Learning Python – Mark Lutz

 Python is a powerful multiparadigm computer programming language, optimized for programmer productivity, code readability, and software quality.

Python is a popular open source programming language used for both standalone programs and scripting applications in a wide variety of domains.

Programming Python, 4th Edition

Powerful Object-Oriented Programming

By [Mark Lutz](http://www.oreillynet.com/pub/au/446)

Publisher: [O'Reilly Media](https://learning.oreilly.com/library/publisher/oreilly-media-inc/)

Release Date: December 2010

Pages: 1632

<http://shop.oreilly.com/product/9780596158118.do>

--

Python Pocket Reference, 3rd Edition

By [Mark Lutz](http://www.oreillynet.com/pub/au/446)

**Publisher:** [O'Reilly Media](https://learning.oreilly.com/library/publisher/oreilly-media-inc/)

**Release Date:** June 2009

**Pages:** 160

<http://shop.oreilly.com/product/9780596009403.do>

--

Python’s classes are largely just packages of functions that process built-in types.

Functions turn out to be a simple way to package code for reuse and avoid code redundancy.

Python modules let you organize statements and functions into larger components.

Why do people use Python:

1. Softwre quality
2. Developer productivity
3. Program portability
4. Support libraries
5. Component integration

Python is deliberately optimized for speed of development—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools.

It is commonly defined as an object-oriented scripting language

Python is probably better known as a general-purpose programming language that blends procedural, functional, and object-oriented paradigms—a statement that captures the richness and scope of today’s Python.

# What’s the Downside?

After using it for 21 years, writing about it for 18, and teaching it for 16, I’ve found that the only significant universal downside to Python is that, as currently implemented, its execution speed may not always be as fast as that of fully compiled and lower-level languages such as C and C++. Though relatively rare today, for some tasks you may still occasionally need to get “closer to the iron” by using lower-level languages such as these that are more directly mapped to the underlying hardware architecture.

The standard implementations of Python today compile (i.e., translate) source code statements to an intermediate format known as byte code and then interpret the byte code. Byte code provides portability, as it is a platform-independent format.

As a general-purpose language, Python’s roles are virtually unlimited: you can use it for everything from website development and gaming to robotics and spacecraft control.

Python’s standard library comes with POSIX bindings and support for all the usual OS tools: environment variables, files, sockets, pipes, processes, multiple threads, regular expression pattern matching, command-line arguments, standard stream interfaces, shell-command launchers, filename expansion, zip file utilities, XML and JSON parsers, CSV file handlers, and more.

Python comes with a standard object-oriented interface to the Tk GUI API called tkinter (Tkinter in 2.X) that allows Python programs to implement portable GUIs with a native look and feel. A free extension package, PMW, adds advanced widgets to the tkinter toolkit. Higher-level toolkits such as Dabo are built on top of base APIs such as wxPython and tkinter.

Full-blown web development framework packages for Python, such as Django, TurboGears, web2py, Pylons, Zope, and WebWare, support quick construction of full-featured and production-quality websites with Python. Python also has moved into cloud computing, with App Engine

The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use.

Python programs are automatically compiled to portable byte code, which runs the same on any platform with a compatible version of Python installed

As a preview, here are some of the main things you’ll find in Python’s toolbox:

Dynamic typing

Python keeps track of the kinds of objects your program uses when it runs; it doesn’t require complicated type and size declarations in your code. In fact, as you’ll see in [Chapter 6](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch06.html#the_dynamic_typing_interlude), there is no such thing as a type or variable declaration anywhere in Python. Because Python code does not constrain data types, it is also usually automatically applicable to a whole range of objects.

Automatic memory management

Python automatically allocates objects and reclaims (“garbage collects”) them when they are no longer used, and most can grow and shrink on demand. As you’ll learn, Python keeps track of low-level memory details so you don’t have to.

Programming-in-the-large support

For building larger systems, Python includes tools such as modules, classes, and exceptions. These tools allow you to organize systems into components, use OOP to reuse and customize code, and handle events and errors gracefully. Python’s functional programming tools, described earlier, provide additional ways to meet many of the same goals.

Built-in object types

Python provides commonly used data structures such as lists, dictionaries, and strings as intrinsic parts of the language; as you’ll see, they’re both flexible and easy to use. For instance, built-in objects can grow and shrink on demand, can be arbitrarily nested to represent complex information, and more.

Built-in tools

To process all those object types, Python comes with powerful and standard operations, including concatenation (joining collections), slicing (extracting sections), sorting, mapping, and more.

Library utilities

For more specific tasks, Python also comes with a large collection of precoded library tools that support everything from regular expression matching to networking. Once you learn the language itself, Python’s library tools are where much of the application-level action occurs.

Third-party utilities

Because Python is open source, developers are encouraged to contribute precoded tools that support tasks beyond those supported by its built-ins; on the Web, you’ll find free support for COM, imaging, numeric programming, XML, database access, and much more.

‘import this’ triggers an Easter egg inside Python that displays some of the design philosophies underlying the language.

Python is engineering, not art like Perl.

An **interpreter** is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

When installing Python in the system, it generates a standard library and an interpreter at minimum.

By convention, Python program files are given names that end in .py; technically, this naming scheme is required only for files that are “imported”

When you instruct Python to run your script, there are a few steps that Python carries out before your code actually starts crunching away. Specifically, it’s first compiled to something called “byte code” and then routed to something called a “virtual machine.”

byte code. Compilation is simply a translation step, and byte code is a lower-level, platform-independent representation of your source code.

(“.pyc” means compiled “.py” source). In 3.2 and later, Python instead saves its .pyc byte code files in a subdirectory named \_\_pycache\_\_ located in the directory where your source files reside, and in files whose names identify the Python version that created them (e.g., script.cpython-33.pyc).

Finally, keep in mind that byte code is saved in files only for files that are imported, not for the top-level files of a program that are only run as scripts (strictly speaking, it’s an import optimization). Byte code is a Python-specific representation.

the PVM is just a big code loop that iterates through your byte code instructions, one by one, to carry out their operations. The PVM is the runtime engine of Python; it’s always present as part of the Python system, and it’s the component that truly runs your scripts.

The eval and exec built-ins, for instance, accept and run strings containing Python program code. This structure is also why Python lends itself to product customization—because Python code can be changed on the fly, users can modify the Python parts of a system onsite without needing to have or compile the entire system’s code.

There are at least five implementations of the Python language—CPython, Jython, IronPython, Stackless, and PyPy.

Other potential candidates here include the Cython and Shed Skin systems, but they are discussed later as optimization tools because they do not implement the standard Python language (the former is a Python/C mix, and the latter is implicitly statically typed).

The original, and standard, implementation of Python is usually called CPython. This name comes from the fact that it is coded in portable ANSI C language code.

Jython consists of Java classes that compile Python source code to Java byte code and then route the resulting byte code to the Java Virtual Machine (JVM). Because Jython is slower and less robust than CPython, though, it is usually seen as a tool of interest primarily to Java developers looking for a scripting language to serve as a frontend to Java code.

The Stackless Python system is an enhanced version and reimplementation of the standard CPython language oriented toward concurrency. Because it does not save state on the C language call stack, Stackless Python can make Python easier to port to small stack architectures, provides efficient multiprocessing options, and fosters novel programming structures such as coroutines.

The *PyPy* system is another standard CPython reimplementation, focused on *performance*. It provides a fast Python implementation with a *JIT* (just-in-time) compiler, provides tools for a “sandbox” model that can run untrusted code in a secure environment, and by default includes support for the prior section’s *Stackless* Python systems and its microthreads to support massive concurrency.

PyPy is the successor to the original *Psyco* JIT.

A JIT is really just an extension to the PVM—that translates portions of your byte code all the way to binary machine code for faster execution. It does this as your program is running, not in a prerun compile step, and is able to create type-specific machine code for the dynamic Python language by keeping track of the data types of the objects your program processes. By replacing portions of your byte code this way, your program runs faster and faster as it is executing. If you write CPU-intensive code, PyPy deserves your attention.

The Cython system (based on work done by the Pyrex project) is a hybrid language that combines Python code with the ability to call C functions and use C type declarations for variables, parameters, and class attributes. Cython code can be compiled to C code that uses the Python/C API, which may then be compiled completely.

Shed Skin is an emerging system that takes a different approach to Python program execution—it attempts to translate Python source code to C++ code, which your computer’s C++ compiler then compiles to machine code.

The Psyco system is not another Python implementation, but rather a component that extends the byte code execution model to make programs run faster.

With the help of third-party tools that you can fetch off the Web, it is possible to turn your Python programs into true executables, known as frozen binaries in the Python world. These programs can be run without requiring a Python installation.

Frozen binaries bundle together the byte code of your program files, along with the PVM (interpreter) and any Python support files your program needs, into a single package. There are some variations on this theme, but the end result can be a single binary executable program (e.g., an .exe file on Windows) that can easily be shipped to customers.

py2exe for Windows only, but with broad Windows support; PyInstaller, which is similar to py2exe but also works on Linux and Mac OS X and is capable of generating self-installing binaries;

Assuming the interpreter is installed as an executable program on your system, the most platform-neutral way to start an interactive interpreter session is usually just to type **python** at your operating system’s prompt, without any arguments. On Windows, a Ctrl-Z gets you out of this session; on Unix, try Ctrl-D instead.

In Python \* means multiply for numbers, but repeat for strings

Bear in mind that the interactive prompt runs just one statement at a time: you must press Enter twice to run a loop or other multiline statement before you can type the next statement.

To save programs permanently, you need to write your code in files, which are usually known as modules. Modules are simply text files containing Python statements. Python executes all the code in a module file from top to bottom each time you run the file.

a program is considered to be a series of precoded statements stored in a file for repeated execution. Module files that are run directly are also sometimes called scripts—an informal term usually meaning a top-level program file. Some reserve the term “module” for a file imported from another file, and “script” for the main file of a program.

print statements are not required in an interactive session, since Python automatically echoes expression results.

Python uses Windows filename associations to automatically register itself to be the program that opens Python program files when they are clicked.

The contents of a module are made available to the outside world through its attributes.

Python 3.X moved the reload built-in function to the imp standard library module. It still reloads files as before, but you must import it in order to use it. In 3.X, run an import imp and use imp.reload(M), or run a from imp import reload and use reload(M), as shown here.

a module is mostly just a package of variable names, known as a namespace, and the names within that package are called attributes. An attribute is simply a variable name that is attached to a specific object

Fetch a list of all the names available inside a module. Use dir(modulename)

from copies variables from one file to another, it can cause same-named variables in the importing file to be overwritten, and won’t warn you if it does.

reloads aren’t transitive—reloading a module reloads that module only, not any modules it may import.

**exec(open('script1.py').read())**

IDLE is usually referred to as an integrated development environment (IDE), because it binds together various development tasks into a single view.

Eclipse is an advanced open source IDE GUI. Originally developed as a Java IDE, Eclipse also supports Python development when you install the PyDev (or a similar) plug-in.

Python comes with a source code debugger named pdb, available as a module in Python’s standard library. Pdb.pm()

Introducing Python object types:

in Python we do things with stuff.1 “Things” take the form of operations like addition and concatenation, and “stuff” refers to the objects on which we perform those operations.

Somewhat more formally, in Python, data takes the form of objects—either built-in objects that Python provides, or objects we create using Python classes or external language tools such as C extension libraries.

objects are essentially just pieces of memory, with values and sets of associated operations.

Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. Expressions create and process objects.

Traditional introductions to programming often stress its three pillars of sequence (“Do this, then that”), selection (“Do this if that is true”), and repetition (“Do this many times”). Python has tools in all three categories, along with some for definition—of functions and classes.

Because Python provides powerful object types as an intrinsic part of the language, there’s usually no need to code object implementations before you start solving problems.

literals—that is, the expressions that generate these objects

| Built-in objects preview | |
| --- | --- |
| **Object type** | **Example literals/creation** |
| Numbers | 1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction() |
| Strings | 'spam', "Bob's", b'a\x01c', u'sp\xc4m' |
| Lists | [1, [2, 'three'], 4.5], list(range(10)) |
| Dictionaries | {'food': 'spam', 'taste': 'yum'}, dict(hours=10) |
| Tuples | (1, 'spam', 4, 'U'), tuple('spam'), namedtuple |
| Files | open('eggs.txt'), open(r'C:\ham.bin', 'wb') |
| Sets | set('abc'), {'a', 'b', 'c'} |
| Other core types | Booleans, types, None |
| Program unit types | Functions, modules, classes |
| Implementation-related types | Compiled code, stack tracebacks |

there are no type declarations in Python, the syntax of the expressions you run determines the types of objects you create and use.

it is also strongly typed, a constraint that means you can perform on an object only operations that are valid for its type.

