Chapter 17. Testing, Debugging, and Optimizing

You're not finished with a programming task when you're done writing the code; you're finished only when the code runs correctly and with acceptable performance. *Testing* means verifying that code runs correctly by automatically exercising the code under known conditions and checking that the results are as expected. *Debugging* means discovering causes of incorrect behavior and repairing them (repair is often easy, once you figure out the causes).

Optimizing is often used as an umbrella term for activities meant to ensure acceptable performance. Optimizing breaks down into *benchmarking* (measuring performance for given tasks to check that it's within acceptable bounds), *profiling* (*instrumenting* the program with extra code to identify performance bottlenecks), and actual optimizing (removing bottlenecks to improve program performance). Clearly, you can't remove performance bottlenecks until you've found out where they are (via profiling), which in turn requires knowing that there *are* performance problems (via benchmarking).

This chapter covers these subjects in the natural order in which they occur in development: testing first and foremost, debugging next, and optimizing last. Most programmers' enthusiasm focuses on optimization: testing and debugging are often (wrongly!) perceived as being chores, while optimization is seen as being fun. Were you to read only one section of the chapter, we might suggest that section be **"Developing a Fast-Enough Python Application"**, which summarizes the Pythonic approach to optimization—close to Jackson's classic "**Rules of Optimization**: Rule 1. Don't do it. Rule 2 (for experts only). Don't do it *yet*."

All of these tasks are important; discussion of each could fill at least a book by itself. This chapter cannot even come close to exploring every related technique; rather, it focuses on Python-specific approaches and tools. Often, for best results, you should approach the issue from the higher-level viewpoint of *system analysis and design*, rather than focusing only on implementation (in Python and/or any other mix of programming languages). Start by studying a good general book on the subject, such as *Systems Analysis and Design* by Alan Dennis, Barbara Wixom, and Roberta Roth (Wiley).

Testing

In this chapter, we distinguish between two different kinds of testing: *unit testing* and *system testing*. Testing is a rich, important field: many more distinctions could be drawn, but we focus on the issues that most matter to most software developers. Many developers are reluctant to spend time on testing, seeing it as time stolen from "real" development, but this is shortsighted: problems in code are easier to fix the earlier you find out about them. An extra hour spent developing tests will amply pay for itself as you find defects early, saving you many hours of debugging that would otherwise have been needed in later phases of the software development cycle.[1]

Unit Testing and System Testing

Unit testing means writing and running tests to exercise a single module, or an even smaller unit, such as a class or function. *System testing* (also known as *functional*, *integration*, or *end-to-end* testing) involves running an entire program with known inputs. Some classic books on testing also draw the distinction between *white-box testing*, done with knowledge of a program's internals, and *black-box testing*, done without such knowledge. This classic viewpoint parallels, but does not exactly duplicate, the modern one of unit versus system testing.

Unit and system testing serve different goals. Unit testing proceeds apace with development; you can and should test each unit as you're developing it. One relatively modern approach (first proposed in 1971 in Gerald

Weinberg's immortal classic *The Psychology of Computer Programming* [Dorset House]) is known as *test-driven development* (TDD): for each feature that your program must have, you first write unit tests, and only then do you proceed to write code that implements the feature and makes the tests pass. TDD may seem upside down, but it has advantages; for example, it ensures that you won't omit unit tests for some feature. This approach is helpful because it urges you to focus first on exactly *what tasks* a certain function, class, or method should accomplish, dealing only afterward with *how* to implement that function, class, or method. An innovation along the lines of TDD is ***behavior-driven development* (BDD)**.

To test a unit—which may depend on other units not yet fully developed—you often have to write *stubs*, also known as *mocks*:[2] fake implementations of various units' interfaces giving known, correct responses in cases needed to test other units. The `mock` module (part of Python's standard library, in the package **`unittest`**) helps you implement such stubs.

System testing comes later, since it requires the system to exist, with at least some subset of system functionality believed (based on unit testing) to be working. System testing offers a soundness check: each module in the program works properly (passes unit tests), but does the *whole* program work? If each unit is OK but the system is not, there's a problem in the integration between units—the way the units cooperate. For this reason, system testing is also known as integration testing.

System testing is similar to running the system in production use, except that you fix inputs in advance so that any problems you may find are easy to reproduce. The cost of failures in system testing is lower than in production use, since outputs from system testing are not used to make decisions, serve customers, control external systems, and so on. Rather, outputs from system testing are systematically compared with the outputs that the system *should* produce given the known inputs. The purpose is to find, in cheap and reproducible ways, discrepancies between what the program *should* do and what the program actually *does*.

Failures discovered by system testing (just like system failures in production use) may reveal defects in unit tests, as well as defects in the code.

Unit testing may have been insufficient: a module's unit tests may have failed to exercise all the needed functionality of the module. In that case, the unit tests need to be beefed up. Do that *before* you change your code to fix the problem, then run the newly enhanced unit tests to confirm that they now show the problem. Then fix the problem, and run the unit tests again to confirm that they no longer show it. Finally, rerun the system tests to confirm that the problem has indeed gone away.

BUG-FIXING BEST PRACTICE

This best practice is a specific application of test-driven design that we recommend without reservation: never fix a bug before having added unit tests that would have revealed the bug. This provides an excellent, cheap insurance against **software regression bugs**.

Often, failures in system testing reveal communication problems within the development team:[3] a module correctly implements a certain functionality, but another module expects different functionality. This kind of problem (an integration problem in the strict sense) is hard to pinpoint in unit testing. In good development practice, unit tests must run often, so it is crucial that they run fast. It's therefore essential, in the unit testing phase, that each unit can assume other units are working correctly and as expected.

Unit tests run in reasonably late stages of development can reveal integration problems if the system architecture is hierarchical, a common and reasonable organization. In such an architecture, low-level modules depend on no others (except library modules, which you can typically assume to be correct), so the unit tests of such low-level modules, if complete, suffice to provide confidence of correctness. High-level modules depend on low-level ones, and thus also depend on correct understanding about what functionality each module expects and supplies. Running complete unit tests on high-level modules (using true low-level modules, not stubs) exercises interfaces between modules, as well as the high-level modules' own code.

Unit tests for high-level modules are thus run in two ways. You run the tests with stubs for the low levels during the early stages of development, when the low-level modules are not yet ready or, later, when you only need to check the correctness of the high levels. During later stages of development, you also regularly run the high-level modules' unit tests using the true low-level modules. In this way, you check the correctness of the whole subsystem, from the high levels downward. Even in this favorable case, you *still* need to run system tests to ensure that you have checked that all of the system's functionality is exercised and you have neglected no interfaces between modules.

System testing is similar to running the program in normal ways. You need special support only to ensure supply of known inputs and capture of resulting outputs for comparison with expected outputs. This is easy for programs that perform I/O on files, and hard for programs whose I/O relies on a GUI, network, or other communication with external entities. To simulate such external entities and make them predictable and entirely observable, you generally need platform-dependent infrastructure. Another useful piece of supporting infrastructure for system testing is a *testing framework* to automate the running of system tests, including logging of successes and failures. Such a framework can also help testers prepare sets of known inputs and corresponding expected outputs.

Both free and commercial programs for these purposes exist, and usually do not depend on which programming languages are used in the system under test. System testing is a close relative of what was classically known as black-box testing: testing that is independent from the implementation of the system under test (and thus, in particular, independent from the programming languages used for implementation). Instead, testing frameworks usually depend on the operating system platform on which they run, since the tasks they perform are platform dependent. These include:

- Running programs with given inputs
- Capturing their outputs
- Simulating/capturing GUI, network, and other interprocess communication I/O

Since frameworks for system testing depend on the platform, not on programming languages, we do not cover them further in this book. For a thorough list of Python testing tools, see the Python wiki.

