The identifiers listed in a **`global`** statement refer to the global variables (i.e., attributes of the module object) that the function needs to bind or rebind. For example, the function `counter` that we saw in **"Other attributes of function objects"** could be implemented using **`global`** and a global variable, rather than an attribute of the function object:

```
_count = 0
def counter():
    global _count
    _count += 1
    return _count
```

Without the **`global`** statement, the `counter` function would raise an `UnboundLocalError` exception when called, because `_count` would then be an uninitialized (unbound) local variable. While the **`global`** statement enables this kind of programming, this style is inelegant and ill-advised. As we mentioned earlier, when you want to group together some state and some behavior, the object-oriented mechanisms covered in **Chapter 4** are usually best.

ESCHEW GLOBAL

Never use **`global`** if the function body just *uses* a global variable (including mutating the object bound to that variable, when the object is mutable). Use a **`global`** statement only if the function body *rebinds* a global variable (generally by assigning to the variable's name). As a matter of style, don't use **`global`** unless it's strictly necessary, as its presence causes readers of your program to assume the statement is there for some useful purpose. Never use **`global`** except as the first statement in a function body.

Nested functions and nested scopes

A **`def`** statement within a function body defines a *nested function*, and the function whose body includes the **`def`** is known as an *outer function* to the nested one. Code in a nested function's body may access (but *not* rebind) local variables of an outer function, also known as *free variables* of the nested function.

The simplest way to let a nested function access a value is often not to rely on nested scopes, but rather to pass that value explicitly as one of the function's arguments. If need be, you can bind the argument's value at

nested-function `def` time: just use the value as the default for an optional argument. For example:

```
def percent1(a, b, c):
    def pc(x, total=a+b+c):
        return (x*100.0) / total
    print('Percentages are:', pc(a), pc(b), pc(c))
```

Here's the same functionality using nested scopes:

```
def percent2(a, b, c):
    def pc(x):
        return (x*100.0) / (a+b+c)
    print('Percentages are:', pc(a), pc(b), pc(c))
```

In this specific case, `percent1` has one tiny advantage: the computation of `a+b+c` happens only once, while `percent2`'s inner function `pc` repeats the computation three times. However, when the outer function rebinds local variables between calls to the nested function, repeating the computation can be necessary: be aware of both approaches, and choose the appropriate one on a case-by-case basis.

A nested function that accesses values from local variables of "outer" (containing) functions is also known as a *closure*. The following example shows how to build a closure:

```
def make_adder(augend):
    def add(addend):
        return addend+augend
    return add
add5 = make_adder(5)
add9 = make_adder(9)

print(add5(100))   # prints: 105
print(add9(100))   # prints: 109
```

Closures are sometimes an exception to the general rule that the object-oriented mechanisms covered in the next chapter are the best way to bundle together data and code. When you specifically need to construct callable objects, with some parameters fixed at object construction time, closures can be simpler and more direct than classes. For example, the result of `make_adder(7)` is a function that accepts a single argument and returns `7` plus that argument. An outer function that returns a closure is a "factory" for members of a family of functions distinguished by some parameters, such as the value of the argument `augend` in the previous example, which may often help you avoid code duplication.

The **`nonlocal`** keyword acts similarly to **`global`**, but it refers to a name in the namespace of some lexically surrounding function. When it occurs in a function definition nested several levels deep (a rarely needed structure!), the compiler searches the namespace of the most deeply nested containing function, then the function containing that one, and so on, until the name is found or there are no further containing functions, in which case the compiler raises an error (a global name, if any, does not match).

Here's a nested functions approach to the "counter" functionality we implemented in previous sections using a function attribute, then a global variable:

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

c1 = make_counter()
c2 = make_counter()
print(c1(), c1(), c1())     # prints: 1 2 3
print(c2(), c2())           # prints: 1 2
print(c1(), c2(), c1())     # prints: 4 3 5
```

A key advantage of this approach versus the previous ones is that these two nested functions, just like an object-oriented approach would, let you make independent counters—here `c1` and `c2`. Each closure keeps its own state and doesn't interfere with the other one.

lambda Expressions

If a function body is a single **`return`** *`expression`* statement, you may (*very* optionally!) choose to replace the function with the special **`lambda`** expression form:

```
lambda parameters: expression
```

A **`lambda`** expression is the anonymous equivalent of a normal function whose body is a single **`return`** statement. The **`lambda`** syntax does not use the **`return`** keyword. You can use a **`lambda`** expression wherever you could use a reference to a function. **`lambda`** can sometimes be handy when you want to use an *extremely simple* function as an argument or return value.

Here's an example that uses a **`lambda`** expression as an argument to the built-in `sorted` function (covered in **Table 8-2**):

```
a_list = [-2, -1, 0, 1, 2]
sorted(a_list, key=lambda x: x * x)  # returns: [0, -1, 1, -2, 2]
```

Alternatively, you can always use a local **`def`** statement to give the function object a name, then use this name as an argument or return value. Here's the same `sorted` example using a local **`def`** statement:

```
a_list = [-2, -1, 0, 1, 2]
def square(value):
```

```
    return value * value
sorted(a_list, key=square)          # returns: [0, -1, 1, -2, 2]
```

While **`lambda`** can at times be handy, **`def`** is usually better: it's more general and helps you make your code more readable, since you can choose a clear name for the function.

Generators

When the body of a function contains one or more occurrences of the keyword **`yield`**, the function is known as a *generator*, or more precisely a *generator function*. When you call a generator, the function body does not execute. Instead, the generator function returns a special iterator object, known as a *generator object* (sometimes, quite confusingly, also called just "a generator"), wrapping the function body, its local variables (including parameters), and the current point of execution (initially, the start of the function).

When you (implicitly or explicitly) call `next` on a generator object, the function body executes from the current point up to the next **`yield`**, which takes the form:

```
yield expression
```

A bare **`yield`** without the expression is also legal, and equivalent to **`yield None`**. When **`yield`** executes, the function execution is "frozen," preserving the current point of execution and local variables, and the expression following **`yield`** becomes the result of `next`. When you call `next` again, execution of the function body resumes where it left off, again up to the next **`yield`**. When the function body ends, or executes a **`return`** statement, the iterator raises a `StopIteration` exception to indicate that the iteration is finished. The expression after **`return`**, if any, is the argument to the `StopIteration` exception.

yield is an expression, not a statement. When you call *g*.send(*value*) on a generator object *g*, the value of **yield** is *value*; when you call next(*g*), the value of **yield** is **None**. We'll talk more about this shortly: it's an elementary building block for implementing **coroutines** in Python.

A generator function is often a handy way to build an iterator. Since the most common way to use an iterator is to loop on it with a **for** statement, you typically call a generator like this (with the call to next being implicit in the **for** statement):

```
for avariable in somegenerator(arguments):
```

For example, say that you want a sequence of numbers counting up from 1 to *N* and then down to 1 again. A generator can help:

```
def updown(N):
    for x in range(1, N):
        yield x
    for x in range(N, 0, -1):
        yield x
for i in updown(3):
    print(i)              # prints: 1 2 3 2 1
```

Here is a generator that works somewhat like built-in range, but returns an iterator on floating-point values rather than on integers:

```
def frange(start, stop, stride=1.0):
    start = float(start)  # force all yielded values to be floats
    while start < stop:
        yield start
        start += stride
```

This example is only *somewhat* like range because, for simplicity, it makes the arguments start and stop mandatory, and assumes that stride is

positive.

Generator functions are more flexible than functions that return lists. A generator function may return an unbounded iterator, meaning one that yields an infinite stream of results (to use only in loops that terminate by other means, e.g., via a conditionally executed **break** statement). Further, a generator object iterator performs *lazy evaluation*: the iterator can compute each successive item only when and if needed, "just in time," while the equivalent function does all computations in advance and may require large amounts of memory to hold the results list. Therefore, if all you need is the ability to iterate on a computed sequence, it is usually best to compute the sequence in a generator object, rather than in a function returning a list. If the caller needs a list of all the items produced by some bounded generator object built by *g*(*arguments*), the caller can simply use the following code to explicitly request that Python build a list:

```
resulting_list = list(g(arguments))
```

yield from

To improve execution efficiency and clarity when multiple levels of iteration are yielding values, you can use the form **yield from** *expression*, where *expression* is iterable. This yields the values from *expression* one at a time into the calling environment, avoiding the need to **yield** repeatedly. We can thus simplify the updown generator we defined earlier:

```
def updown(N):
    yield from range(1, N)
    yield from range(N, 0, -1)
for i in updown(3):
    print(i)              # prints: 1 2 3 2 1
```

Moreover, using **yield from** lets you use generators as *coroutines*, discussed next.

Generators as near-coroutines

Generators are further enhanced with the possibility of receiving a value (or an exception) back from the caller as each **`yield`** executes. This lets generators implement **coroutines**, as explained in **PEP 342**. When a generator object resumes (i.e., you call `next` on it), the corresponding **`yield`**'s value is **`None`**. To pass a value *x* into some generator object *g* (so that *g* receives *x* as the value of the **`yield`** on which it's suspended), instead of calling `next(g)`, call `g.send(x)` (`g.send(None)` is just like `next(g)`).

Other enhancements to generators regard exceptions: we cover them in **"Generators and Exceptions"**.

Generator expressions

Python offers an even simpler way to code particularly simple generators: *generator expressions*, commonly known as *genexps*. The syntax of a genexp is just like that of a list comprehension (as covered in **"List comprehensions"**), except that a genexp is within parentheses (`()`) instead of brackets (`[]`). The semantics of a genexp are the same as those of the corresponding list comprehension, except that a genexp produces an iterator yielding one item at a time, while a list comprehension produces a list of all results in memory (therefore, using a genexp, when appropriate, saves memory). For example, to sum the squares of all single-digit integers, you could code `sum([x*x for x in range(10)])`, but you can express this better as `sum(x*x for x in range(10))` (just the same, but omitting the brackets): you get the same result but consume less memory. The parentheses that indicate the function call also do "double duty" and enclose the genexp. Parentheses are, however, required when the genexp is not the sole argument. Additional parentheses don't hurt, but are usually best omitted, for clarity.

WARNING: DON'T ITERATE OVER A GENERATOR MULTIPLE TIMES

A limitation of generators and generator expressions is that you can iterate over them only once. Calling `next` on a generator that has been consumed will just raise `StopIteration` again, which most functions will take as an indication that the generator returns no values. If your code is not careful about reusing a consumed generator, this can introduce bugs:

```
# create a generator and list its items and their sum
squares = (x*x for x in range(5))
print(list(squares))  # prints [0, 1, 4, 9, 16]
print(sum(squares))   # Bug! Prints 0
```

You can write code to guard against accidentally iterating over a consumed generator by using a class to wrap the generator, like the following:

```
class ConsumedGeneratorError(Exception):
    """Raised if a generator is accessed after
       already consumed.
    """

class StrictGenerator:
    """Wrapper for generator that will only permit it
       to be consumed once. Additional accesses will
       raise ConsumedGeneratorError.
    """
    def __init__(self, gen):
        self._gen = gen
        self._gen_consumed = False
    def __iter__(self):
        return self
    def __next__(self):
        try:
            return next(self._gen)
        except StopIteration:
            if self._gen_consumed:
                raise ConsumedGeneratorError() from None
            self._gen_consumed = True
            raise
```

Now an erroneous reuse of a generator will raise an exception:

```
squares = StrictGenerator(x*x for x in range(5))
print(list(squares))  # prints: [0, 1, 4, 9, 16]
print(sum(squares))   # raises ConsumedGeneratorError
```

Recursion

Python supports recursion (i.e., a Python function can call itself, directly or indirectly), but there is a limit to how deep the recursion can go. By default, Python interrupts recursion and raises a `RecursionLimitExceeded` exception (covered in **"Standard Exception Classes"**) when it detects that recursion has exceeded a depth of 1,000. You can change this default recursion limit by calling the `setrecursionlimit` function in the module `sys`, covered in **Table 8-3**.

Note that changing the recursion limit does not give you unlimited recursion. The absolute maximum limit depends on the platform on which your program is running, and particularly on the underlying operating system and C runtime library, but it's typically a few thousand levels. If recursive calls get too deep, your program crashes. Such runaway recursion, after a call to `setrecursionlimit` that exceeds the platform's capabilities, is one of the few things that can cause a Python program to crash—really crash, hard, without the usual safety net of Python's exception mechanism. Therefore, beware of "fixing" a program that is getting `RecursionLimitExceeded` exceptions by raising the recursion limit with `setrecursionlimit`. While this *is* a valid technique, most often you're better advised to look for ways to remove the recursion unless you are confident you've been able to limit the depth of recursion that your program needs.

Readers who are familiar with Lisp, Scheme, or functional programming languages must in particular be aware that Python does *not* implement the optimization of *tail-call elimination*, which is so crucial in those languages. In Python, any call, recursive or not, has the same "cost" in terms of both time and memory space, dependent only on the number of arguments: the cost does not change, whether the call is a "tail call" (meaning

that it's the last operation that the caller executes) or not. This makes recursion removal even more important.

For example, consider a classic use for recursion: walking a binary tree. Suppose you represent a binary tree structure as nodes, where each node is a three-item tuple `(payload, left, right)`, and `left` and `right` are either similar tuples or `None`, representing the left-side and right-side descendants. A simple example might be `(23, (42, (5, None, None), (55, None, None)), (94, None, None))` to represent the tree shown in Figure 3-1.

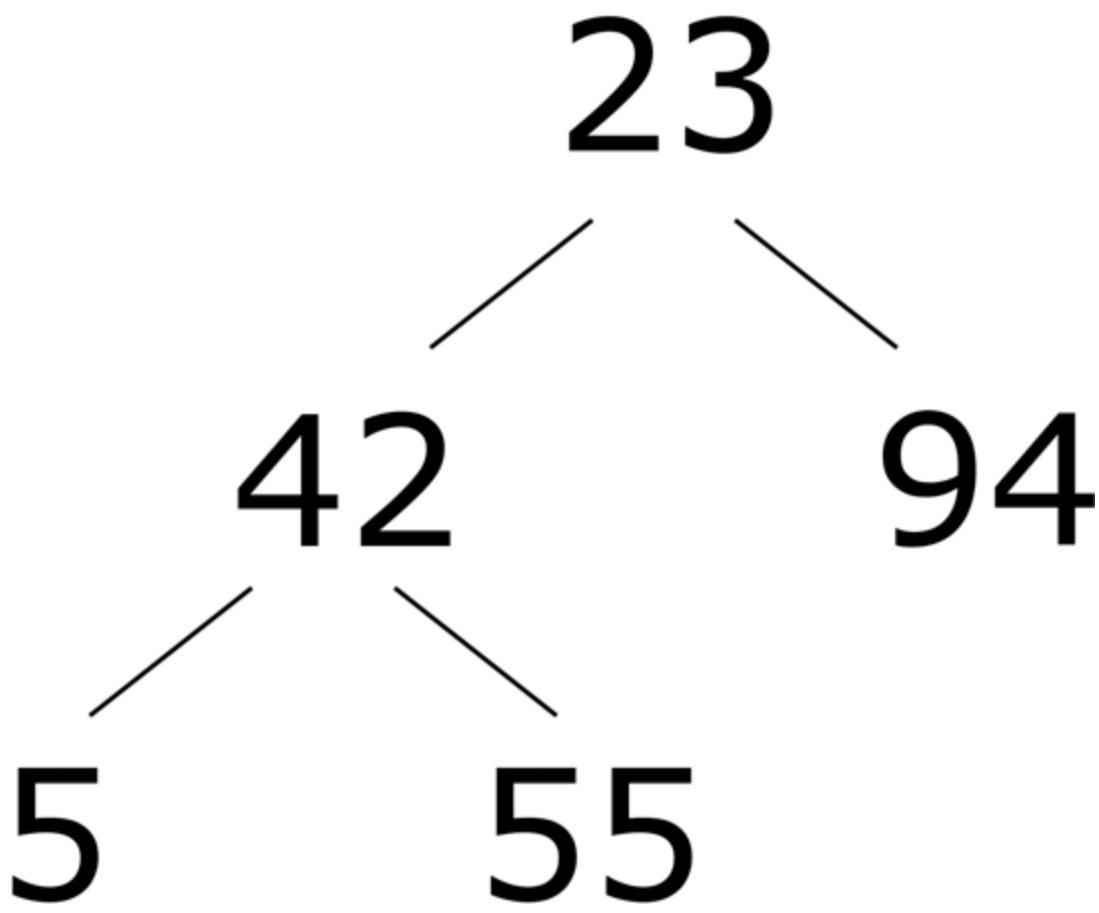

Figure 3-1. An example of a binary tree

To write a generator function that, given the root of such a tree, "walks" the tree, yielding each payload in top-down order, the simplest approach is recursion:

```
def rec(t):
    yield t[0]
    for i in (1, 2):
```

```
            if t[i] is not None:
                yield from rec(t[i])
```

But if a tree is very deep, recursion can become a problem. To remove recursion, we can handle our own stack—a list used in last-in, first-out (LIFO) fashion, thanks to its `append` and `pop` methods. To wit:

```
def norec(t):
    stack = [t]
    while stack:
        t = stack.pop()
        yield t[0]
        for i in (2, 1):
            if t[i] is not None:
                stack.append(t[i])
```

The only small issue to be careful about, to keep exactly the same order of **`yield`**s as `rec`, is switching the `(1, 2)` index order in which to examine descendants to `(2, 1)`, adjusting to the "reversed" (last-in, first-out) behavior of `stack`.

1 Identifiers referring to constants are all uppercase, by convention.

2 Control characters include nonprinting characters such as `\t` (tab) and `\n` (newline), both of which count as whitespace, and others such as `\a` (alarm, aka "beep") and `\b` (backspace), which are not whitespace.

3 "Container displays," per the **online docs** (e.g., `list_display`), but specifically ones with literal items.

4 There's also a `bytearray` object, covered shortly, which is a `bytes`-like "string" that *is* mutable.

5 This syntax is sometimes called a "tuple display."

6 This syntax is sometimes called a "list display."

7 This syntax is sometimes called a “set display.”

8 Each specific mapping type may put some constraints on the type of keys it accepts: in particular, dictionaries only accept hashable keys.

9 This syntax is sometimes called a “dictionary display.”

10 See **“Shape, indexing, and slicing”**.

11 Strictly speaking, *almost* any: NumPy arrays, covered in **Chapter 16**, are an exception.

12 With exactly the same exception of NumPy arrays.

13 Sometimes referred to as the *ternary* operator, as it is so called in C (Python’s original implementation language).

14 This is not, strictly speaking, the “coercion” you observe in other languages; however, among built-in numeric types, it produces pretty much the same effect.

15 Hence the upper limit of 36 for the radix: 10 numeric digits plus 26 alphabetic characters.

16 The second item of `divmod`’s result, just like the result of `%`, is the *remainder*, not the *modulo*, despite the function’s misleading name. The difference matters when the divisor is negative. In some other languages, such as C# and JavaScript, the result of a `%` operator is, in fact, the modulo; in others yet, such as C and C++, it is machine-dependent whether the result is the modulo or the remainder when either operand is negative. In Python, it’s the remainder.

17 Timsort has the distinction of being the only sorting algorithm mentioned by the US Supreme Court, specifically in the case of **Oracle v. Google**.

18 Except, as already noted, a NumPy array with more than one element.

19 It is notable that the **`match`** statement specifically excludes matching values of type `str`, `bytes`, and `bytearray` with *sequence* patterns.

20 Indeed, the syntax notation used in the Python online documentation required, and got, updates to concisely describe some of Python’s more recent syntax additions.

21 Although comparing `float` or `complex` numbers for exact equality is often dubious practice.

22 For this unique use case, it's common to break the normal style conventions about starting class names with an uppercase letter and avoiding using semicolons to stash multiple assignments within one line; however, the authors haven't yet found a style guide that blesses this peculiar, rather new usage.

23 And its subclasses, for example, `collections.defaultdict`.

24 Except that the loop variables' scope is within the comprehension only, different from the way scoping works in a **`for`** statement.

25 In that paper, Knuth also first proposed using "devices like indentation, rather than delimiters" to express program structure—just as Python does!

26 "Alas" because they have nothing to do with Python keywords, so the terminology is confusing; if you use an actual Python keyword to name a named parameter, that raises `SyntaxError`.

27 Python developers introduced positional-only arguments when they realized that parameters to many built-in functions effectively had no valid names as far as the interpreter was concerned.

28 An "optional parameter" being one for which the function's signature supplies a default value.