Numbers: Python’s core objects set includes the usual suspects: integers that have no fractional part, floating-point numbers that do, and more exotic types—complex numbers with imaginary parts, decimals with fixed precision, rationals with numerator and denominator, and full-featured sets.

Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes (such as an image file’s contents).

sequence—a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are sequences of one-character strings; other, more general sequence types include lists and tuples, ranges

slicing, which is a way to extract an entire section (slice) in a single step. In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence being sliced.

as sequences, strings also support concatenation with the plus sign (joining two strings into a new string) and repetition (making a new string by repeating another). the meaning of an operation depends on the objects being operated on.

Every string operation is defined to produce a new string as its result, because strings are immutable in Python—they cannot be changed in place after they are created.

numbers, strings, and tuples are immutable; lists, dictionaries, and sets are not—they can be changed in place freely, as can most new objects you’ll code with classes.

bytearray is a distinct hybrid of immutable bytes strings (whose b'...' syntax is required in 3.X and optional in 2.X) and mutable lists (coded and displayed in [])

>>> B = bytearray(b'spam') # A bytes/list hybrid (ahead)

>>> B.extend(b'eggs') # 'b' needed in 3.X, not 2.X

>>> B # B[i] = ord(x) works here toobytearray(b'spameggs')

>>> B.decode() # Translate to normal string'spameggs'

Sequence operations are common for sequence types. Python runs from left to right during multi operation on an object.

Formating:

>>> **'%s, eggs, and %s' % ('spam', 'SPAM!')** # Formatting expression (all)

'spam, eggs, and SPAM!'

>>> **'{0}, eggs, and {1}'.format('spam', 'SPAM!')** # Formatting method (2.6+, 3.0+)

'spam, eggs, and SPAM!'

>>> **'{}, eggs, and {}'.format('spam', 'SPAM!')** # Numbers optional (2.7+, 3.1+)

'spam, eggs, and SPAM!'

As a rule of thumb, Python’s toolset is layered: generic operations that span multiple types show up as built-in functions or expressions (e.g., len(X), X[0]), but type-specific operations are method calls (e.g., aString.upper()).

call the built-in dir function. This function lists variables assigned in the caller’s scope when called with no argument; more usefully, it returns a list of all the attributes available for any object passed to it. Because methods are function attributes, they will show up in this list.

help is one of a handful of interfaces to a system of code that ships with Python known as PyDoc—a tool for extracting documentation from objects.

Python also supports a raw string literal that turns off the backslash escape mechanism. Such literals start with the letter r and are useful for strings like directory paths on Windows (e.g., r'C:\text\new').

The Python list object is the most general sequence provided by the language. Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in place by assignment to offsets as well as a variety of list method calls.

the list append method expands the list’s size and inserts an item at the end; the pop method (or an equivalent del statement) then removes an item at a given offset, causing the list to shrink. Other list methods insert an item at an arbitrary position (insert), remove a given item by value (remove), add multiple items at the end (extend), sort, reverse and so on.

## Nesting

One nice feature of Python’s core data types is that they support arbitrary nesting—we can nest them in any combination, and as deeply as we like.

list comprehension expression: are composed of an expression and a looping construct that share a variable name

range—a built-in that generates successive integers.

List comprehensions work on any type that is a sequence in Python, as well as some types that are not.

The dictionary index operation uses the same syntax as that used for sequences, but the item in the square brackets is a key, not a relative position.

The dictionary in membership expression allows us to query the existence of a key and branch on the result with a Python if statement.

The for loop, and its more general colleague the while loop, are the main ways we code repetitive tasks as statements in our scripts. Really, though, the for loop, like its relative the list comprehension introduced earlier, is a sequence operation. It works on any object that is a sequence and, like the list comprehension, even on some things that are not.

for loop looks like the list comprehension expression introduced earlier, it should: both are really general iteration tools. In fact, both will work on any iterable object that follows the iteration protocol.

an object is iterable if it is either a physically stored sequence in memory, or an object that generates one item at a time in the context of an iteration operation—a sort of “virtual” sequence. More formally, both types of objects are considered iterable because they support the iteration protocol—they respond to the iter call with an object that advances in response to next calls and raises an exception when finished producing values.

every Python tool that scans an object from left to right uses the iteration protocol.

The tuple object (pronounced “toople” or “tuhple,” depending on whom you ask) is roughly like a list that cannot be changed—tuples are sequences, like lists, but they are immutable, like strings. Functionally, they’re used to represent fixed collections of items: the components of a specific calendar date, for instance. Syntactically, they are normally coded in parentheses instead of square brackets, and they support arbitrary types, arbitrary nesting, and the usual sequence operations.

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences. Like lists and dictionaries, tuples support mixed types and nesting, but they don’t grow and shrink because they are immutable.

best way to read a file today is to not read it at all—files provide an iterator that automatically reads line by line in for loops and other contexts:>>> for line in open('data.txt'): print(line).

binary files are useful for processing media, accessing data created by C programs, and so on. To illustrate, Python’s struct module can both create and unpack packed binary data—raw bytes that record values that are not Python objects—to be written to a file in binary mode.

Sets are unordered collections of unique and immutable objects.

decimal numbers, which are fixed-precision floating-point numbers, and fraction numbers, which are rational numbers with both a numerator and a denominator.

The type object, returned by the type built-in function, is an object that gives the type of another object.

common sequence operations, such as indexing, concatenation, and slicing.

The term “mapping” denotes an object that maps keys to associated values. Python’s dictionary is the only mapping type in the core type set.

Numeric Types:

In Python, data takes the form of objects—either built-in objects that Python provides, or objects we create using Python tools and other languages such as C.

A complete inventory of Python’s numeric toolbox includes:

* Integer and floating-point objects
* Complex number objects
* Decimal: fixed-precision objects
* Fraction: rational number objects
* Sets: collections with numeric operations
* Booleans: true and false
* Built-in functions and modules: round, math, random, etc.
* Expressions; unlimited integer precision; bitwise operations; hex, octal, and binary formats
* Third-party extensions: vectors, libraries, visualization, plotting, etc.

Sets have both numeric and collection qualities.

Python provides integers, which are positive and negative whole numbers, and floating-point numbers, which are numbers with a fractional part (sometimes called “floats” for verbal economy).

If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression.

Python complex literals are written as realpart+imaginarypart, where the imaginarypart is terminated with a j or J. The realpart is technically optional, so the imaginarypart may appear on its own. Complex numbers may also be created with the complex(*real*, *imag*) built-in call.

Python provides a set of tools for processing number objects:Expression operators+, -, \*, /, >>, \*\*, &, etc.Built-in mathematical functionspow, abs, round, int, hex, bin, etc.Utility modulesrandom, math, etc.

% computes a division remainder. 10%3 = 1

the is operator tests object identity (i.e., address in memory, a strict form of equality), and lambda creates unnamed functions.

| Python expression operators and precedence | |
| --- | --- |
| **Operators** | **Description** |
| yield x | Generator function send protocol |
| lambda args: expression | Anonymous function generation |
| x if y else z | Ternary selection (x is evaluated only if y is true) |
| x or y | Logical OR (y is evaluated only if x is false) |
| x and y | Logical AND (y is evaluated only if x is true) |
| not x | Logical negation |
| x in y, x not in y  x is y, x is not y  x < y, x <= y, x > y, x >= y  x == y, x != y | Membership (iterables, sets)  Object identity tests  Magnitude comparison, set subset and superset  Value equality operators |
| x | y | Bitwise OR, set union |
| x ^ y | Bitwise XOR, set symmetric difference |
| x & y | Bitwise AND, set intersection |
| x << y, x >> y | Shift x left or right by y bits |
| x + y  x – y | Addition, concatenation  Subtraction, set difference |
| x \* y  x % y  x / y, x // y | Multiplication, repetition  Remainder, format  Division: true and floor |
| −x, +x  ˜x | Negation, identity  Bitwise NOT (inversion) |
| x \*\* y | Power (exponentiation) |
| x[i]  x[i:j:k]  x(...)  x.attr | Indexing (sequence, mapping, others)  Slicing  Call (function, method, class, other callable)  Attribute reference |
| (...)  [...]  {...} | Tuple, expression, generator expression  List, list comprehension  Dictionary, set, set and dictionary comprehensions |

in mixed-type numeric expressions, Python first converts operands up to the type of the most complicated operand, and then performs the math on same-type operands.

all Python operators may be overloaded (i.e., implemented) by Python classes and C extension types to work on objects you create. the + operator performs addition when applied to numbers but performs concatenation when applied to sequence objects such as strings and lists. In fact, + can mean anything at all when applied to objects you define with classes.

polymorphism—a term indicating that the meaning of an operation depends on the type of the objects being operated on.

Python also allows us to chain multiple comparisons together to perform range tests. Chained comparisons are a sort of shorthand for larger Boolean expressions.

C:\code> **C:\Python27\python**

>>> **from \_\_future\_\_ import division**

Complex numbers are typically used in engineering and science applications. Complex numbers also allow us to extract their parts as attributes, support all the usual mathematical expressions, and may be processed with tools in the standard cmath module (the complex version of the standard math module).

The eval function, treats strings as though they were Python code. Therefore, it has a similar effect, but usually runs more slowly—it actually compiles and runs the string as a piece of a program, and it assumes the string being run comes from a trusted source—a clever user might be able to submit a string that deletes files on your machine, so be careful with this call.

Notice that standard library modules such as math must be imported, but built-in functions such as abs and round are always available without imports. In other words, modules are external components, but built-in functions live in an implied namespace that Python automatically searches to find names used in your program.

the set—an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory.

because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves.

Sets are containers of iterable. Note that {} is still a dictionary in all Pythons. Empty sets must be created with the set built-in.

sets can only contain immutable (a.k.a. “hashable”) object types. Hence, lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values.

Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the frozenset built-in call works just like set but creates an immutable set that cannot change and thus can be embedded in other sets.

Set comprehensions run a loop and collect the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression. The result is a new set you create by running the code, with all the normal set behavior.

two sets are equal if and only if every element of each set is contained in the other—that is, each is a subset of the other, regardless of order.

Python today has an explicit Boolean data type called bool, with the values True and False available as preassigned built-in names.

Internally, the names True and False are instances of bool, which is in turn just a subclass (in the object-oriented sense) of the built-in integer type int.

an optional extension for Python called NumPy (Numeric Python) provides advanced numeric programming tools, such as a matrix data type, vector processing, and sophisticated computation libraries.

Variable types

A variable never has any type information or constraints associated with it. The notion of type lives with objects, not names. Variables are generic in nature; they always simply refer to a particular object at a particular point in time.

Variables always link to objects and never to other variables, but larger objects may link to other objects (for instance, a list object has links to the objects it contains).

These links from variables to objects are called references in Python—that is, a reference is a kind of association, implemented as a pointer in memory.[1](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch06.html#CHP-6-FN-1) Whenever the variables are later used (i.e., referenced), Python automatically follows the variable-to-object links.

Variables are entries in a system table, with spaces for links to objects.Objects are pieces of allocated memory, with enough space to represent the values for which they stand.References are automatically followed pointers from variables to objects.

Each object also has two standard header fields: a type designator used to mark the type of the object, and a reference counter used to determine when it’s OK to reclaim the object.

multiple names referencing the same object—is usually called a shared reference (and sometimes just a shared object).

in Python variables are always pointers to objects, not labels of changeable memory areas.

there are objects and operations that perform in-place object changes—Python’s mutable types, including lists, dictionaries, and sets.

Python caches and reuses small integers and small strings.

the == operator, tests whether the two referenced objects have the same values; this is the method almost always used for equality checks in Python. The second method, the is operator, instead tests for object identity—it returns True only if both names point to the exact same object, so it is a much stronger form of equality testing and is rarely applied in most programs.

the getrefcount function in the standard sys module returns the object’s reference count.

Python string—an ordered collection of characters used to store and represent text- and bytes-based information.

Unicode strings and files—tools for dealing with non-ASCII text. Unicode is a key tool for some programmers, especially those who work in the Internet domain. It can pop up, for example, in web pages, email content and headers, FTP transfers, GUI APIs, directory tools, and HTML, XML and JSON text.