The doctest Module

The `doctest` module exists to let you create good examples in your code's docstrings, checking that the examples do in fact produce the results that your docstrings show for them. `doctest` recognizes such examples by looking within the docstring for the interactive Python prompt `>>>`, followed on the same line by a Python statement, and the statement's expected output on the next line(s).

As you develop a module, keep the docstrings up-to-date and enrich them with examples. Each time a part of the module (e.g., a function) is ready, or partially ready, make it a habit to add examples to its docstring. Import the module into an interactive session, and use the parts you just developed to provide examples with a mix of typical cases, limit cases, and failing cases. For this specific purpose only, use **`from`** *`module`* **`import`** `*` so that your examples don't prefix *`module`*`.` to each name the module supplies. Copy and paste the interactive session into the docstring in an editor, adjust any glitches, and you're almost done.

Your documentation is now enriched with examples, and readers will have an easier time following it (assuming you choose a good mix of examples, wisely seasoned with nonexample text). Make sure you have docstrings, with examples, for the module as a whole, and for each function, class, and method the module exports. You may choose to skip functions, classes, and methods whose names start with `_`, since (as their names indicate) they're private implementation details; `doctest` by default ignores them, and so should readers of your module's source code.

MAKE YOUR EXAMPLES MATCH REALITY

Examples that don't match the way your code works are worse than useless. Documentation and comments are useful only if they match reality; docs and comments that "lie" can be seriously damaging.

Docstrings and comments often get out of date as code changes, and thus become misinformation, hampering, rather than helping, any reader of the source. It's better to have no comments and docstrings at all, poor as such a choice would be, than to have ones that lie. `doctest` can help you by running and checking the examples in your docstrings. A failing `doctest` run should prompt you to review the docstring that contains the failing example, thus reminding you to keep the whole docstring updated.

At the end of your module's source, insert the following snippet:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

This code calls the function `testmod` of the module `doctest` when you run your module as the main program. `testmod` examines docstrings (the module's docstring, and the docstrings of all public functions, classes, and methods thereof). In each docstring, `testmod` finds all examples (by looking for occurrences of the interpreter prompt `>>>`, possibly preceded by whitespace) and runs each example. `testmod` checks that each example's results match the output given in the docstring right after the example. In case of exceptions, `testmod` ignores the traceback, and just checks that the expected and observed error messages are equal.

When everything goes right, `testmod` terminates silently. Otherwise, it outputs detailed messages about the examples that failed, showing expected and actual output. Example 17-1 shows a typical example of `doctest` at work on a module *mod.py*.

Example 17-1. Using `doctest`

```
"""
This module supplies a single function reverse_words that reverses
a string word by word.

>>> reverse_words('four score and seven years')
```

```
'years seven and score four'
>>> reverse_words('justoneword')
'justoneword'
>>> reverse_words('')
''

You must call reverse_words with a single argument, a string:

>>> reverse_words()
Traceback (most recent call last):
    ...
TypeError: reverse_words() missing 1 required positional argument: 'astring'
>>> reverse_words('one', 'another')
Traceback (most recent call last):
    ...
TypeError: reverse_words() takes 1 positional argument but 2 were given
>>> reverse_words(1)
Traceback (most recent call last):
    ...
AttributeError: 'int' object has no attribute 'split'
>>> reverse_words('𝓤𝓷𝓲𝓬𝓸𝓭𝓮 is all right too')
'too right all is 𝓤𝓷𝓲𝓬𝓸𝓭𝓮'

As a side effect, reverse_words eliminates any redundant spacing:

>>> reverse_words('with  redundant   spacing')
'spacing redundant with'
"""


def reverse_words(astring):
    words = astring.split()
    words.reverse()
    return ' '.join(words)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

In this module's docstring, we snipped the tracebacks from the docstring and replaced them with ellipses (`...`): this is good practice, since `doctest` ignores tracebacks, which add nothing to the explanatory value of a failing case. Apart from this snipping, the docstring is the copy and paste of an interactive session, plus some explanatory text and empty lines for readability. Save this source as *mod.py*, and then run it with **`python mod.py`**. It produces no output, meaning that all the examples work right. Try **`python mod.py -v`** to get an account of all tests it tries, and a verbose summary at the end. Finally, alter the example results in the module docstring, making them incorrect, to see the messages `doctest` provides for errant examples.

While `doctest` is not meant for general-purpose unit testing, it can be tempting to use it for that purpose. The recommended way to do unit testing in Python is with a test framework such as `unittest`, `pytest`, or `nose2` (covered in the following sections). However, unit testing with `doctest` can be easier and faster to set up, since it requires little more than copying and pasting from an interactive session. If you need to maintain a module that lacks unit tests, retrofitting such tests into the module with `doctest` is a reasonable short-term compromise, as a first step. It's better to have just `doctest`-based unit tests than not to have any unit tests at all, as might otherwise happen should you decide that setting up tests properly with `unittest` from the start would take you too long.[4]

If you do decide to use `doctest` for unit testing, don't cram extra tests into your module's docstrings. This would damage the docstrings by making them too long and hard to read. Keep in the docstrings the right amount and kind of examples, strictly for explanatory purposes, just as if unit testing were not in the picture. Instead, put the extra tests into a global variable of your module, a dictionary named `__test__`. The keys in `__test__` are strings to use as arbitrary test names; corresponding values are strings that `doctest` picks up and uses just like it uses docstrings. The values in `__test__` may also be function and class objects, in which case `doctest` examines their docstrings for tests to run. This latter feature is a convenient way to run `doctest` on objects with private names, which `doctest` skips by default.

The `doctest` module also supplies two functions that return instances of the `unittest.TestSuite` class based on doctests, so that you can integrate such tests into testing frameworks based on `unittest`. Full documentation for this advanced functionality is available **online**.

The unittest Module

The unittest module is the Python version of a unit testing framework originally developed by Kent Beck for Smalltalk. Similar, widespread versions of the framework also exist for many other programming languages (e.g., the JUnit package for Java) and are often collectively referred to as xUnit.

To use unittest, don't put your testing code in the same source file as the tested module: rather, write a separate test module for each module to test. A popular convention is to name the test module like the module being tested, with a prefix such as 'test_', and put it in a subdirectory of the source's directory named *test*. For example, the test module for *mod.py* can be *test/test_mod.py*. A simple, consistent naming convention helps you write and maintain auxiliary scripts that find and run all unit tests for a package.

Separation between a module's source code and its unit testing code lets you refactor the module more easily, including possibly recoding some functionality in C without perturbing the unit testing code. Knowing that *test_mod.py* stays intact, whatever changes you make to *mod.py*, enhances your confidence that passing the tests in *test_mod.py* indicates that *mod.py* still works correctly after the changes.

A unit testing module defines one or more subclasses of unittest's TestCase class. Each such subclass specifies one or more test cases by defining *test case methods*: methods that are callable without arguments and whose names start with test.

The subclass usually overrides setUp, which the framework calls to prepare a new instance just before each test case, and often also tearDown, which the framework calls to clean things up right after each test case; the entire setup/teardown arrangement is known as a *test fixture*.

Each test case calls, on self, methods of the class TestCase whose names start with assert to express the conditions that the test must meet. unittest runs the test case methods within a TestCase subclass in arbitrary order, each on a new instance of the subclass, running `setUp` just before each test case and `tearDown` just after each test case.

HAVE SETUP USE ADDCLEANUP WHEN NEEDED

When `setUp` propagates an exception, `tearDown` does not execute. So, when `setUp` prepares several things needing eventual cleanup, and some preparation steps might cause uncaught exceptions, it should not rely on `tearDown` for the cleanup work. Instead, right after each preparation step succeeds, call `self.addCleanup(`*`f`*`, *`*`a`*`, **`*`k`*`)`, passing a cleanup callable *`f`* (and optionally positional and named arguments for *`f`*). In this case, *`f`*`(*`*`a`*`, **`*`k`*`)` does get called after the test case (after `tearDown` when `setUp` propagates no exception, but, unconditionally, even when `setUp` does propagate exceptions), so the necessary cleanup code always executes.

`unittest` provides other facilities, such as grouping test cases into test suites, per-class and per-module fixtures, test discovery, and other, even more advanced functionality. You do not need such extras unless you're building a custom unit testing framework on top of `unittest`, or, at the very least, structuring complex testing procedures for equally complex packages. In most cases, the concepts and details covered in this section are enough to perform effective and systematic unit testing. Example 17-2 shows how to use `unittest` to provide unit tests for the module *mod.py* of Example 17-1. This example, for purely demonstrative purposes, uses `unittest` to perform exactly the same tests that Example 17-1 uses as examples in docstrings using `doctest`.

Example 17-2. Using `unittest`

```
"""This module tests function reverse_words
provided by module mod.py."""
import unittest
import mod

class ModTest(unittest.TestCase):

    def testNormalCaseWorks(self):
        self.assertEqual(
            'years seven and score four',
            mod.reverse_words('four score and seven years'))

    def testSingleWordIsNoop(self):
        self.assertEqual(
            'justoneword',
            mod.reverse_words('justoneword'))

    def testEmptyWorks(self):
        self.assertEqual('', mod.reverse_words(''))

    def testRedundantSpacingGetsRemoved(self):
        self.assertEqual(
            'spacing redundant with',
            mod.reverse_words('with   redundant   spacing'))

    def testUnicodeWorks(self):
        self.assertEqual(
            'too right all is 𝓤𝓷𝓲𝓬𝓸𝓭𝓮'
            mod.reverse_words('𝓤𝓷𝓲𝓬𝓸𝓭𝓮 is all right too'))

    def testExactlyOneArgumentIsEnforced(self):
        with self.assertRaises(TypeError):
            mod.reverse_words('one', 'another')

    def testArgumentMustBeString(self):
        with self.assertRaises((AttributeError, TypeError)):
            mod.reverse_words(1)

if __name__=='__main__':
    unittest.main()
```

Running this script with **`python test/test_mod.py`** (or, equivalently, **`python -m test.test_mod`**) is just a bit more verbose than using **`python mod.py`** to run doctest, as in Example 17-1. *test_mod.py* outputs a `.` (dot) for each test case it runs, then a separator line of dashes, and finally a summary line, such as “Ran 7 tests in 0.110s,” and a final line of “OK” if every test passed.

Each test case method makes one or more calls to methods whose names start with `assert`. Here, no method has more than one such call; in more complicated cases, however, multiple calls to assert methods from a single test case method are common.

Even in a case as simple as this, one minor aspect shows that, for unit testing, `unittest` is more powerful and flexible than `doctest`. In the method `testArgumentMustBeString`, we pass as the argument to `assertRaises` a pair of exception classes, meaning we accept either kind of exception. *test_mod.py* therefore accepts these as valid multiple implementations of *mod.py*. It accepts the implementation in **Example 17-1**, which tries calling the method `split` on its argument, and therefore raises `AttributeError` when called with an argument that is not a string. However, it also accepts a different hypothetical implementation, one that raises `TypeError` instead when called with an argument of the wrong type. It is possible to code such checks with `doctest`, but it would be awkward and nonobvious, while `unittest` makes it simple and natural.

This kind of flexibility is crucial for real-life unit tests, which to some extent are executable specifications for their modules. You could, pessimistically, view the need for test flexibility as meaning the interface of the code you're testing is not well-defined. However, it's best to view the interface as being defined with a useful amount of flexibility for the implementer: under circumstance *X* (argument of invalid type passed to function `reverse_words`, in this example), either of two things (raising `AttributeError` or `TypeError`) is allowed to happen.

Thus, implementations with either of the two behaviors are correct: the implementer can choose between them on the basis of such considerations as performance and clarity. Viewed in this way—as executable specifications for their modules (the modern view, and the basis of test-driven development), rather than as white-box tests strictly constrained to a specific implementation (as in some traditional taxonomies of testing)—unit tests become an even more vital component of the software development process.

The TestCase class

With `unittest`, you write test cases by extending `TestCase`, adding methods, callable without arguments, whose names start with `test`. These test case methods, in turn, call methods that your class inherits from `TestCase`, whose names start with `assert`, to indicate conditions that must hold for the tests to succeed.

The `TestCase` class also defines two methods that your class can optionally override to group actions to perform right before and after each test case method runs. This doesn't exhaust `TestCase`'s functionality, but you won't need the rest unless you're developing testing frameworks or performing other advanced tasks. Table 17-1 lists the frequently called methods of a `TestCase` instance *`t`*.

Table 17-1. Methods of an instance `t` of `TestCase`

`assertAlmost Equal`	`assertAlmostEqual(`*`first`*`, `*`second`*`, places=7, ms` Fails and outputs `msg` when *`first`* `!=` *`second`* to with decimal digits; otherwise, does nothing. This metho than `assertEqual` to compare `float`s, since they ar approximations that may differ in less significant d digits. When producing diagnostic messages if the t `unittest` will assume that *`first`* is the expected val *`second`* is the observed value.
`assertEqual`	`assertEqual(`*`first`*`, `*`second`*`, msg=`**`None`**`)` Fails and outputs `msg` when *`first`* `!=` *`second`*; otherv nothing. When producing diagnostic messages if th `unittest` will assume that *`first`* is the expected val *`second`* is the observed value.
`assertFalse`	`assertFalse(`*`condition`*`, msg=None)` Fails and outputs `msg` when *`condition`* is true; other nothing.

`assertNotAlmostEqual`	`assertNotAlmostEqual(`*`first`*`,` *`second`*`, places=7,` Fails and outputs `msg` when *`first`* `==` *`second`* to with decimal digits; otherwise, does nothing.
`assertNotEqual`	`assertNotEqual(`*`first`*`,` *`second`*`, msg=`**`None`**`)` Fails and outputs `msg` when *`first`* `==` *`second`*; otherv nothing.

`assertRaises`	`assertRaises(`*`exceptionSpec`*`,` *`callable`*`,` *`*args`*`, *` Calls *`callable`*`(`*`*args`*`,` *`**kwargs`*`)`. Fails when the c raise any exception. When the call raises an excepti not meet *`exceptionSpec`*, `assertRaises` propagates exception. When the call raises an exception that m *`exceptionSpec`*, `assertRaises` does nothing. *`except`* be an exception class or a tuple of classes, just like t argument of the **`except`** clause in a **`try`**`/`**`except`** stat The preferred way to use `assertRaises` is as a cont —that is, in a **`with`** statement:
	
	 `with self.assertRaises(exceptionSpec):`
	`    # ...a block of code...`
	
	 Here, the block of code indented in the **`with`** statem rather than just the *`callable`* being called with cert arguments. The expectation (which avoids the cons is that the block of code raises an exception meeting exception specification (an exception class or a tupl This alternative approach is more general and read passing a callable.

`assertRaisesRegex`	`assertRaisesRegex(`*`exceptionSpec`*`,` *`expected_reg`* *`callable`*`,` *`*args`*`,` *`**kwargs`*`)` Just like `assertRaises`, but also checks that the exce error message matches *`regex`*; *`regex`* can be a regul expression object or a string pattern to compile into

the test (when the expected exception has been rais
the error message by calling `search` on the RE objec
Just like `assertRaises`, `assertRaisesRegex` is best u
context manager—that is, in a **`with`** statement:

```
with self.assertRaisesRegex(exceptionSpec,
    # ...a block of code...
```

`enterContext`

`enterContext(ctx_manager)`
3.11+ Use this call in a `TestCase.setup()` method.
value from calling `ctx_manager.__enter__`, and ad
`ctx_manager.__exit__` to the list of cleanup metho
framework is to run during the `TestCase`'s cleanup

`fail`

`fail(msg=None)`
Fails unconditionally and outputs `msg`. An example
might be:

```
if not complex_check_if_its_ok(some, thing
    self.fail(
      'Complex checks failed on'
      f' {some}, {thing}'
    )
```

`setUp`

`setUp()`
The framework calls *`t`*`.setUp()` just before calling e
case method. `setUp` in `TestCase` does nothing; it exi
let your class override the method when your class
perform some preparation for each test.

`subTest`

`subTest(msg=None, **`*`k`*`)`
Returns a context manager that can define a portio

	within a test method. Use `subTest` when a test meth same test multiple times with varying parameters. I these parameterized tests in `subTest` ensures that a will be run, even if some of them fail or raise excep
`tearDown`	`tearDown()` The framework calls *`t`*`.tearDown()` just after each t method. `tearDown` in the base `TestCase` class does n exists only to let your class override the method wh class needs to perform some cleanup after each test

In addition, a `TestCase` instance maintains a LIFO stack of *cleanup functions*. When code in one of your tests (or in `setUp`) does something that requires cleanup, call `self.addCleanup`, passing a cleanup callable *`f`* and optionally positional and named arguments for *`f`*. To perform the stacked cleanups, you may call `doCleanups`; however, the framework itself calls `doCleanups` after `tearDown`. Table 17-2 lists the signatures of the two cleanup methods of a `TestCase` instance *`t`*.

Table 17-2. Cleanup methods of an instance `t` of `TestCase`

`addCleanup`	`addCleanup(`*`func`*`, *`*`a`*`, **`*`k`*`)` Appends `(`*`func`*`, `*`a`*`, `*`k`*`)` at the end of the cleanups list.

`doCleanups`	`doCleanups()`
	Performs all cleanups, if any are stacked. Substantially equivalent to:
	`while self.list_of_cleanups:`
	`    func, a, k = self.list_of_cleanups.pop()`
	`    func(*a, **k)`
	for a hypothetical stack `self.list_of_cleanups`, plus, of course, error checking and reporting.

Unit tests dealing with large amounts of data

Unit tests must be fast, as you should run them often as you develop. So, when feasible, unit test each aspect of your modules on small amounts of data. This makes your unit tests faster, and lets you embed the data in the test's source code. When you test a function that reads from or writes to a file object, use an instance of the class `io.TextIO` for a text file (or `io.BytesIO` for a binary file, as covered in **"In-Memory Files: io.StringIO and io.BytesIO"**) to get a file with the data in memory: this approach is faster than writing to disk, and it requires no cleanup (removing disk files after the tests).

In rare cases, it may be impossible to exercise a module's functionality without supplying and/or comparing data in quantities larger than can be reasonably embedded in a test's source code. In such cases, your unit test must rely on auxiliary, external data files to hold the data to supply to the module it tests, and/or the data it needs to compare to the output. Even then, you're generally better off using instances of the abovementioned `io` classes, rather than directing the tested module to perform actual disk I/O. Even more importantly, we strongly suggest that you generally use stubs to unit test modules that interact with external entities, such as databases, GUIs, or other programs over a network. It's easier to control all aspects of the test when using stubs rather than real external entities. Also, to reiterate, the speed at which you can run unit tests is important, and it's faster to perform simulated operations with stubs than real operations.

MAKE TEST RANDOMNESS REPRODUCIBLE BY SUPPLYING A SEED

If your code uses pseudorandom numbers (e.g., as covered in **"The random Module"**), you can make it easier to test by ensuring its "random" behavior is *reproducible*: specifically, ensure that it's easy for your tests to call `random.seed` with a known argument, so that the ensuing pseudorandom numbers are fully predictable. This also applies when you use pseudorandom numbers to set up your tests by generating inputs: such generation should default to a known seed, to be used in most testing, keeping the flexibility of changing seeds only for specific techniques such as **fuzzing**.

Testing with nose2

nose2 is a pip-installable third-party test utility and framework that builds on top of unittest to provide additional plug-ins, classes, and decorators to aid in writing and running your test suite. nose2 will “sniff out” test cases in your project, building its test suite by looking for unittest test cases stored in files named *test*.py*.

Here is an example of using nose2’s params decorator to pass data parameters to a test function:

```
import unittest
from nose2.tools import params

class TestCase(unittest.TestCase):