In Python 3.X there are three string types: str is used for Unicode text (including ASCII), bytes is used for binary data (including encoded text), and bytearray is a mutable variant of bytes. Files work in two modes: text, which represents content as str and implements Unicode encodings, and binary, which deals in raw bytes and does no data translation.

the primary distinction of Unicode often lies in the translation (a.k.a. encoding) step required to move it to and from files.

Python strings are categorized as immutable sequences, meaning that the characters they contain have a left-to-right positional order and that they cannot be changed in place. In fact, strings are the first representative of the larger class of objects called sequences.

| Common string literals and operations | |
| --- | --- |
| **Operation** | **Interpretation** |
| S = '' | Empty string |
| S = "spam's" | Double quotes, same as single |
| S = 's\np\ta\x00m' | Escape sequences |
| S = """...*multiline*...""" | Triple-quoted block strings |
| S = r'\temp\spam' | Raw strings (no escapes) |
| B = b'sp\xc4m' | Byte strings in 2.6, 2.7, and 3.X ([Chapter 4](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch04.html#introducing_python_object_types), [Chapter 37](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch37.html#unicode_and_byte_strings)) |
| U = u'sp\u00c4m' | Unicode strings in 2.X and 3.3+ ([Chapter 4](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch04.html#introducing_python_object_types), [Chapter 37](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch37.html#unicode_and_byte_strings)) |
| S1 + S2  S \* 3 | Concatenate, repeat |
| S[i]  S[i:j]  len(S) | Index, slice, length |
| "a %s parrot" % kind | String formatting expression |
| "a {0} parrot".format(kind) | String formatting method in 2.6, 2.7, and 3.X |
| S.find('pa')  S.rstrip()  S.replace('pa', 'xx')  S.split(',')  S.isdigit()  S.lower()  S.endswith('spam')  'spam'.join(strlist)  S.encode('latin-1')  B.decode('utf8') | String methods (see ahead for all 43): search,  remove whitespace,  replacement,  split on delimiter,  content test,  case conversion,  end test,  delimiter join,  Unicode encoding,  Unicode decoding, etc. (see [Table 7-3](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch07.html#string_method_calls_in_python_3.3)) |
| for x in S: print(x)  'spam' in S  [c \* 2 for c in S]  map(ord, S) | Iteration, membership |
| re.match('sp(.\*)am', line) | Pattern matching: library module |

backslashes are used to introduce special character codings known as escape sequences.

no character terminates a string in Python.

If you need to end a raw string with a single backslash, you can use two and slice off the second (r'1\nb\tc\\'[:-1]), tack one on manually (r'1\nb\tc' + '\\'), or skip the raw string syntax and just double up the backslashes in a normal string ('1\\nb\\tc\\'). All three of these forms create the same eight-character string containing three backslashes.

Extended slicing (S[i:j:k]) accepts a step (or stride) k, which defaults to +1:Allows for skipping items and reversing order.

One of Python’s design mottos is that it refuses the temptation to guess. As a prime example, you cannot add a number and a string together in Python, even if the string looks like a number (i.e., is all digits):# Python 3.X>>> "42" + 1TypeError: Can't convert 'int' object to str implicitly.

>>> ord('s') 115

>>> chr(115) 's

In Python, expressions and built-in functions may work across a range of types, but methods are generally specific to object types.

| String formatting type codes | |
| --- | --- |
| **Code** | **Meaning** |
| s | String (or any object’s str(X) string) |
| r | Same as s, but uses repr, not str |
| c | Character (int or str) |
| d | Decimal (base-10 integer) |
| i | Integer |
| u | Same as d (obsolete: no longer unsigned) |
| o | Octal integer (base 8) |
| x | Hex integer (base 16) |
| X | Same as x, but with uppercase letters |
| e | Floating point with exponent, lowercase |
| E | Same as e, but uses uppercase letters |
| f | Floating-point decimal |
| F | Same as f, but uses uppercase letters |
| g | Floating-point e or f |
| G | Floating-point E or F |
| % | Literal % (coded as %%) |

## Method Call Syntax

methods are simply functions that are associated with and act upon particular objects. Technically, they are attributes attached to objects that happen to reference callable functions which always have an implied subject. In finer-grained detail, functions are packages of code, and method calls combine two operations at once—an attribute fetch and a call:

Attribute fetches

An expression of the form *object*.*attribute* means “fetch the value of *attribute* in *object*.”

Call expressions

An expression of the form *function*(*arguments*) means “invoke the code of *function*, passing zero or more comma-separated *argument* objects to it, and return *function*’s result value.”

String method calls in Python 3.3S.capitalize()S.ljust(width [, fill])S.casefold()S.lower()S.center(width [, fill])S.lstrip([chars])S.count(sub [, start [, end]])S.maketrans(x[, y[, z]])S.encode([encoding [,errors]])S.partition(sep)S.endswith(suffix [, start [, end]])S.replace(old, new [, count])S.expandtabs([tabsize])S.rfind(sub [,start [,end]])S.find(sub [, start [, end]])S.rindex(sub [, start [, end]])S.format(fmtstr, \*args, \*\*kwargs)S.rjust(width [, fill])S.index(sub [, start [, end]])S.rpartition(sep)S.isalnum()S.rsplit([sep[, maxsplit]])S.isalpha()S.rstrip([chars])S.isdecimal()S.split([sep [,maxsplit]])S.isdigit()S.splitlines([keepends])S.isidentifier()S.startswith(prefix [, start [, end]])S.islower()S.strip([chars])S.isnumeric()S.swapcase()S.isprintable()S.title()S.isspace()S.translate(map)S.istitle()S.upper()S.isupper()S.zfill(width)S.join(iterable)

The bytearray supports in-place text changes in 2.6, 3.0, and later, but only for simple 8-bit types.

String formatting expressions: '...%s...' % (values)

String formatting method calls: '...{}...'.format(values)

%[(*keyname*)][*flags*][*width*][.*precision*]*typecode*

{*fieldname component* !*conversionflag* :*formatspec*}

In this substitution target syntax:

* *fieldname* is an optional number or keyword identifying an argument, which may be omitted to use relative argument numbering in 2.7, 3.1, and later.
* *component* is a string of zero or more “.*name*” or “[*index*]” references used to fetch attributes and indexed values of the argument, which may be omitted to use the whole argument value.
* *conversionflag* starts with a ! if present, which is followed by r, s, or a to call repr, str, or ascii built-in functions on the value, respectively.
* *formatspec* starts with a : if present, which is followed by text that specifies how the value should be presented, including details such as field width, alignment, padding, decimal precision, and so on, and ends with an optional data type code.

The *formatspec* component after the colon character has a rich format all its own, and is formally described as follows (brackets denote optional components and are not coded literally):

[[*fill*]*align*][*sign*][#][0][*width*][,][.*precision*][*typecode*]

In this, *fill* can be any fill character other than { or }; *align* may be <, >, =, or ^, for left alignment, right alignment, padding after a sign character, or centered alignment, respectively; sign may be +, −, or space.

Numbers (integer, floating-point, decimal, fraction, others)

Support addition, multiplication, etc.

Sequences (strings, lists, tuples)

Support indexing, slicing, concatenation, etc.

Mappings (dictionaries)

Support indexing by key, etc.

Immutables (numbers, strings, tuples, frozensets)

Mutables (lists, dictionaries, sets, bytearray)

# Chapter 8. Lists and Dictionaries

Lists:

Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more—they are mutable objects.

Ordered collections of arbitrary objects

Accessed by offset

Variable-length, heterogeneous, and arbitrarily nestable

Of the category “mutable sequence”

Arrays of object references

| Common list literals and operations | |
| --- | --- |
| **Operation** | **Interpretation** |
| L = [] | An empty list |
| L = [123, 'abc', 1.23, {}] | Four items: indexes 0..3 |
| L = ['Bob', 40.0, ['dev', 'mgr']] | Nested sublists |
| L = list('spam')  L = list(range(-4, 4)) | List of an iterable’s items, list of successive integers |
| L[i]  L[i][j]  L[i:j]  len(L) | Index, index of index, slice, length |
| L1 + L2  L \* 3 | Concatenate, repeat |
| for x in L: print(x)  3 in L | Iteration, membership |
| L.append(4)  L.extend([5,6,7])  L.insert(i, X) | Methods: growing |
| L.index(X)  L.count(X) | Methods: searching |
| L.sort()  L.reverse()  L.copy()  L.clear() | Methods: sorting, reversing,  copying (3.3+), clearing (3.3+) |
| L.pop(i)  L.remove(X)  del L[i]  del L[i:j]  L[i:j] = [] | Methods, statements: shrinking |
| L[i] = 3  L[i:j] = [4,5,6] | Index assignment, slice assignment |
| L = [x\*\*2 for x in range(5)]  list(map(ord, 'spam')) | List comprehensions and maps |

Although the + operator works the same for lists and strings, it’s important to know that it expects the same sort of sequence on both sides—otherwise, you get a type error when the code runs.

>>> l = ['a','b','c']

>>> l1 = = 'hello'

>>> 'the string l1 %s and %s l are joned as %s' % (l1, l, str(l)+l1)

"the string l1 hello and ['a', 'b', 'c'] l are joned as ['a', 'b', 'c']hello"

for loops step through items in any sequence from left to right, executing one or more statements for each item; range produces successive integers.

list comprehensions are a way to build a new list by applying an expression to each item in a sequence (really, in any iterable), and are close relatives to for loops

Methods in brief, they are functions (really, object attributes that reference functions) that are associated with and act upon particular objects. Methods provide type-specific tools.

delete a section of a list by assigning an empty list to a slice (L[i:j]=[])

Dictionaries:

think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by key, instead of by positional offset. While lists can serve roles similar to arrays in other languages, dictionaries take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

Dictionaries are: Accessed by key, not offset position, Unordered collections of arbitrary objects, Variable-length, heterogeneous, and arbitrarily nestable, Of the category “mutable mapping”, Tables of object references (hash tables).

| Common dictionary literals and operations | |
| --- | --- |
| **Operation** | **Interpretation** |
| D = {} | Empty dictionary |
| D = {'name': 'Bob', 'age': 40} | Two-item dictionary |
| E = {'cto': {'name': 'Bob', 'age': 40}} | Nesting |
| D = dict(name='Bob', age=40)  D = dict([('name', 'Bob'), ('age', 40)])  D = dict(zip(keyslist, valueslist))  D = dict.fromkeys(['name', 'age']) | Alternative construction techniques:  keywords, key/value pairs, zipped key/value pairs, key lists |
| D['name']  E['cto']['age'] | Indexing by key |
| 'age' in D | Membership: key present test |
| D.keys()  D.values()  D.items()  D.copy()  D.clear()  D.update(D2)  D.get(key, default?)  D.pop(key, default?)  D.setdefault(key, default?)  D.popitem() | Methods: all keys,  all values,  all key+value tuples,  copy (top-level),  clear (remove all items),  merge by keys,  fetch by key, if absent default (or None),  remove by key, if absent default (or error)  fetch by key, if absent set default (or None),  remove/return any (key, value) pair; etc. |
| len(D) | Length: number of stored entries |
| D[key] = 42 | Adding keys, changing key values |
| del D[key] | Deleting entries by key |
| list(D.keys())  D1.keys() & D2.keys() | Dictionary views (Python 3.X) |
| D.viewkeys(), D.viewvalues() | Dictionary views (Python 2.7) |
| D = {x: x\*2 for x in range(10)} | Dictionary comprehensions (Python 3.X, 2.7) |

Dictionaries, like lists, are mutable, so you can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The del statement works here, too; it deletes the entry associated with the key specified as an index.

the get method returns a default value—None, or a passed-in default—if the key doesn’t exist. It’s an easy way to fill in a default for a key that isn’t present, and avoid a missing-key error when your program can’t anticipate contents ahead of time.

The update method provides something similar to concatenation for dictionaries, though it has nothing to do with left-to-right ordering (again, there is no such thing in dictionaries). It *merges* the keys and values of one dictionary into another, blindly overwriting values of the same key if there’s a clash:

There are extension types in Python’s standard library that maintain insertion order among their keys—see OrderedDict in the collections module—but they are hybrids that incur extra space and speed overheads to achieve their extra utility, and are not true dictionaries. In short, keys are kept redundantly in a linked list to support sequence operations.

a namedtuple that allows tuple items to be accessed by both attribute name and sequence position.

## Dictionary Usage Notes

Dictionaries are fairly straightforward tools once you get the hang of them, but here are a few additional pointers and reminders you should be aware of when using them:

* **Sequence operations don’t work**. Dictionaries are mappings, not sequences; because there’s no notion of ordering among their items, things like concatenation (an ordered joining) and slicing (extracting a contiguous section) simply don’t apply. In fact, Python raises an error when your code runs if you try to do such things.
* **Assigning to new indexes adds entries**. Keys can be created when you write a dictionary literal (embedded in the code of the literal itself), or when you assign values to new keys of an existing dictionary object individually. The end result is the same.
* **Keys need not always be strings**. Our examples so far have used strings as keys, but any other immutable objects work just as well. For instance, you can use integers as keys, which makes the dictionary look much like a list (when indexing, at least). Tuples may be used as dictionary keys too, allowing compound key values—such as dates and IP addresses—to have associated values. User-defined class instance objects can also be used as keys, as long as they have the proper protocol methods; roughly, they need to tell Python that their values are “hashable” and thus won’t change, as otherwise they would be useless as fixed keys. Mutable objects such as lists, sets, and other dictionaries don’t work as keys, but are allowed as values.

>>> d = dict([('name','Bob'),('age',90)])

>>> d

{'name': 'Bob', 'age': 90}

For some types of data, the list’s access-by-position makes sense—a list of employees in a company, the files in a directory, or numeric matrixes, for example.

In practice, dictionaries tend to be best for data with labeled components, as well as structures that can benefit from quick, direct lookups by name, instead of slower linear searches.

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the fromkeys method.

>>> **D = dict.fromkeys(['a', 'b', 'c'], 0)** # Initialize dict from keys

>>> **D**

{'b': 0, 'c': 0, 'a': 0}

>>> **D = dict.fromkeys('spam')** # Other iterables, default value

>>> **D**

{'s': None, 'p': None, 'a': None, 'm': None}

In 3.X the dictionary keys, values, and items methods all return view objects.

View objects are iterables, which simply means objects that generate result items one at a time, instead of producing the result list all at once in memory. Besides being iterable, dictionary views also retain the original order of dictionary components, reflect future changes to the dictionary, and may support set operations. On the other hand, because they are not lists, they do not directly support operations like indexing or the list sort method, and do not display their items as a normal list when printed.

JSON—a language-neutral data format used for databases and data transfer.

 Dictionaries are generally better when the data is labeled (a record with field names, for example); lists are best suited to collections of unlabeled items (such as all the files in a directory). Dictionary lookup is also usually quicker than searching a list, though this might vary per program.

# Chapter 9. Tuples, Files, and Everything Else

tuple, a collection of other objects that cannot be changed, and the file, an interface to external files on your computer.

Tuples construct simple groups of objects. They work exactly like lists, except that tuples can’t be changed in place (they’re immutable) and are usually written as a series of items in parentheses, not square brackets. Although they don’t support as many methods, tuples share most of their properties with lists.

Ordered collections of arbitrary objects

Like strings and lists, tuples are positionally ordered collections of objects (i.e., they maintain a left-to-right order among their contents); like lists, they can embed any kind of object.

Accessed by offset

Like strings and lists, items in a tuple are accessed by offset (not by key); they support all the offset-based access operations, such as indexing and slicing.

Of the category “immutable sequence”

Like strings and lists, tuples are sequences; they support many of the same operations. However, like strings, tuples are immutable; they don’t support any of the in-place change operations applied to lists.

Fixed-length, heterogeneous, and arbitrarily nestable

Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and so support arbitrary nesting.

Arrays of object references

Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (references), and indexing a tuple is relatively quick.

| Table 9-1. Common tuple literals and operations | |
| --- | --- |
| **Operation** | **Interpretation** |
| () | An empty tuple |
| T = (0,) | A one-item tuple (not an expression) |
| T = (0, 'Ni', 1.2, 3) | A four-item tuple |
| T = 0, 'Ni', 1.2, 3 | Another four-item tuple (same as prior line) |
| T = ('Bob', ('dev', 'mgr')) | Nested tuples |
| T = tuple('spam') | Tuple of items in an iterable |
| T[i]  T[i][j]  T[i:j]  len(T) | Index, index of index, slice, length |
| T1 + T2  T \* 3 | Concatenate, repeat |
| for x in T: print(x)  'spam' in T  [x \*\* 2 for x in T] | Iteration, membership |
| T.index('Ni')  T.count('Ni') | Methods in 2.6, 2.7, and 3.X: search, count |
| namedtuple('Emp', ['name', 'jobs']) |  |

+, \*, and slicing operations return new tuples when applied to tuples, and that tuples don’t provide the same methods you saw for strings, lists, and dictionaries.

List comprehensions are really sequence operations—they always build new lists, but they may be used to iterate over any sequence objects, including tuples, strings, and other lists.

index and count work as they do for lists, but they are defined for tuple objects

the rule about tuple immutability applies only to the top level of the tuple itself, not to its contents. A list inside a tuple, for instance, can be changed as usual

integrity—you can be sure a tuple won’t be changed through another reference elsewhere in a program, but there’s no such guarantee for lists. Tuples and other immutables, therefore, serve a similar role to “constant” declarations in other languages, though the notion of constantness is associated with objects in Python, not variables.

Tuples can also be used in places that lists cannot—for example, as dictionary keys.

As a rule of thumb, lists are the tool of choice for ordered collections that might need to change; tuples can handle the other cases of fixed associations.

we can convert parts of the dictionary to a tuple if needed:

>>> **tuple(bob.values())**

>>> **list(bob.items())**

the namedtuple utility, available in the standard library’s collections module implements an extension type that adds logic to tuples that allows components to be accessed by both position and attribute name, and can be converted to dictionary-like form for access by key if desired. Attribute names come from classes and are not exactly dictionary keys.

Converting to a dictionary supports key-based behavior when needed:

>>> **O = bob.\_asdict()** // bob is a namedtuple

In short, named tuples build new classes that extend the tuple type, inserting a property accessor method for each named field that maps the name to its position.

Files: You may already be familiar with the notion of files, which are named storage compartments on your computer that are managed by your operating system.

the built-in open function creates a Python file object, which serves as a link to a file residing on your machine. After calling open, you can transfer strings of data to and from the associated external file by calling the returned file object’s methods.

| Common file operations | |
| --- | --- |
| **Operation** | **Interpretation** |
| output = open(r'C:\spam', 'w') | Create output file ('w' means write) |
| input = open('data', 'r') | Create input file ('r' means read) |
| input = open('data') | Same as prior line ('r' is the default) |
| aString = input.read() | Read entire file into a single string |
| aString = input.read(N) | Read up to next N characters (or bytes) into a string |
| aString = input.readline() | Read next line (including \n newline) into a string |
| aList = input.readlines() | Read entire file into list of line strings (with \n) |
| output.write(aString) | Write a string of characters (or bytes) into file |
| output.writelines(aList) | Write all line strings in a list into file |
| output.close() | Manual close (done for you when file is collected) |
| output.flush() | Flush output buffer to disk without closing |
| anyFile.seek(N) | Change file position to offset N for next operation |
| for line in open('data'): *use line* | File iterators read line by line |
| open('f.txt', encoding='latin-1') | Python 3.X Unicode text files (str strings) |
| open('f.bin', 'rb') | Python 3.X bytes files (bytes strings) |
| codecs.open('f.txt', encoding='utf8') | Python 2.X Unicode text files (unicode strings) |
| open('f.bin', 'rb') | Python 2.X bytes files (str strings) |

## Opening Files

To open a file, a program calls the built-in open function, with the external filename first, followed by a processing mode. The call returns a file object, which in turn has methods for data transfer:

afile = open(*filename*, *mode*) // afile is a file object.

File iterators are best for reading lines: files also have an iterator that automatically reads one line at a time in a for loop, list comprehension, or other iteration context.

Content is Strings, not objects; data read from a file always comes back to your script as a string.

Python does not add any formatting and does not convert objects to strings automatically when you write data to a file—you must send an already formatted string. Because of this, the tools we have already met to convert objects to and from strings (e.g., int, float, str, and the string formatting expression and method) come in handy when dealing with files.

Files are buffered and seekable: By default, output files are always buffered, which means that text you write may not be transferred from memory to disk immediately—closing a file, or running its flush method, forces the buffered data to disk. their seek method allows your scripts to jump around to read and write at specific locations.

Calling the file close method terminates your connection to the external file, releases its system resources, and flushes its buffered output to disk if any is still in memory. When file objects are reclaimed, Python also automatically closes the files if they are still open.

write methods don’t add the end-of-line character for us, so we must include it to properly terminate our lines (otherwise the next write will simply extend the current line in the file).

if you want to scan a text file line by line, *file iterators* are often your best option:

>>> **for line in open('myfile.txt'):** # Use file iterators, not reads

... **print(line, end='')**

Text files represent content as normal str strings, perform Unicode encoding and decoding automatically, and perform end-of-line translation by default.Binary files represent content as a special bytes string type and allow programs to access file content unaltered.

Using eval to convert from strings to objects

The pickle module is a more advanced tool that allows us to store almost any Python object in a file directly, with no to- or from-string conversion requirement on our part. It’s like a super-general data formatting and parsing utility.

To store a dictionary in a file, for instance, we pickle it directly:

>>> **D = {'a': 1, 'b': 2}**

>>> **F = open('datafile.pkl', 'wb')**

>>> **import pickle**

>>> **pickle.dump(D, F)** # Pickle any object to file

>>> **F.close()**

Then, to get the dictionary back later, we simply use pickle again to re-create it:

>>> **F = open('datafile.pkl', 'rb')**

>>> **E = pickle.load(F)** # Load any object from file

>>> **E**

{'a': 1, 'b': 2}

The pickle module performs what is known as object serialization—converting objects to and from strings of bytes—but requires very little work on our part. In fact, pickle internally translates our dictionary to a string form.

shelve is a tool that uses pickle to store Python objects in an access-by-key filesystem.

JSON is a newer and emerging data interchange format, which is both programming-language-neutral and supported by a variety of systems. MongoDB, for instance, stores data in a JSON document database (using a binary JSON format).

Note that strings are all Unicode in JSON to support text drawn from international character sets

the standard library’s csv module. It parses and creates CSV (comma-separated value) data in files and strings.

the struct module knows how to both compose and parse packed binary data.

>>> **F = open('data.bin', 'wb')** # Open binary output file

>>> **import struct**

>>> **data = struct.pack('>i4sh', 7, b'spam', 8)** # Make packed binary data

>>> **data**

b'\x00\x00\x00\x07spam\x00\x08'

>>> **F.write(data)** # Write byte string

>>> **F.close()**

The with context manager scheme ensures release of system resources in all Pythons

there are additional file-like tools in the Python toolset. Among these:

Standard streams

Preopened file objects in the sys module, such as sys.stdout (see [“Print Operations”](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch11.html#print_operations) in [Chapter 11](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch11.html#assignments_comma_expressions_comma_and) for details)

Descriptor files in the os module

Integer file handles that support lower-level tools such as file locking (see also the “x” mode in Python 3.3’s open for exclusive creation)

Sockets, pipes, and FIFOs

File-like objects used to synchronize processes or communicate over networks

Access-by-key files known as “shelves”

Used to store unaltered and pickled Python objects directly, by key (used in [Chapter 28](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch28.html#a_more_realistic_example))

Shell command streams

Tools such as os.popen and subprocess.Popen that support spawning shell commands and reading and writing to their standard streams

The third-party open source domain offers even more file-like tools, including support for communicating with serial ports in the PySerial extension and interactive programs in the pexpect system.

| Table 9-3. Object classifications | | |
| --- | --- | --- |
| **Object type** | **Category** | **Mutable?** |
| Numbers (all) | Numeric | No |
| Strings (all) | Sequence | No |
| Lists | Sequence | Yes |
| Dictionaries | Mapping | Yes |
| Tuples | Sequence | No |
| Files | Extension | N/A |
| Sets | Set | Yes |
| Frozenset | Set | No |
| bytearray | Sequence | Yes |

Python compares types as follows:

* Numbers are compared by relative magnitude, after conversion to the common highest type if needed.
* Strings are compared lexicographically (by the character set code point values returned by ord), and character by character until the end or first mismatch ("abc" < "ac").
* Lists and tuples are compared by comparing each component from left to right, and recursively for nested structures, until the end or first mismatch ([2] > [1, 2]).
* Sets are equal if both contain the same items (formally, if each is a subset of the other), and set relative magnitude comparisons apply subset and superset tests.
* Dictionaries compare as equal if their sorted (*key*, *value*) lists are equal. Relative magnitude comparisons are not supported for dictionaries in Python 3.X, but they work in 2.X as though comparing sorted (*key*, *value*) lists.
* Nonnumeric mixed-type magnitude comparisons (e.g., 1 < 'spam') are errors in Python 3.X. They are allowed in Python 2.X, but use a fixed but arbitrary ordering rule based on type name string. By proxy, this also applies to sorts, which use comparisons internally: nonnumeric mixed-type collections cannot be sorted in 3.X.

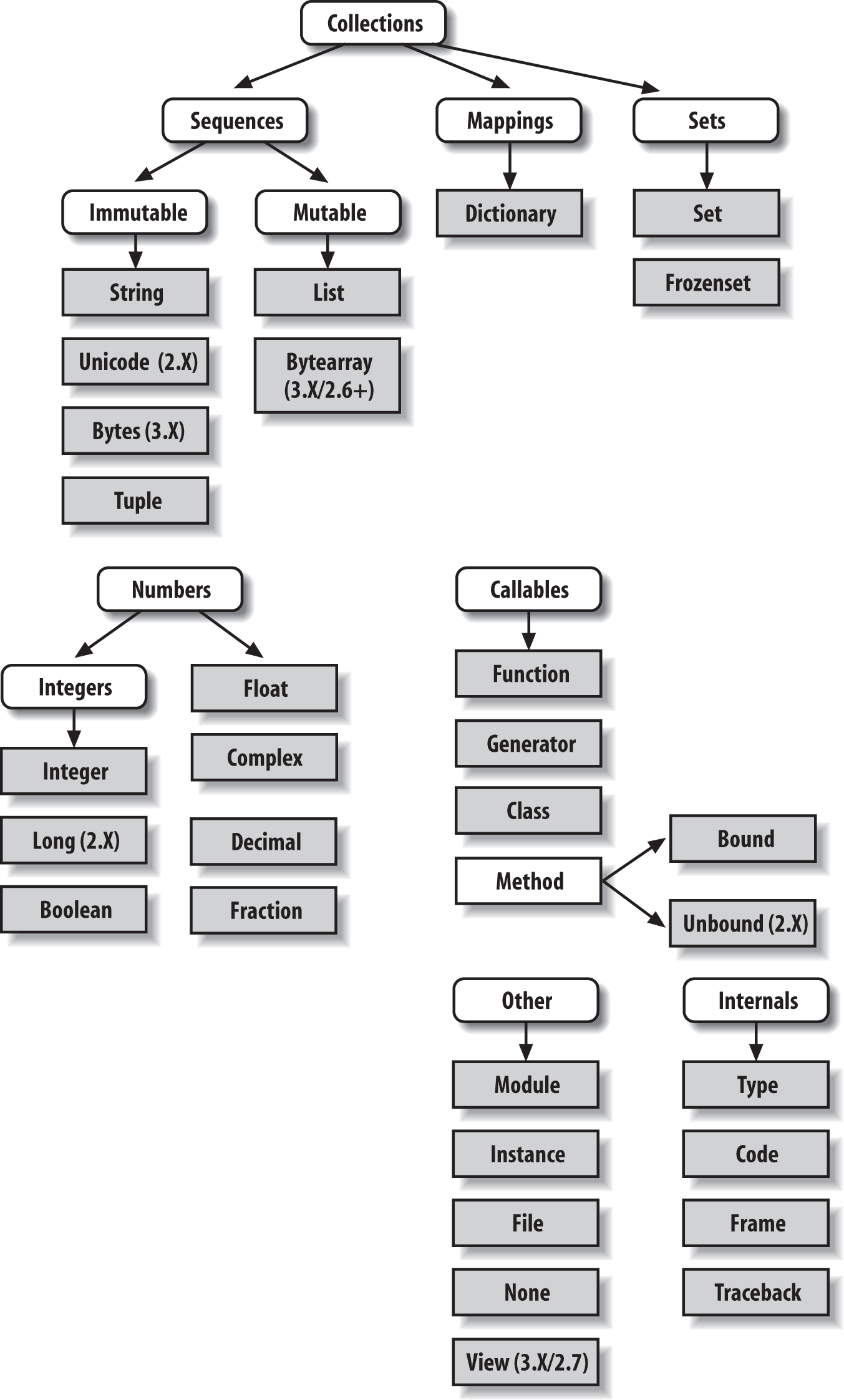
| Example object truth values | |
| --- | --- |
| **Object** | **Value** |
| "spam" | True |
| "" | False |
| [1, 2] | True |
| [] | False |
| {'a': 1} | True |
| {} | False |
| 1 | True |
| 0.0 | False |
| None | False |

None does not mean “undefined.” That is, None is something, not nothing (despite its name!)—it is a real object and a real piece of memory that is created and given a built-in name by Python itself. it is also the default return value of functions that don’t exit by running into a return statement with a result value.

Python also provides a bool built-in function that can be used to test the Boolean value of an object

The largest point to notice here is that everything in a Python system is an object type and may be processed by your Python programs.

even types themselves are an object type in Python: the type of an object is an object of type type.



to make a regular expression object, you import re and call re.compile()

## Assignment Creates References, Not Copies

Because this is such a central concept, I’ll mention it again: shared references to mutable objects in your program can matter.

if a collection object contains a reference to itself, it’s called a cyclic object. Python prints a [...] whenever it detects a cycle in the object, rather than getting stuck in an infinite loop.

>>> l = ['Grail']

>>> l.append(l)

>>> l

['Grail', [...]]

statement syntax—the way you code processing logic in your scripts.

a string is a collection of characters, but Python characters are one-character strings.

statements are the things you write to tell Python what your programs should do. Python is a procedural, statement-based language; by combining statements, you specify a procedure that Python performs to satisfy a program’s goals.

Python program structure:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. Expressions create and process objects.

statements are where objects spring into existence (e.g., in expressions within assignment statements), and some statements create entirely new kinds of objects (functions, classes, and so on).

| Python statements | | |
| --- | --- | --- |
| **Statement** | **Role** | **Example** |
| Assignment | Creating references | a, b = 'good', 'bad' |
| Calls and other expressions | Running functions | log.write("spam, ham") |
| print calls | Printing objects | print('The Killer', joke) |
| if/elif/else | Selecting actions | if "python" in text:  print(text) |
| for/else | Iteration | for x in mylist:  print(x) |
| while/else | General loops | while X > Y:  print('hello') |
| pass | Empty placeholder | while True:  pass |
| break | Loop exit | while True:  if exittest(): break |
| continue | Loop continue | while True:  if skiptest(): continue |
| def | Functions and methods | def f(a, b, c=1, \*d):  print(a+b+c+d[0]) |
| return | Functions results | def f(a, b, c=1, \*d):  return a+b+c+d[0] |
| yield | Generator functions | def gen(n):  for i in n: yield i\*2 |
| global | Namespaces | x = 'old'  def function():  global x, y; x = 'new' |
| nonlocal | Namespaces (3.X) | def outer():  x = 'old'  def function():  nonlocal x; x = 'new' |
| import | Module access | import sys |
| from | Attribute access | from sys import stdin |
| class | Building objects | class Subclass(Superclass):  staticData = []  def method(self): pass |
| try/except/ finally | Catching exceptions | try:  action()  except:  print('action error') |
| raise | Triggering exceptions | raise EndSearch(location) |
| assert | Debugging checks | assert X > Y, 'X too small' |
| with/as | Context managers (3.X, 2.6+) | with open('data') as myfile:  process(myfile) |
| del | Deleting references | del data[k]  del data[i:j]  del obj.attr  del variable |

compound statements—statements that have other statements nested inside them.

Indentation syntax:

It essentially means that you must line up your code vertically, in columns, according to its logical structure. The net effect is to make your code more consistent and readable.

As a rule of thumb, you probably shouldn’t mix tabs and spaces in the same block in Python, unless you do so consistently; use tabs or spaces in a given block, but not both

Although statements normally appear one per line, it is possible to squeeze more than one statement onto a single line in Python by separating them with semicolons:

a = 1; b = 2; print(a + b)

This is the only place in Python where semicolons are required: as statement separators.

The other special rule for statements is essentially the inverse: you can make a single statement span across multiple lines. To make this work, you simply have to enclose part of your statement in a bracketed pair—parentheses (()), square brackets ([]), or curly braces ({}). Any code enclosed in these constructs can cross multiple lines: your statement doesn’t end until Python reaches the line containing the closing part of the pair.

An older rule also allows for continuation lines when the prior line ends in a backslash:

X = A + B + \

C + D

As one special case here, the body of a compound statement can instead appear on the same line as the header in Python, after the colon:

if x > y: print(x)

In terms of statement nesting, because the words try, except, and else are all indented to the same level, they are all considered part of the same single try statement. Notice that the else part is associated with the try here, not the if.

>>> eval('print("hello")')

hello

>>> help(compile)

Help on built-in function compile in module builtins:

compile(source, filename, mode, flags=0, dont\_inherit=False, optimize=-1, \*, \_feature\_version=-1)

Compile source into a code object that can be executed by exec() or eval().

The source code may represent a Python module, statement or expression.

The filename will be used for run-time error messages.

The mode must be 'exec' to compile a module, 'single' to compile a

single (interactive) statement, or 'eval' to compile an expression.

The flags argument, if present, controls which future statements influence

the compilation of the code.

The dont\_inherit argument, if true, stops the compilation inheriting

the effects of any future statements in effect in the code calling

compile; if absent or false these statements do influence the compilation,

in addition to any features explicitly specified.

Assignment statements: you write the target of an assignment on the left of an equals sign, and the object to be assigned on the right. The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that computes an object.

Assignments create object references. Python assignments store references to objects in names or data structure components. They always create references to objects instead of copying the objects.

Once assigned, a name is replaced with the value it references whenever it appears in an expression.

module imports, function and class definitions, for loop variables, and function arguments are all implicit assignments.

Sequence-unpacking assignments also give rise to another common coding idiom in Python—assigning an integer series to a set of variables:

>>> **red, green, blue = range(3)**

extended sequence unpacking assignment is just a convenience. We can usually achieve the same effects with explicit indexing and slicing

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

print(a,b,c)

1 [2, 3] 7

4 [5, 6] 8

|  |  |  |  |
| --- | --- | --- | --- |
| Augmented assignment statements | | | |
| X += Y | X &= Y | X −= Y | X |= Y |
| X \*= Y | X ^= Y | X /= Y | X >>= Y |
| X %= Y | X <<= Y | X \*\*= Y | X //= Y |

Augmented assignment works on any type that supports the implied binary expression.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Python 3.X reserved words | | | | |
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass |  |
| break | except | in | raise |  |

Here is a list of the conventions Python follows:

* Names that begin with a single underscore (\_X) are not imported by a from module import \* statement (described in [Chapter 23](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch23.html#module_coding_basics)).
* Names that have two leading and trailing underscores (\_\_X\_\_) are system-defined names that have special meaning to the interpreter.
* Names that begin with two underscores and do not end with two more (\_\_X) are localized (“mangled”) to enclosing classes (see the discussion of pseudoprivate attributes in [Chapter 31](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch31.html#designing_with_classes)).
* The name that is just a single underscore (\_) retains the result of the last expression when you are working interactively.

although expressions can appear as statements in Python, statements cannot be used as expressions.

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

>>> **import sys** # Printing the hard way

>>> **sys.stdout.write('hello world\n')**

from \_\_future\_\_ import print\_function

This statement changes 2.X to support 3.X’s print functions exactly.

A **statement is** a complete line of code that performs some action, while an **expression is** any section of the code that evaluates to a value.

Here’s a general rule of thumb: If you can print it, or assign it to a variable, it’s an **expression**. If you can’t, it’s a **statement**.

Here are some examples of expressions:

1. 2 + 2
2. 3 \* 7
3. 1 + 2 + 3 \* (8 \*\* 9) - sqrt(4.0)
4. min(2, 22)
5. max(3, 94)
6. round(81.5)
7. "foo"
8. "bar"
9. "foo" + "bar"
10. None
11. True
12. False
13. 2
14. 3
15. 4.0

All of the above can be printed or assigned to a variable.

Here are some examples of statements:

1. if CONDITION:
2. elif CONDITION:
3. else:
4. for VARIABLE in SEQUENCE:
5. while CONDITION:
6. try:
7. except EXCEPTION as e:
8. class MYCLASS:
9. def MYFUNCTION():
10. return SOMETHING
11. raise SOMETHING
12. with SOMETHING:

compound statements—statements that embed other statements

The general form of an if statement looks like this:

if *test1*: # if test

*statements1* # Associated block

elif *test2*: # Optional elifs

*statements2*

else: # Optional else

*statements3*

dictionaries and lists can be built at runtime dynamically, they are sometimes more flexible than hardcoded if logic in your script:

>>> **choice = 'ham'**

>>> **print({'spam': 1.25,** # A dictionary-based 'switch'

... **'ham': 1.99,** # Use has\_key or get for default

... **'eggs': 0.99,**

... **'bacon': 1.10}[choice])**

1.99

dictionaries can also contain functions to represent more complex branch actions and implement general jump tables. Such functions appear as dictionary values, they may be coded as function names or inline lambdas, and they are called by adding parentheses to trigger their actions.

**def function(): ...**

**def default(): ...**

**branch = {'spam': lambda: ...,** # A table of callable function objects

**'ham': function,**

**'eggs': lambda: ...}**

**branch.get(choice, default)()**

Python’s path through a program is called the control flow, statements such as if that affect it are often called control-flow statements.

Compound statements = header + “:” + indented statements.

One rule of thumb: although you can use spaces or tabs to indent, it’s usually not a good idea to mix the two within a block—use one or the other.

**Statements may span multiple lines if you’re continuing an open syntactic pair.** Python lets you continue typing a statement on the next line if you’re coding something enclosed in a (), {}, or [] pair.

**Statements may span multiple lines if they end in a backslash.**

triple-quoted string blocks are designed to span multiple lines normally.

you can terminate a statement with a semicolon—this convention is sometimes used to squeeze more than one simple (noncompound) statement onto a single line.

* All objects have an inherent Boolean true or false value.
* Any nonzero number or nonempty object is true.
* Zero numbers, empty objects, and the special object None are considered false.
* Comparisons and equality tests are applied recursively to data structures.
* Comparisons and equality tests return True or False (custom versions of 1 and 0).
* Boolean and and or operators return a true or false operand object.
* Boolean operators stop evaluating (“short circuit”) as soon as a result is known.

Ternary: A = Y if X else Z

>>> ['f', 't'][bool('spam')]

't'

 An if statement with multiple elif clauses is often the most straightforward way to code a multiway branch, though not necessarily the most concise or flexible. Dictionary indexing can often achieve the same result, especially if the dictionary contains callable functions coded with def statements or lambda expressions.

# Chapter 13. while and for Loops

looping constructs—statements that repeat an action over and over.

the while statement, provides a way to code general loops. The second, the for statement, is designed for stepping through the items in a sequence or other iterable object and running a block of code for each.

built-ins commonly used with loops, such as range, zip, and map.

While: it repeatedly executes a block of (normally indented) statements as long as a test at the top keeps evaluating to a true value. It is called a “loop” because control keeps looping back to the start of the statement until the test becomes false.

while *test*: # Loop test

*statements* # Loop body

else: # Optional else

*statements* # Run if didn't exit loop with break

else part that is executed if control exits the loop without a break statement being encountered.

break

Jumps out of the closest enclosing loop (past the entire loop statement)

continue

Jumps to the top of the closest enclosing loop (to the loop’s header line)

pass

Does nothing at all: it’s an empty statement placeholder

Loop else block

Runs if and only if the loop is exited normally (i.e., without hitting a break)

while *test*:

*statements*

if *test*: break # Exit loop now, skip else if present

if *test*: continue # Go to test at top of loop now

else:

*statements* # Run if we didn't hit a 'break'

break and continue statements can appear anywhere inside the while (or for) loop’s body, but they are usually coded further nested in an if test to take action in response to some condition.

## pass

Simple things first: the pass statement is a no-operation placeholder that is used when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement.

pass is roughly to statements as None is to objects.

A pass is also sometime coded to mean “to be filled in later,” to stub out the bodies of functions temporarily:

def func1():

pass # Add real code here later

## break

The break statement causes an immediate exit from a loop. Because the code that follows it in the loop is not executed if the break is reached, you can also sometimes avoid nesting by including a break.

‘input’ returns user input as a string.

x = 0

while x:

print(x)

x += 1

if x == 10: break

else:

print("hai")

# for Loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The for statement works on strings, lists, tuples, and other built-in iterables, as well as new user-defined objects

The Python for loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through.

for *target* in *object*: # Assign object items to target

*statements* # Repeated loop body: use target

else: # Optional else part

*statements* # If we didn't hit a 'break'

When Python runs a for loop, it assigns the items in the iterable object to the target one by one and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

The break and continue statements introduced earlier also work the same in a for loop as they do in a while. The for loop’s complete format can be described this way:

for *target* in *object*: # Assign object items to target

*statements*

if *test*: break # Exit loop now, skip else

if *test*: continue # Go to top of loop now

else:

*statements* # If we didn't hit a 'break'

Any sequence works in a for, as it’s a generic tool.

### Tuple assignment in for loops

If you’re iterating through a sequence of tuples, the loop target itself can actually be a tuple of targets. This is just another case of the tuple-unpacking assignment we studied in at work. Remember, the for loop assigns items in the sequence object to the target, and assignment works the same everywhere:

>>> **T = [(1, 2), (3, 4), (5, 6)]**

>>> **for (a, b) in T:** # Tuple assignment at work

... **print(a, b)**

...

1 2

3 4

5 6

You typically read binary data in blocks. To read text files *line by line*, though, the for loop tends to be easiest to code and the quickest to run:

for line in open('test.txt').readlines():

print(line.rstrip())

for line in open('test.txt'): # Use iterators: best for text input

print(line.rstrip())

Both of these versions work in both Python 2.X and 3.X. The first uses the file readlines method to load a file all at once into a line-string list, and the last example here relies on file *iterators* to automatically read one line on each loop iteration.

* The built-in range function (available since Python 0.X) produces a series of successively higher integers, which can be used as indexes in a for.
* The built-in zip function (available since Python 2.0) returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a for.
* The built-in enumerate function (available since Python 2.3) generates both the values and indexes of items in an iterable, so we don’t need to count manually.
* The built-in map function (available since Python 1.0) can have a similar effect to zip in Python 2.X, though this role is removed in 3.X.

Range can be used anywhere you need a series of integers.

in 3.X, range is an iterable that generates items on demand, so we need to wrap it in a list call to display its results all at once.

With one argument, range generates a list of integers from zero up to but not including the argument’s value. If you pass in two arguments, the first is taken as the lower bound. An optional third argument can give a step; if it is used, Python adds the step to each successive integer in the result (the step defaults to +1). Ranges can also be nonpositive and nonascending, if you want them to be.

As a general rule, use for instead of while whenever possible, and don’t use range calls in for loops except as a last resort. This simpler solution is almost always better.

>>> **L = [1, 2, 3, 4, 5]**

>>> **for i in range(len(L)):** # Add one to each item in L

... **L[i] += 1** # Or L[i] = L[i] + 1

...

>>> **L**

[2, 3, 4, 5, 6]

the built-in zip function allows us to use for loops to visit multiple sequences in parallel—not overlapping in time, but during the same loop. In basic operation, zip takes one or more sequences as arguments and returns a series of tuples that pair up parallel items taken from those sequences.

zip truncates result tuples at the length of the shortest sequence when the argument lengths differ.

Normally, map takes a function and one or more sequence arguments and collects the results of calling the function with parallel items taken from the sequence(s).

like zip, map is a value generator in 3.X and so must be passed to list to collect all its results at once in 3.X only):>>> list(map(ord, 'spam'))[115, 112, 97, 109]

Enumerate: net effect is to give loops a counter “for free,” without sacrificing the simplicity of automatic iteration:

>>> **S = 'spam'**

>>> **for (offset, item) in enumerate(S):**

... **print(item, 'appears at offset', offset)**

...

s appears at offset 0

p appears at offset 1

a appears at offset 2

m appears at offset 3

A generator object supports iteration protocol. The enumerate function returns a generator object. it has a method called by the next built-in function, which returns an (*index*, *value*) tuple each time through the loop.

to run a shell command and read its standard output text, pass the command as a string to os.popen, and read text from the file-like object it returns.

>>> **import os**

>>> **F = os.popen('dir')** # Read line by line

>>> **F.readline()**

' Volume in drive C has no label.\n'

>>> **F = os.popen('dir')** # Read by sized blocks

>>> **F.read(50)**

' Volume in drive C has no label.\n Volume Serial Nu'

>>> **import os**

>>> **F = os.popen('dir')** # Read line by line

>>> **F.readline()**

' Volume in drive C has no label.\n'

>>> **F = os.popen('dir')** # Read by sized blocks

>>> **F.read(50)**

' Volume in drive C has no label.\n Volume Serial Nu'

Tools like os.popen and os.system (and the subprocess module not shown here) allow you to leverage every command-line program on your computer, but you can also write emulators with in-process code.

Chapter 14. Iterations and Comprehensions

iterable objects include both physical sequences and virtual sequences computed on demand.

the for loop turns out to be even more generic than this—it works on any iterable object. In fact, this is true of all iteration tools that scan objects from left to right in Python, including for loops, the list comprehensions we’ll study in this chapter, in membership tests, the map built-in function, and more.

**the term iterable to refer to an object that supports the iter call, and iterator to refer to an object returned by an iterable on iter that supports the next(*I*) call.**

Difference between readline() and \_\_next\_\_ on file object: The only noticeable difference is that \_\_next\_\_ raises a built-in StopIteration exception at end-of-file instead of returning an empty string as in readline():

all iteration tools normally work internally by calling \_\_next\_\_ on each iteration and catching the StopIteration exception to determine when to exit.

the best way to read a text file line by line today is to not read it at all—instead, allow the for loop to automatically call \_\_next\_\_ to advance to the next line on each iteration. The file object’s iterator will do the work of automatically loading lines as you go.

Technically, there is one more piece to the iteration protocol alluded to earlier. When the for loop begins, it first obtains an iterator from the iterable object by passing it to the iter built-in function; the object returned by iter in turn has the required next method. The iter function internally runs the \_\_iter\_\_ method, much like next and \_\_next\_\_.

in some cases these two objects are the same when only a single scan is supported (e.g., files), and the iterator object is often temporary, used internally by the iteration tool.

Moreover, some objects are both an iteration context tool (they iterate) and an iterable object (their results are iterable). generator expressions, and map and zip, range and some dictionary methods.

>>> **L = [1, 2, 3]**

>>> **I = iter(L)** # Obtain an iterator object from an iterable

>>> **I.\_\_next\_\_()** # Call iterator's next to advance to next item

1

a file object is its own iterator. Because they support just one iteration (they can’t seek backward to support multiple active scans), files have their own \_\_next\_\_ method and do not need to return a different object that does.

>>> **f = open('script2.py')**

>>> **iter(f) is f**

True

Lists and many other built-in objects, though, are not their own iterators because they do support multiple open iterations

>>> **L = [1, 2, 3]**

>>> **iter(L) is L**

False

The following interaction demonstrates the equivalence between automatic and manual iteration:[2](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch14.html#CHP-14-FN-2)

>>> **L = [1, 2, 3]**

>>>

>>> **for X in L:** # Automatic iteration

... **print(X \*\* 2, end=' ')** # Obtains iter, calls \_\_next\_\_, catches exceptions

...

1 4 9

>>> **I = iter(L)** # Manual iteration: what for loops usually do

>>> **while True:**

... **try:** # try statement catches exceptions

... **X = next(I)** # Or call I.\_\_next\_\_ in 3.X

... **except StopIteration:**

... **break**

... **print(X \*\* 2, end=' ')**

...

1 4 9

In recent versions of Python, though, dictionaries are iterables with an iterator that automatically returns one key at a time in an iteration context:

>>> **I = iter(D)**

>>> **next(I)**

'a'

>>> **next(I)**

'b'

A generator object (object, which is returned by the generator function) is an iterator also.

*shelves* (an access-by-key filesystem for Python objects) and the results from os.popen (a tool for reading the output of shell commands, which we met in the preceding chapter) are iterable as well:

>>> **import os**

>>> **P = os.popen('dir')**

>>> **P.\_\_next\_\_()**

' Volume in drive C has no label.\n'

>>> **P.\_\_next\_\_()**

Also in the systems domain, the standard directory walker in Python, os.walk, is similarly iterable.

Together with for loops, list comprehensions are one of the most prominent contexts in which the iteration protocol is applied.

Extended list comprehension syntax:

### Filter clauses: if

As one particularly useful extension, the for loop nested in a comprehension expression can have an associated if clause to filter out of the result items for which the test is not true.

Adding an if filter clause to our expression does the trick:

>>> **lines = [line.rstrip() for line in open('script2.py') if line[0] == 'p']**

>>> **lines**

['print(sys.path)', 'print(x \*\* 32)']

nested loops, coded as a series of for clauses. In fact, their full syntax allows for any number of for clauses, each of which can have an optional associated if clause.

>>> **[x + y for x in 'abc' for y in 'lmn']**

['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

Keep in mind, though, that every built-in tool that scans from left to right across objects uses the iteration protocol.

Map: it’s a built-in that applies a function call to each item in the passed-in iterable object. map is similar to a list comprehension but is more limited because it requires a function instead of an arbitrary expression. It also returns an iterable object itself in Python 3.X, so we must wrap it in a list call to force it to give us all its values at once.

Many of Python’s other built-ins process iterables, too. For example, sorted sorts items in an iterable; zip combines items from iterables; enumerate pairs items in an iterable with relative positions; filter selects items for which a function is true; and reduce runs pairs of items in an iterable through a function. All of these accept iterables, and zip, enumerate, and filter also return an iterable in Python 3.X, like map.

unlike map and others, sorted returns an actual list in Python 3.X instead of an iterable.

string join method (which makes a new string by putting a substring between strings contained in an iterable). It uses iteration protocol

both set and dictionary comprehensions support the extended syntax of list comprehensions we met earlier in this chapter, including if tests

Later in the book we’ll meet a relative of comprehensions—generator expressions—that deploys the same syntax and works on iterables too, but is also iterable itself.

such as the range built-in and dictionary view objects, return iterables instead of processing them.

range objects in 3.X support only iteration, indexing, and the len function.

multiple iterators are usually supported by returning new objects for the iter call; a single iterator generally means an object returns itself.

* User-defined functions can be turned into iterable *generator functions*, with yield statements.
* List comprehensions morph into iterable *generator expressions* when coded in parentheses.
* User-defined classes are made iterable with \_\_iter\_\_ or \_\_getitem\_\_ *operator overloading*.

Iteration contexts in Python include the for loop; list comprehensions; the map built-in function; the in membership test expression; and the built-in functions sorted, sum, any, and all. This category also includes the list and tuple built-ins, string join methods, and sequence assignments, all of which use the iteration protocol (see answer #1) to step across iterable objects one item at a time.

# Chapter 15. The Documentation Interlude

the PyDoc system covered here can render a module’s internal documentation as either plain text in a shell, or HTML in a web browser.

documentation strings (docstrings) and the PyDoc system that makes use of them.

| Table 15-1. Python documentation sources | |
| --- | --- |
| **Form** | **Role** |
| # comments | In-file documentation |
| The dir function | Lists of attributes available in objects |
| Docstrings: \_\_doc\_\_ | In-file documentation attached to objects |
| PyDoc: the help function | Interactive help for objects |
| PyDoc: HTML reports | Module documentation in a browser |
| Sphinx third-party tool | Richer documentation for larger projects |
| The standard manual set | Official language and library descriptions |
| Web resources | Online tutorials, examples, and so on |
| Published books | Commercially polished reference texts |

current best practice generally dictates that docstrings are best for larger functional documentation (e.g., “my file does this”), and # comments are best limited to smaller code documentation (e.g., “this strange expression does that”) and are best limited in scope to a statement or small group of statements within a script or function.

the built-in dir function is an easy way to grab a list of all the attributes available inside an object (i.e., its methods and simpler data items). It can be called with no arguments to list variables in the caller’s scope.

>>> **dir([])**

>>> **dir('')**

>>> **[a for a in dir(list) if not a.startswith('\_\_')]**

['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',

'remove', 'reverse', 'sort']

>>> **[a for a in dir(dict) if not a.startswith('\_\_')]**

['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',

'setdefault', 'update', 'values']

""

Module documentation

Words Go Here

"""

spam = 40

def square(x):

"""

function documentation

can we have your liver then?

"""

return x \*\* 2 # square

class Employee:

"class documentation"

pass

print(square(4))

print(square.\_\_doc\_\_)

The whole point of this documentation protocol is that your comments are *retained* for inspection in \_\_doc\_\_ attributes after the file is imported.

To fetch the docstring of a method function inside a class within a module, you would simply extend the path to go through the class: module.class.method.\_\_doc\_\_

common practice today recommends hash-mark comments for only smaller-scale documentation about an expression, statement, or small group of statements. Docstrings are better used for higher-level and broader functional documentation for a file, function, or class, and have become an expected part of Python software. Beyond these guidelines, though, you still must decide what to write.

Python ships with PyDoc in its standard library.

the two most prominent PyDoc interfaces are the built-in help function and the PyDoc GUI- and web-based HTML report interfaces.

>>> **import sys**

>>> **help(sys.getrefcount)**

To display pydoc in the browser: **python -m pydoc –b**

the effect is to start PyDoc as a locally running web server on a dedicated (but by default arbitrary unused) port, and pop up a web browser to act as client, displaying a page giving links to documentation for all the modules importable on your module search path (including the directory where PyDoc is launched).

PyDoc is mostly intended for documenting importable modules, but can sometimes be used to show documentation for scripts too. A selected file must be imported in order to render its documentation, and as we’ve learned, importing runs a file’s code.

You can also run PyDoc to generate a plain-text form of the documentation (its Unix “manpage” flavor shown earlier in this chapter)—the following command line is equivalent to the help call at an interactive Python prompt: c:\code> **py −3 -m pydoc timeit**

c:\code> set PYTHONPATH=.;%PYTHONPATH%

c:\code> py −3.2 -m pydoc -g

This setting was also required to see the current directory for the new all-browser pydoc -b mode in 3.2. However, Python 3.3 automatically includes “.” in its index list, so no path setting is required to view files in the directory where PyDoc is started—a minor but noteworthy improvement.

## Beyond docstrings: Sphinx

If you’re looking for a way to document your Python system in a more sophisticated way, you may wish to check out *Sphinx* (currently at [*http://sphinx-doc.org*](http://sphinx-doc.org/)).

It uses simple reStructuredText as its markup language, and inherits much from the Docutils suite of reStructuredText parsing and translating tools.

Among other things, Sphinx supports a variety of output formats (HTML including Windows HTML Help, LaTeX for printable PDF versions, manual pages, and plain text); extensive and automatic cross-references; hierarchical structure with automatic links to relatives; automatic indexes; automatic code highlighting using Pygments (itself a notable Python tool); and more. This is probably overkill for smaller programs where docstrings and PyDoc may suffice, but can yield professional-grade documentation for large projects

the official Python website (<http://www.python.org>

# Common Coding Gotchas

Before the programming exercises for this part of the book, let’s run through some of the most common mistakes beginners make when coding Python statements and programs. Many of these are warnings I’ve thrown out earlier in this part of the book, collected here for ease of reference. You’ll learn to avoid these pitfalls once you’ve gained a bit of Python coding experience, but a few words now might help you avoid falling into some of these traps initially:

* **Don’t forget the colons**. Always remember to type a : at the end of compound statement headers—the first line of an if, while, for, etc. You’ll probably forget at first (I did, and so have most of my roughly 4,000 Python students over the years), but you can take some comfort from the fact that it will soon become an unconscious habit.
* **Start in column 1**. Be sure to start top-level (unnested) code in column 1. That includes unnested code typed into module files, as well as unnested code typed at the interactive prompt.
* **Blank lines matter at the interactive prompt**. Blank lines in compound statements are always irrelevant and ignored in module files, but when you’re typing code at the interactive prompt, they end the statement. In other words, blank lines tell the interactive command line that you’ve finished a compound statement; if you want to continue, don’t hit the Enter key at the ... prompt (or in IDLE) until you’re really done. This also means you can’t paste multiline code at this prompt; it must run one full statement at a time.
* **Indent consistently**. Avoid mixing tabs and spaces in the indentation of a block, unless you know what your text editor does with tabs. Otherwise, what you see in your editor may not be what Python sees when it counts tabs as a number of spaces. This is true in any block-structured language, not just Python—if the next programmer has tabs set differently, it will be difficult or impossible to understand the structure of your code. It’s safer to use all tabs or all spaces for each block.
* **Don’t code C in Python**. A reminder for C/C++ programmers: you don’t need to type parentheses around tests in if and while headers (e.g., if (X==1):). You can, if you like (any expression can be enclosed in parentheses), but they are fully superfluous in this context. Also, do not terminate all your statements with semicolons; it’s technically legal to do this in Python as well, but it’s totally useless unless you’re placing more than one statement on a single line (the end of a line normally terminates a statement). And remember, don’t embed assignment statements in while loop tests, and don’t use {} around blocks (indent your nested code blocks consistently instead).
* **Use simple** **for** **loops instead of** **while** **or** **range**. Another reminder: a simple for loop (e.g., for x in seq:) is almost always simpler to code and often quicker to run than a while- or range-based counter loop. Because Python handles indexing internally for a simple for, it can sometimes be faster than the equivalent while, though this can vary per code and Python. For code simplicity alone, though, avoid the temptation to count things in Python!
* **Beware of mutables in assignments**. I mentioned this in [Chapter 11](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch11.html#assignments_comma_expressions_comma_and): you need to be careful about using mutables in a multiple-target assignment (a = b = []), as well as in an augmented assignment (a += [1, 2]). In both cases, in-place changes may impact other variables. See [Chapter 11](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch11.html#assignments_comma_expressions_comma_and) for details if you’ve forgotten why this is true.
* **Don’t expect results from functions that change objects in place**. We encountered this one earlier, too: in-place change operations like the list.append and list.sort methods introduced in [Chapter 8](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch08.html#lists_and_dictionaries) do not return values (other than None), so you should call them without assigning the result. It’s not uncommon for beginners to say something like mylist = mylist.append(X) to try to get the result of an append, but what this actually does is assign mylist to None, not to the modified list (in fact, you’ll lose your reference to the list altogether).

A more devious example of this pops up in Python 2.X code when trying to step through dictionary items in a sorted fashion. It’s fairly common to see code like for k in D.keys().sort():. This almost works—the keys method builds a keys list, and the sort method orders it—but because the sort method returns None, the loop fails because it is ultimately a loop over None (a nonsequence). This fails even sooner in Python 3.X, because dictionary keys are views, not lists! To code this correctly, either use the newer sorted built-in function, which returns the sorted list, or split the method calls out to statements: Ks = list(D.keys()), then Ks.sort(), and finally, for k in Ks:. This, by the way, is one case where you may still want to call the keys method explicitly for looping, instead of relying on the dictionary iterators—iterators do not sort.

* **Always use parentheses to call a function**. You must add parentheses after a function name to call it, whether it takes arguments or not (e.g., use function(), not function). In the next part of this book, we’ll learn that functions are simply objects that have a special operation—a call that you trigger with the parentheses. They can be referenced like any other object without triggering a call.

In classes, this problem seems to occur most often with files; it’s common to see beginners type file.close to close a file, rather than file.close(). Because it’s legal to reference a function without calling it, the first version with no parentheses succeeds silently, but it does not close the file!

* **Don’t use extensions or paths in imports and reloads**. Omit directory paths and file extensions in import statements—say import mod, not import mod.py. We discussed module basics in [Chapter 3](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/ch03.html#how_you_run_programs) and will continue studying modules in [Part V](https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/part05.html#modules_and_packages). Because modules may have other extensions besides .py (.pyc, for instance), hardcoding a particular extension is not only illegal syntax, it doesn’t make sense. Python picks an extension automatically, and any platform-specific directory path syntax comes from module search path settings, not the import statement.
* **And other pitfalls in other parts**. Be sure to also see the built-in type warnings at the end of the prior part, as they may qualify as coding issues too. There are additional “gotchas” that crop up commonly in Python coding—losing a built-in function by reassigning its name, hiding a library module by using its name for one of your own, changing mutable argument defaults, and so on

a function is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name.

functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

two ways to make functions (def and lambda), two ways to manage scope visibility (global and nonlocal), and two ways to send results back to callers (return and yield).

| **Statement or expression** | **Examples** |
| --- | --- |
| Call expressions | myfunc('spam', 'eggs', meat=ham, \*rest) |
| def | def printer(message):  print('Hello ' + message) |
| return | def adder(a, b=1, \*c):  return a + b + c[0] |
| global | x = 'old'  def changer():  global x; x = 'new' |
| nonlocal (3.X) | def outer():  x = 'old'  def changer():  nonlocal x; x = 'new' |
| yield | def squares(x):  for i in range(x): yield i \*\* 2 |
| lambda | funcs = [lambda x: x\*\*2, lambda x: x\*\*3] |
| *Function-related statements and expressions* | |

functions serve two primary development roles:

*Maximizing code reuse and minimizing redundancy*

As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Up until now, all the code we’ve been writing has run immediately. Functions allow us to group and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many places, Python functions are the most basic *factoring* tool in the language: they allow us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

*Procedural decomposition*

Functions also provide a tool for splitting systems into pieces that have well-defined roles. one function for each subtask in the process. It’s easier to implement the smaller tasks in isolation than it is to implement the entire process at once. In general, functions are about *procedure*—how to do something, rather than what you’re doing it to.

**def** **is executable code**. Python functions are written with a new statement, the def. Unlike functions in compiled languages such as C, def is an executable statement—your function does not exist until Python reaches and runs the def.

**def** **creates an object and assigns it to a name**. When Python reaches and runs a def statement, it generates a new function object and assigns it to the function’s name. As with all assignments, the function name becomes a reference to the function object. There’s nothing magic about the name of a function—as you’ll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined attributes attached to them to record data.

**lambda** **creates an object but returns it as a result**. Functions may also be created with the lambda expression, a feature that allows us to in-line function definitions in places where a def statement won’t work syntactically.

**return** **sends a result object back to the caller**. When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a return statement; the returned value becomes the result of the function call. A return without a value simply returns to the caller (and sends back None, the default result).

**yield** **sends a result object back to the caller, but remembers where it left off**. Functions known as generators may also use the yield statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time.

* **global** **declares module-level variables that are to be assigned**. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a global statement. More generally, names are always looked up in scopes—places where variables are stored—and assignments bind names to scopes.
* **nonlocal** **declares enclosing function variables that are to be assigned**. Similarly, the nonlocal statement added in Python 3.X allows a function to assign a name that exists in the scope of a syntactically enclosing def statement. This allows enclosing functions to serve as a place to retain state—information remembered between function calls—without using shared global names.

**Arguments are passed by assignment (object reference)**. In Python, arguments are passed to functions by assignment (which, as we’ve learned, means by object reference). As you’ll see, in Python’s model the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.

**Arguments are passed by position, unless you say otherwise**. Values you pass in a function call match argument names in a function’s definition from left to right by default. For flexibility, function calls can also pass arguments by name with *name*=*value* keyword syntax, and unpack arbitrarily many arguments to send with \**pargs* and \*\**kargs* starred-argument notation. Function definitions use the same two forms to specify argument defaults, and collect arbitrarily many arguments received.

**Arguments, return values, and variables are not declared**. As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on.

The def header line specifies a function name that is assigned the function object, along with a list of zero or more arguments (sometimes called parameters) in parentheses. The argument names in the header are assigned to the objects passed in parentheses at the point of call.

The Python return statement can show up anywhere in a function body; when reached, it ends the function call and sends a result back to the caller. The return statement consists of an optional object value expression that gives the function’s result. If the value is omitted, return sends back a None.

Technically, a function without a return statement also returns the None object automatically, but this return value is usually ignored at the call.

Functions may also contain yield statements, which are designed to produce a series of values over time,

Def executes at runtime: a def can appear anywhere a statement can—even nested in other statements. For instance, although defs normally are run when the module enclosing them is imported, it’s also completely legal to nest a function def inside an if statement to select between alternative definitions.

if test:

def func(): # Define func this way

...

else:

def func(): # Or else this way

...

...

func()

defs are not evaluated until they are reached and run, and the code inside defs is not evaluated until the functions are later called.

othername = func # Assign function object

othername() # Call func again

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just *objects*; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary *attributes* to be attached to record information for later use:

def func(): ... # Create function object

func() # Call object

func.attr = value # Attach attributes

there are two sides to the function picture: a definition (the def that creates a function) and a call (an expression that tells Python to run the function’s body).

 \* works on both numbers and sequences.

polymorphism, a term essentially means that the meaning of an operation depends on the objects being operated upon. Because it’s a dynamically typed language, polymorphism runs rampant in Python. In fact, every operation is a polymorphic operation in Python

A single function, for instance, can generally be applied to a whole category of object types automatically. As long as those objects support the expected interface (a.k.a. protocol), the function can process them. That is, if the objects passed into a function have the expected methods and expression operators, they are plug-and-play compatible with the function’s logic.

if the objects passed in do not support this expected interface, Python will detect the error when the \* expression is run and raise an exception automatically.

(When an object is created inside a function, and its returned, the object is referenced by a new variable. Local variable goes away)

Because of this, a function’s variables won’t remember values between calls; although the object returned by a function lives on, retaining other sorts of state information requires other sorts of techniques.

This polymorphic behavior has in recent years come to also be known as duck typing—the essential idea being that your code is not supposed to care if an object is a duck, only that it quacks. Anything that quacks will do, duck or not.

# Chapter 17. Scopes

scopes—the places where variables are defined and looked up. Like module files, scopes help prevent name clashes across your program’s code: names defined in one program unit don’t interfere with names in another.

scope usage can have a major impact on program maintenance effort; overuse of globals, for example, is a generally bad thing. On the plus side, we’ll learn that scopes can provide a way to retain state information between function calls, and offer an alternative to classes in some roles.

When you use a name in a program, Python creates, changes, or looks up the name in what is known as a namespace—a place where names live. When we talk about the search for a name’s value in relation to code, the term scope refers to a namespace: that is, the location of a name’s assignment in your source code determines the scope of the name’s visibility to your code.

names in Python spring into existence when they are first assigned values, and they must be assigned before they are used. Because names are not declared ahead of time, Python uses the location of the assignment of a name to associate it with (i.e., bind it to) a particular namespace. the place where you assign a name in your source code determines the namespace it will live in, and hence its scope of visibility.

the scope of a variable (where it can be used) is always determined by where it is assigned in your source code and has nothing to do with which functions call which

* If a variable is assigned inside a def, it is local to that function.
* If a variable is assigned in an enclosing def, it is nonlocal to nested functions.
* If a variable is assigned outside all defs, it is global to the entire file.

We call this lexical scoping because variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls.

Technically, the interactive prompt is a module named \_\_main\_\_ it’s like the top level of a module file.

Functions define a local scope and modules define a global scope.

Global variables become attributes of a module object to the outside world after imports but can also be used as simple variables within the module file itself.

in-place changes to objects do not classify names as locals; only actual name assignments do. changing an object is not an assignment to a name.

L = list()

L.append([1,2,3])

def func():

L.append([4,5,6]) # modifies the object inplace

func()

print(L)

output: [[1, 2, 3], [4, 5, 6]]

--

LEGB rule: Within a def statement:

* Name *assignments* create or change local names by default.
* Name *references* search at most four scopes: local, then enclosing functions (if any), then global, then built-in.
* Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes, respectively.

there are technically three more scopes in Python—temporary loop variables in some comprehensions, exception reference variables in some try handlers, and local scopes in class statements. The first two of these are special cases that rarely impact real code, and the third falls under the LEGB umbrella rule.

Comprehension variables are local to the expression itself in all comprehension forms: generator, list, set, and dictionary.

for loop statements never localize their variables to the statement block in any Python.

Exception variables are local to that except block, and in fact are removed when the block is exited.

names assigned inside a class don’t clash with names elsewhere, and follow the LEGB lookup rule, where the class block is the “L” level.

Unlike functions, though, class names are not created per call: class object calls generate instances, which inherit names assigned in the class and record per-object state as attributes.

the built-in scope is just a built-in module called builtins, but you have to import builtins to query built-ins because the name builtins is not itself built in...

>>> **import builtins**

>>> **dir(builtins)**

don’t redefine a built-in name you need.

third-party tools such as PyChecker, and others such as PyLint, will warn about common programming mistakes, including accidental assignment to built-in names (this is usually known as “shadowing” a built-in in such tools). It’s not a bad idea to run your first few Python programs through tools like these to see what they point out.

* Global names are variables assigned at the top level of the enclosing module file.
* Global names must be declared only if they are assigned within a function.
* Global names may be referenced within a function without being declared.

In general, functions should rely on arguments and return values instead of globals

global variables are probably the most straightforward way in Python to retain shared state information—information that a function needs to remember for use the next time it is called.

some programs designate a single module to collect globals; as long as this is expected, it is not as harmful. Programs that use multithreading to do parallel processing in Python also commonly depend on global variables—they become shared memory between functions running in parallel threads, and so act as a communication device.

 each module is a self-contained namespace (package of variables), and we must import one module to see inside it from another.

importers automatically have access to all of the file’s global variables, because a file’s global scope morphs into an object’s attribute namespace when it is imported.

the best way to communicate across file boundaries is to call functions, passing in arguments and getting back return values.

 Enclosing scopes are sometimes also called statically nested scopes.

closure or a factory function—the former describing a functional programming technique, and the latter denoting a design pattern. Whatever the label, the function object in question remembers values in enclosing scopes regardless of whether those scopes are still present in memory. In effect, they have attached packets of memory (a.k.a. state retention), which are local to each copy of the nested function created, and often provide a simple alternative to classes in this role.

Factory functions (a.k.a. closures) are sometimes used by programs that need to generate event handlers on the fly in response to conditions at runtime.

>>> def maker(X):

def action(N):

return X \* N

return action

>>> f = maker(4)

>>> f(3)

12

>>> **def maker(N):**

**return lambda X: X \*\* N**# lambda functions retain state too

>>> **h = maker(3)**

>>> **h(4)**

From a broader perspective, there are multiple ways for Python functions to retain state between calls. Although the values of normal local variables go away when a function returns, values can be retained from call to call in global variables; in class instance attributes; in the enclosing scope references we’ve met here; and in argument defaults and function attributes.

###### NOTE

Closures can also be created when a class is nested in a def: the values of the enclosing function’s local names are retained by references within the class, or one of its method functions. the outer def in such code serves a similar role: it becomes a class factory, and provides state retention for the nested class.

programmers typically used *default argument values* to pass in and remember the objects in an enclosing scope:

def f1():

x = 88

def f2(x=x): # Remember enclosing scope X with defaults

print(x)

f2()

f1() # Prints 88

Python automatically remembers any values required in the enclosing scope for use in nested defs.

Lambda: it’s an expression that generates a new function to be called later, much like a def statement. Because it’s an expression, though, it can be used in places that def cannot, such as within list and dictionary literals.

Like a def, a lambda expression also introduces a new local scope for the function it creates. Thanks to the enclosing scopes lookup layer, lambdas can see all the variables that live in the functions in which they are coded.

def func():

x = 4

action = (lambda n: x \*\* n) # x remembered from enclosing def

return action

x = func()

print(x(2)) # Prints 16, 4 \*\* 2

in Python, we say flat is better than nested

nonlocal variables must exist in the enclosing function when declared. Globals not required to exist when declared.

we could also make our class objects look like callable functions using operator overloading. \_\_call\_\_ intercepts direct calls on an instance

class attributes are always changeable and don’t require a nonlocal statement, and classes are designed to scale up to implementing richer objects with many attributes and behaviors.

Function attribute in nested function:

>>> **def tester(start):**

**def nested(label):**

**print(label, nested.state)** # nested is in enclosing scope

**nested.state += 1** # Change attr, not nested itself

**nested.state = start** # Initial state after func defined

**return nested**

>>> **F = tester(0)**

>>> **F('spam')** # F is a 'nested' with state attached

spam 0

>>> **F('ham')**

ham 1

>>> **F.state** # Can access state outside functions too

2

Chapter 18. Arguments

Python argument passing—the way that objects are sent to functions as inputs.

Python provides extra tools, such as keywords, defaults, and arbitrary argument collectors and extractors that allow for wide flexibility in the way arguments are sent to a function

Function arguments—references to (possibly) shared objects sent by the caller—

xcross