    @params((5, 5), (-1, 1), ('a', None, TypeError))
    def test_abs_value(self, x, expected, should_raise=None):
        if should_raise is not None:
            with self.assertRaises(should_raise):
                abs(x)
        else:
            assert abs(x) == expected
```

nose2 also includes additional decorators, the such context manager to define groups of test functions, and a plug-in framework to provide testing metafunctions such as logging, debugging, and coverage reporting. For more information, see the online docs.

Testing with pytest

The pytest module is a pip-installable third-party unit testing framework that introspects a project’s modules to find test cases in *test_*.py* or **_test.py* files, with method names starting with test at the module level, or in classes with names starting with Test. Unlike the built-in unittest framework, pytest does not require that test cases extend any testing class hierarchy; it runs the discovered test methods, which use Python

`assert` statements to determine the success or failure of each test.[5] If a test raises any exception other than `AssertionError`, that indicates that there is an error in the test, rather than a simple test failure.

In place of a hierarchy of test case classes, `pytest` provides a number of helper methods and decorators to simplify writing unit tests. The most common methods are listed in **Table 17-3**; consult the **online docs** for a more complete list of methods and optional arguments.

Table 17-3. Commonly used `pytest` methods

`approx`	`approx(`*`float_value`*`)`
	 Used to support asserts that must compare floating-point va *`float_value`* can be a single value or a sequence of values:
	 `assert 0.1 + 0.2 == approx(0.3)`
	 `assert [0.1, 0.2, 0.1+0.2] == approx([0.1, 0.2, 0`

`fail`	`fail(`*`failure_reason`*`)` Forces failure of the current test. More explicit than injectin **`assert False`** statement, but otherwise equivalent.
`raises`	`raises(`*`expected_exception`*`, match=regex_match)` A context manager that fails unless its context raises an exc *`exc`* such that `isinstance(`*`exc`*`, `*`expected_exception`*`)` is tru When `match` is given, the test fails unless *`exc`*'s `str` represen also matches `re.search(match, str(`*`exc`*`))`.
`skip`	`skip(`*`skip_reason`*`)` Forces skipping of the current test; use this, for example, wh test is dependent on a previous test that has already failed.

`warns`	`warns(`*`expected_warning`*`, match=`*`regex_match`*`)` Similar to *`raises`*; used to wrap code that tests that an expe warning is raised.

The `pytest.mark` subpackage includes decorators to “mark” test methods with additional test behavior, including the ones listed in **Table 17-4**.

Table 17-4. Decorators in the `pytest.mark` subpackage

`parametrize`	`@parametrize(`*`args_string`*`, `*`arg_test_values`*`)` Calls the decorated test method, setting the arguments named in the comma-separated list *`args_string`* to the values from each argument tuple in *`arg_test_values`*. The following code runs `test_is_greater` twice, once with `x=1`, `y=0`, and `expected=`**`True`**; and once with `x=0`, `y=1`, and `expected=`**`False`**.
	`@pytest.mark.parametrize`
	`("x,y,expected",`
	` [(1, 0, True), (0, 1, False)])`
	`def test_is_greater(x, y, expected):`
	`assert (x > y) == expected`

`skip`, `skipif`	`@skip(`*`skip_reason`*`)`, `@skipif(`*`condition`*`, `*`skip_reason`*`)` Skip a test method, optionally based on some global condition.

Debugging

Since Python’s development cycle is fast, the most effective way to debug is often to edit your code to output relevant information at key points.

Python has many ways to let your code explore its own state to extract information that may be relevant for debugging. The `inspect` and `traceback` modules specifically support such exploration, which is also known as *reflection* or *introspection*.

Once you have debugging-relevant information, `print` is often the natural way to display it (`pprint`, covered in **"The pprint Module"**, is also often a good choice). However, it's frequently even better to *log* debugging information to files. Logging is useful for programs that run unattended (e.g., server programs). Displaying debugging information is just like displaying other information, as covered in **Chapter 11**. Logging such information is like writing to files (covered in the same chapter); however, Python's standard library supplies a `logging` module, covered in **"The logging module"**, to help with this frequent task. As covered in **Table 8-3**, rebinding `excepthook` in the module `sys` lets your program log error info just before terminating with a propagating exception.

Python also offers hooks to enable interactive debugging. The `pdb` module supplies a simple text-mode interactive debugger. Other powerful interactive debuggers for Python are part of IDEs such as IDLE and various commercial offerings, as mentioned in **"Python Development Environments"**; we do not cover these advanced debuggers further in this book.

Before You Debug

Before you embark on lengthy debugging explorations, make sure you have thoroughly checked your Python sources with the tools mentioned in **Chapter 2**. Such tools catch only a subset of the bugs in your code, but they're much faster than interactive debugging: their use amply pays for itself.

Moreover, again before starting a debugging session, make sure that all the code involved is well covered by unit tests, as described in **"The unittest Module"**. As mentioned earlier in the chapter, once you have found a bug, *before* you fix it, add to your suite of unit tests (or, if need be, to the suite of system tests) a test or two that would have found the bug had they been present from the start, and run the tests again to confirm that they now reveal and isolate the bug; only once that is done should you proceed to fix the bug. Regularly following this procedure will help you learn to write better, more thorough tests, ensuring that you end up with a more robust test suite and have greater confidence in the overall, *enduring* correctness of your code.

Remember, even with all the facilities offered by Python, its standard library, and whatever IDE you fancy, debugging is still *hard*. Take this into account even before you start designing and coding: write and run plenty of unit tests, and keep your design and code *simple*, to reduce to the minimum the amount of debugging you will need! Brian Kernighan offers this classic advice: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as you can, you are, by definition, not smart enough to debug it.” This is part of why “clever” is not a positive word when used to describe Python code, or a coder.

The inspect Module

The `inspect` module supplies functions to get information about all kinds of objects, including the Python call stack (which records all function calls currently executing) and source files. The most frequently used functions of `inspect` are listed in **Table 17-5**.

Table 17-5. Useful functions of the `inspect` module

`currentframe`	`currentframe()`
	Returns the frame object for the current function (t
	caller of `currentframe`).
	`formatargvalues(*getargvalues(currentframe()`
	for example, returns a string representing the arguments of the calling function.

`getargspec`, `formatargspec`	`getargspec(`*`f`*`)` **-3.11** Deprecated in Python 3.5, removed in Pytho 3.11. The forward-compatible way to introspect callables is to call `inspect.signature(`*`f`*`)` and use t

resulting instance of class `inspect.Signature`, covered in the following subsection.

`getargvalues`, `formatargvalues`

`getargvalues(`*`f`*`)`

f is a frame object—for example, the result of a call to the function `_getframe` in the `sys` module (cover in **“The Frame Type”**) or the function `currentfram` in the `inspect` module. `getargvalues` returns a named tuple with four items: `(`*`args`*`, `*`varargs`*`, `*`keywords`*`, `*`locals`*`)`. *`args`* is the sequence of names *f*’s function’s parameters. *`varargs`* is the name of th special parameter of form **a*, or **`None`** when *f*’s function has no such parameter. *`keywords`* is the name of the special parameter of form ***k*, or **`None`** when *f*’s function has no such parameter. *`locals`* is the dictionary of local variables for *f*. Since arguments, in particular, are local variables, the value of each argument can be obtained from *`loca`* by indexing the *`locals`* dictionary with the argument’s corresponding parameter name. `formatargvalues` accepts one to four arguments tha are the same as the items of the named tuple that `getargvalues` returns, and returns a string with thi information. `formatargvalues(*getargvalues(`*`f`*`))` returns a string with *f*’s arguments in parentheses, named form, as used in the call statement that created *f*. For example:

```
def f(x=23):
    return inspect.currentframe()
print(inspect.formatargvalues(
      *inspect.getargvalues(f())))
# prints: (x=23)
```

`getdoc`	`getdoc(`*`obj`*`)` Returns the docstring for *`obj`*, a multiline string wit tabs expanded to spaces and redundant whitespace stripped from each line.
`getfile`, `getsourcefile`	`getfile(`*`obj`*`)`, `getsourcefile(`*`obj`*`)` `getfile` returns the name of the binary or source fi that defined *`obj`*. Raises `TypeError` when unable to determine the file (for example, when *`obj`* is a built in). `getsourcefile` returns the name of the source file that defined `obj`; it raises `TypeError` when all it can find is a binary file, not the corresponding sour file.

`getmembers`	`getmembers(`*`obj`*`, filter=`**`None`**`)`
	Returns all attributes (members), both data and methods (including special methods) of *`obj`*, as a sorted list of `(`*`name`*`, `*`value`*`)` pairs. When `filter` is r **`None`**, returns only attributes for which callable `filter` returns a truthy result when called on the attribute's *`value`*, equivalent to:
	
	`((n, v) for n, v in getmembers(obj)`
	`        if filter(v))`

`getmodule`	`getmodule(`*`obj`*`)` Returns the module that defined *`obj`*, or **`None`** when unable to determine it.
`getmro`	`getmro(`*`c`*`)` Returns a tuple of bases and ancestors of class *`c`* in method resolution order (discussed in **"Inheritance"**). *`c`* is the first item in the tuple, and

each class appears only once in the tuple. For example:

```python
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
for c in inspect.getmro(D):
    print(c.__name__, end=' ')
# prints: D B C A object
```

`getsource`, `getsourcelines`	`getsource(`*`obj`*`)`, `getsourcelines(`*`obj`*`)` getsource returns a multiline string that is the source code for *obj*, and raises `IOError` if it is unab to determine or fetch it. `getsourcelines` returns a pair: the first item is the source code for *obj* (a list lines), and the second item is the line number of th first line within its file.
`isbuiltin`, `isclass`, `iscode`, `isframe`, `isfunction`, `ismethod`, `ismodule`, `isroutine`	`isbuiltin(`*`obj`*`)`, etc. Each of these functions accepts a single argument and returns **`True`** when *obj* is of the kind indicated the function name. Accepted objects are, respectively: built-in (C-coded) functions, class objects, code objects, frame objects, Python-coded functions (including **`lambda`** expressions), methods, modules, and—for `isroutine`—all methods or functions, either C-coded or Python-coded. These functions are often used as the `filter` argument to `getmembers`.
`stack`	`stack(context=1)` Returns a list of six-item tuples. The first tuple is about `stack`'s caller, the second about the caller's

caller, and so on. The items in each tuple are: frame
object, filename, line number, function name, list o
`context` source lines around the current line, index
of current line within the list.

Introspecting callables

To introspect a callable's signature, call `inspect.signature(`*f*`)`, which returns an instance *s* of class `inspect.Signature`.

s`.parameters` is a `dict` mapping parameter names to `inspect.Parameter` instances. Call *s*`.bind(*`*a*`, **`*k*`)` to bind all parameters to the given positional and named arguments, or *s*`.bind_partial(*`*a*`, **`*k*`)` to bind a subset of them: each returns an instance *b* of `inspect.BoundArguments`.

For detailed information and examples of how to introspect callables' signatures through these classes and their methods, see **PEP 362**.

An example of using inspect

Suppose that somewhere in your program you execute a statement such as:

```
x.f()
```

and unexpectedly receive an `AttributeError` informing you that object `x` has no attribute named `f`. This means that object `x` is not as you expected, so you want to determine more about `x` as a preliminary to ascertaining why `x` is that way and what you should do about it. A simple first approach might be:

```
print(type(x), x)
# or, from v3.8, use an f-string with a trailing '=' to show repr(x)
# print(f'{x=}')
x.f()
```

This will often provide sufficient information to go on; or you might change it to `print(type(x), dir(x), x)` to see what x's methods and attributes are. But if this isn't sufficient, change the statement to:

```
try:
    x.f()
except AttributeError:
    import sys, inspect
    print(f'x is type {type(x).__name__}, ({x!r})', file=sys.stderr)
    print("x's methods are:", file=sys.stderr, end='')
    for n, v in inspect.getmembers(x, callable):
        print(n, file=sys.stderr, end=' ')
    print(file=sys.stderr)
    raise
```

This example properly uses `sys.stderr` (covered in **Table 8-3**), since it displays information related to an error, not program results. The function `getmembers` of the module `inspect` obtains the names of all the methods available on x in order to display them. If you often need this kind of diagnostic functionality, you can package it up into a separate function, such as:

```
import sys, inspect
def show_obj_methods(obj, name, show=sys.stderr.write):
    show(f'{name} is type {type(obj).__name__}({obj!r})\n')
    show(f"{name}'s methods are: ")
    for n, v in inspect.getmembers(obj, callable):
       show(f'{n} ')
    show('\n')
```

And then the example becomes just:

```
try:
    x.f()
except AttributeError:
    show_obj_methods(x, 'x')
    raise
```

Good program structure and organization are just as necessary in code intended for diagnostic and debugging purposes as they are in code that implements your program's functionality. See also **"The assert Statement"** for a good technique to use when defining diagnostic and debugging functions.

The traceback Module

The `traceback` module lets you extract, format, and output information about tracebacks that uncaught exceptions normally produce. By default, this module reproduces the formatting Python uses for tracebacks. However, the `traceback` module also lets you exert fine-grained control. The module supplies many functions, but in typical use you need only one of them:

`print_exc`	`print_exc(limit=None, file=sys.stderr)` Call `print_exc` from an exception handler, or from a function called, directly or indirectly, by an exception handler. `print_exc` outputs to file-like object `file` the traceback that Python outputs to `stderr` for uncaught exceptions. When `limit` is an integer, `print_exc` outputs only `limit` traceback nesting levels. For example, when, in an exception handler, you want to cause a diagnostic message just as if the exception propagated, but stop the exception from propagating further (so that your program keeps running and no further handlers are involved), call `traceback.print_exc()`.

The pdb Module

The `pdb` module uses the Python interpreter's debugging and tracing hooks to implement a simple command-line interactive debugger. `pdb` lets you set breakpoints, single-step and jump to source code, examine stack frames, and so on.

To run code under `pdb`'s control, import `pdb`, then call `pdb.run`, passing as the single argument a string of code to execute. To use `pdb` for post-mortem debugging (debugging of code that just terminated by propagating an exception at an interactive prompt), call `pdb.pm()` without arguments. To trigger `pdb` directly from your application code, use the built-in function `breakpoint`.

When `pdb` starts, it first reads text files named *.pdbrc* in your home directory and in the current directory. Such files can contain any `pdb` commands, but most often you put `alias` commands in them to define useful synonyms and abbreviations for other commands that you use often.

When `pdb` is in control, it prompts with the string `(Pdb)`, and you can enter `pdb` commands. The command `help` (which you can enter in the abbreviated form `h`) lists the available commands. Call `help` with an argument (separated by a space) to get help about any specific command. You can abbreviate most commands to the first one or two letters, but you must always enter commands in lowercase: `pdb`, like Python itself, is case-sensitive. Entering an empty line repeats the previous command. The most frequently used `pdb` commands are listed in **Table 17-6**.

Table 17-6. *Commonly used pdb commands*

`!`	`!` *`statement`* Executes Python statement *`statement`* with the currently selected stack frame (see the `d` and `u` commands later in th table) as the local namespace.
`alias,` `unalias`	`alias [`*`name`*` [`*`command`*`]],` `unalias `*`name`*

	Defines a short form of a frequently used command. *`command`* is any `pdb` command, with arguments, and may contain `%1`, `%2`, and so on to refer to specific arguments passed to the new alias *`name`* being defined, or `%*` to refer t all such arguments. `alias` with no arguments lists current defined aliases. `alias` *`name`* outputs the current definition alias *`name`*. `unalias` *`name`* removes an alias.
`args`, `a`	`args` Lists all arguments passed to the function you are currentl debugging.
`break`, `b`	`break [`*`location`*`[, `*`condition`*`]]` With no arguments, lists the currently defined breakpoint and the number of times each breakpoint has triggered. With an argument, `break` sets a breakpoint at the given *`location`*. *`location`* can be a line number or a function name, optionally preceded by *`filename`*`:` to set a breakpoi in a file that is not the current one or at the start of a function whose name is ambiguous (i.e., a function that exists in more than one file). When *`condition`* is present, i is an expression to evaluate (in the debugged context) each time the given line or function is about to execute; executi breaks only when the expression returns a truthy value. When setting a new breakpoint, `break` returns a breakpoi number, which you can later use to refer to the new breakpoint in any other breakpoint-related `pdb` command.
`clear`, `cl`	`clear [`*`breakpoint-numbers`*`]` Clears (removes) one or more breakpoints. `clear` with no arguments removes all breakpoints after asking for confirmation. To deactivate a breakpoint temporarily, without removing it, see `disable`, covered below.
`condition`	`condition` *`breakpoint-number`* `[`*`expression`*`]` `condition` *`n expression`* sets or changes the condition on

	breakpoint *n*. `condition` *`n`*, without *`expression`*, makes breakpoint *`n`* unconditional.
`continue`, `c`, `cont`	`continue` Continues execution of the code being debugged, up to a breakpoint, if any.
`disable`	`disable [`*`breakpoint-numbers`*`]` Disables one or more breakpoints. `disable` without arguments disables all breakpoints (after asking for confirmation). This differs from `clear` in that the debugge remembers the breakpoint, and you can reactivate it via `enable`.
`down`, `d`	`down` Moves down one frame in the stack (i.e., toward the most recent function call). Normally, the current position in the stack is at the bottom (at the function that was called most recently and is now being debugged), so `down` can't go further down. However, `down` is useful if you have previously executed the command `up`, which moves the current position upward in the stack.
`enable`	`enable [`*`breakpoint-numbers`*`]` Enables one or more breakpoints. `enable` without arguments enables all breakpoints after asking for confirmation.
`ignore`	`ignore` *`breakpoint-number`* `[`*`count`*`]` Sets the breakpoint's ignore count (to `0` if *`count`* is omitted) Triggering a breakpoint whose ignore count is greater tha `0` just decrements the count. Execution stops, presenting y with an interactive `pdb` prompt, only when you trigger a breakpoint whose ignore count is `0`. For example, say that module *fob.py* contains the following code:

```
def f():
    for i in range(1000):
        g(i)
def g(i):
    pass
```

Now consider the following interactive pdb session (minor formatting details may change depending on the Python version you're running):

```
>>> import pdb
>>> import fob
>>> pdb.run('fob.f()')
```

```
> <string>(1)?()
(Pdb) break fob.g
Breakpoint 1 at C:\mydir\fob.py:5
(Pdb) ignore 1 500
Will ignore next 500 crossings of breakpoint 1.
(Pdb) continue
> C:\mydir\fob.py(5)
g()-> pass
(Pdb) print(i)
500
```

The `ignore` command, as `pdb` says, tells `pdb` to ignore the next `500` hits on breakpoint `1`, which we set at `fob.g` in the previous `break` statement. Therefore, when execution finally stops, the function g has already been called `500` times, as we show by printing its argument `i`, which indee is now `500`. The ignore count of breakpoint `1` is now `0`; if w execute another `continue` and `print i`, `i` shows as `501`. In other words, once the ignore count decrements to `0`, execution stops every time the breakpoint is hit. If we war to skip some more hits, we must give `pdb` another `ignore`

	command, setting the ignore count of breakpoint `1` to som value greater than `0` yet again.
`jump`, `j`	`jump` *`line_number`* Sets the next line to execute to the given line number. You can use this to skip over some code by advancing to a line beyond it, or revisit some code that was already run by jumping to a previous line. (Note that a `jump` to a previous source line is not an undo command: any changes to program state made after that line are retained.) `jump` does come with some limitations—for example, you can only jump within the bottom frame, and you cannot jump into a loop or out of a **`finally`** block—but it can still an extremely useful command.
`list`, `l`	`list [`*`first`*` [, `*`last`*`] ]` Without arguments, lists `11` (eleven) lines centered on the current one, or the next `11` lines if the previous command was also a `list`. Arguments to the `list` command can optionally specify the first and last lines to list within the current file; use a dot (`.`) to indicate the current debug line The `list` command lists physical lines, counting and including comments and empty lines, not logical lines. `list`’s output marks the current line with `->`; if the curren line was reached in the course of handling an exception, tl line that raised the exception is marked with `>>`.
`ll`	`ll` Long version of `list`, showing all lines in the current function or frame.
`next`, `n`	`next` Executes the current line, without “stepping into” any function called from the current line. However, hitting breakpoints in functions called directly or indirectly from the current line does stop execution.

`print`, `p`	`print(`*`expression`*`)`, `p` *`expression`* Evaluates *`expression`* in the current context and displays the result.
`quit`, `q`	`quit` Immediately terminates both `pdb` and the program being debugged.
`return`, `r`	`return` Executes the rest of the current function, stopping only at breakpoints, if any.
`step`, `s`	`step` Executes the current line, stepping into any function calle from the current line.
`tbreak`	`tbreak [`*`location`*`[, `*`condition`*`]]` Like `break`, but the breakpoint is temporary (i.e., `pdb` automatically removes the breakpoint as soon as the breakpoint is triggered).
`up`, `u`	`up` Moves up one frame in the stack (i.e., away from the most recent function call and toward the calling function).
`where`, `w`	`where` Shows the stack of frames and indicates the current one (i. in which frame's context the command `!` executes statements, the command `args` shows arguments, the command `print` evaluates expressions, etc.).

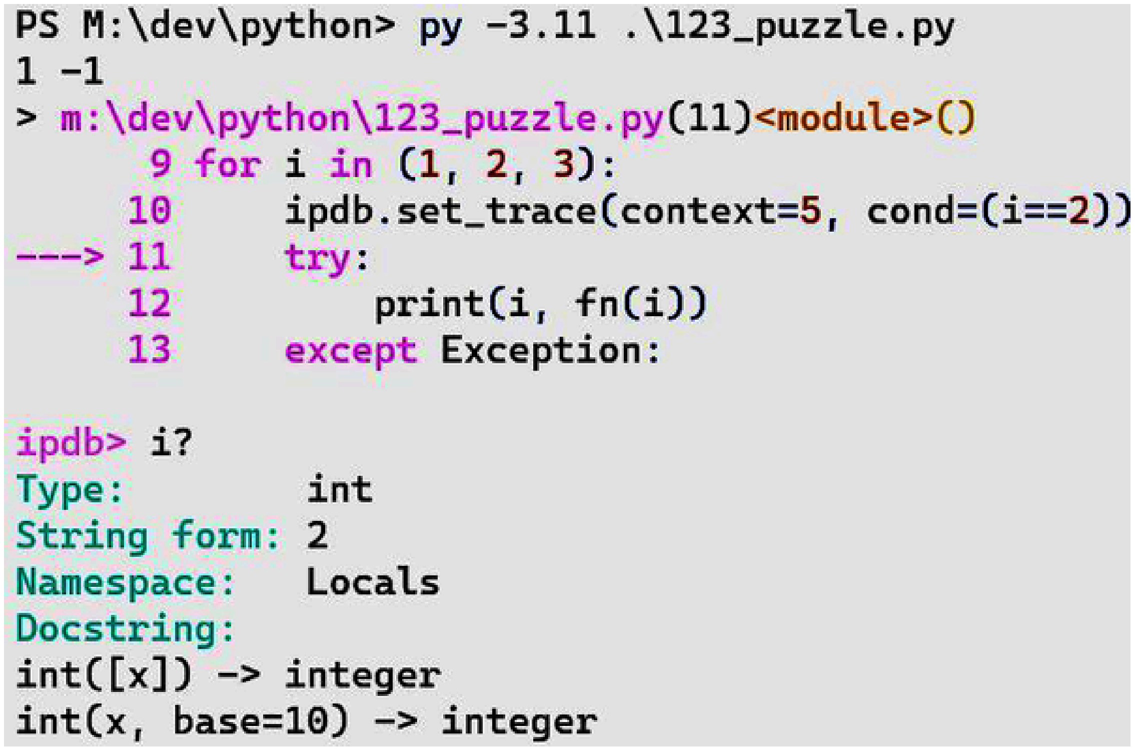
You can also enter a Python expression at the `(Pdb)` prompt, and `pdb` will evaluate it and display the result, just as if you were at the Python interpreter prompt. However, when you enter an expression whose first term coincides with a pdb command, the pdb command will execute. This is especially problematic when debugging code with single-letter variables like *p* and *q*. In these cases, you must begin the expression with `!` or precede it with the `print` or `p` command.

Other Debugging Modules

While `pdb` is built into Python, there are third-party packages that provide enhanced features for debugging.

ipdb

Just as `ipython` extends the interactive interpreter provided by Python, ipdb adds the same inspection, tab completion, command-line editing, and history features (and magic commands) to `pdb`. Figure 17-1 shows an example interaction.



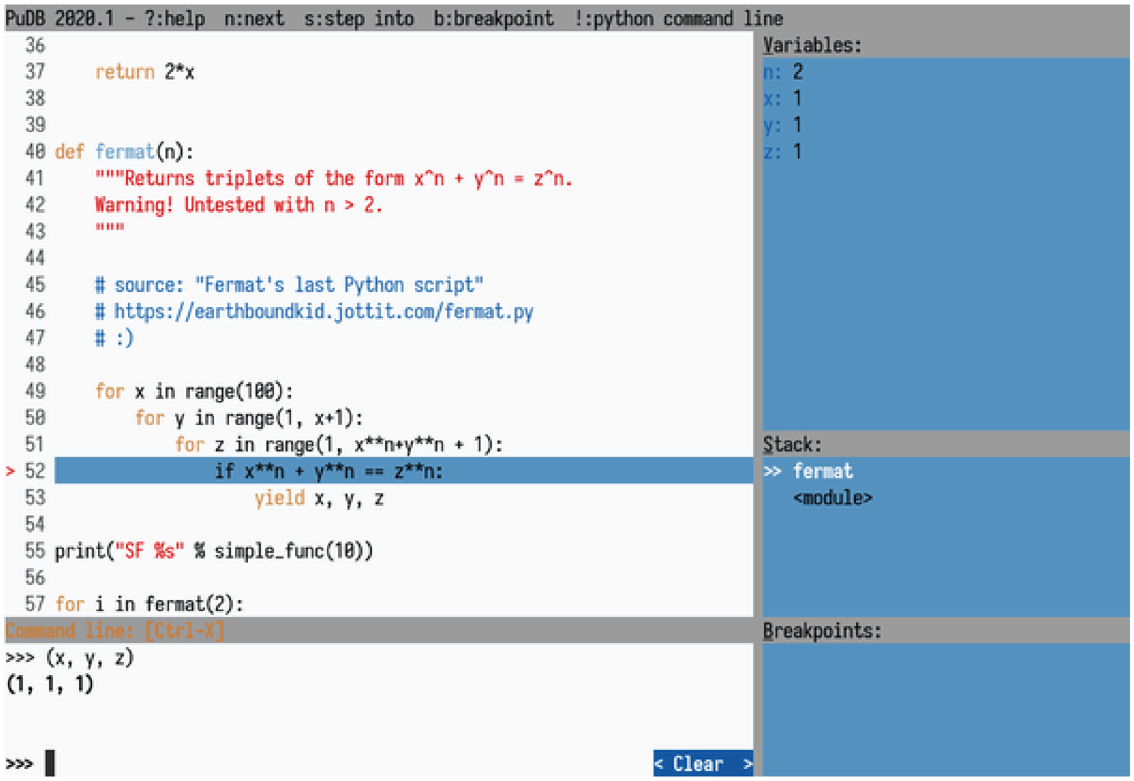
```
PS M:\dev\python> py -3.11 .\123_puzzle.py
1 -1
> m:\dev\python\123_puzzle.py(11)<module>()
      9 for i in (1, 2, 3):
     10     ipdb.set_trace(context=5, cond=(i==2))
---> 11     try:
     12         print(i, fn(i))
     13     except Exception:

ipdb> i?
Type:           int
String form: 2
Namespace:    Locals
Docstring:
int([x]) -> integer
int(x, base=10) -> integer
```

Figure 17-1. *Example of an ipdb session*

`ipdb` also adds configuration and conditional expressions to its version of `set_trace`, giving more control over when your program is to break out into the debugging session. (In this example, the breakpoint is conditional on `i` being equal to 2.)

pudb

pudb is a lightweight “graphical-like” debugger that runs in a terminal console (see Figure 17-2), utilizing the urwid console UI library. It is especially useful when connecting to remote Python environments using terminal sessions such as `ssh`, where a windowed-GUI debugger is not easy to install or run.



```
PuDB 2020.1 - ?:help  n:next  s:step into  b:breakpoint  !:python command line

    return 2*x


def fermat(n):
    """Returns triplets of the form x^n + y^n = z^n.
    Warning! Untested with n > 2.
    """

    # source: "Fermat's last Python script"
    # https://earthboundkid.jottit.com/fermat.py
    # :)

    for x in range(100):
        for y in range(1, x+1):
            for z in range(1, x**n+y**n + 1):
>               if x**n + y**n == z**n:
                    yield x, y, z

print("SF %s" % simple_func(10))

for i in fermat(2):
```

```
Variables:
n: 2
x: 1
y: 1
z: 1

Stack:
>> fermat
   <module>

Breakpoints:
```

```
Command line: [Ctrl-X]
>>> (x, y, z)
(1, 1, 1)

>>>
```

Figure 17-2. *Example of a pudb session*

`pudb` has its own set of debugging commands and interface, which take some practice to use; however, it makes a visual debugging environment handily available when working in tight computing spaces.

The warnings Module

Warnings are messages about errors or anomalies that aren’t serious enough to disrupt the program’s control flow (as would happen by raising an exception). The `warnings` module affords fine-grained control over which warnings are output and what happens to them. You can conditionally output a warning by calling the function `warn` in the `warnings` module. Other functions in the module let you control how warnings are formatted, set their destinations, and conditionally suppress some warnings or transform some warnings into exceptions.

Classes

Exception classes that represent warnings are not supplied by `warnings`: rather, they are built-ins. The class `Warning` subclasses `Exception` and is the base class for all warnings. You may define your own warning classes, which must subclass `Warning`, either directly or via one of its other existing subclasses—these include:

`DeprecationWarning`
: For use of deprecated features which are still supplied only for backward compatibility

`RuntimeWarning`
: For use of features whose semantics are error prone

`SyntaxWarning`
: For use of features whose syntax is error prone

`UserWarning`
: For other user-defined warnings that don't fit any of the above cases

Objects

Python supplies no concrete "warning objects." Rather, a warning is made up of a *`message`* (a string), a *`category`* (a subclass of `Warning`), and two pieces of information to identify where the warning was raised: *`module`* (the name of the module that raised the warning) and *`lineno`* (the number of the line in the source code that raised the warning). Conceptually, you may think of these as attributes of a warning object *`w`*: we use attribute notation later, strictly for clarity, but no specific object *`w`* actually exists.

Filters

At any time, the `warnings` module keeps a list of active filters for warnings. When you import `warnings` for the first time in a run, the module examines `sys.warnoptions` to determine the initial set of filters. You can run Python with the option `-W` to set `sys.warnoptions` for a given run. Do not rely on the initial set of filters being held specifically in `sys.warnoptions`, as this is an implementation detail that may change in future versions of Python.

As each warning *`w`* occurs, `warnings` tests *`w`* against each filter until a filter matches. The first matching filter determines what happens to *`w`*. Each filter is a tuple of five items. The first item, *`action`*, is a string that defines what happens on a match. The other four items, *`message`*, *`category`*, *`module`*, and *`lineno`*, control what it means for *`w`* to match the filter: for a match, all conditions must be satisfied. Here are the meanings of these items (using attribute notation to indicate conceptual attributes of *`w`*):

`message`

A regular expression pattern string; the match condition is `re.match(`*`message`*, *`w`*`.message, re.I)` (the match is case insensitive)

`category`

`Warning` or a subclass; the match condition is `issubclass(`*`w`*`.category,` *`category`*`)`

`module`

A regular expression pattern string; the match condition is `re.match(`*`module`*`,` *`w`*`.module)` (the match is case sensitive)

`lineno`

An `int`; the match condition is *`lineno`* **`in`** `(`*`0`*, *`w`*`.lineno)`: that is, either *`lineno`* is `0`, meaning *`w`*`.lineno` does not matter, or *`w`*`.lineno` must exactly equal *`lineno`*

Upon a match, the first field of the filter, the *`action`*, determines what happens. It can have the following values:

`'always'`

`w``.message` is output whether or not *`w`* has already occurred.

`'default'`

`w``.message` is output if, and only if, this is the first time *`w`* has occurred from this specific location (i.e., this specific `(`*`w`*`.module,` *`w`*`.location)` pair).

`'error'`

`w``.category(`*`w`*`.message)` is raised as an exception.

`'ignore'`

`w` is ignored.

`'module'`

`w.message` is output if, and only if, this is the first time *w* occurs from `w.module`.

`'once'`

`w.message` is output if, and only if, this is the first time *w* occurs from any location.

When a module issues a warning, `warnings` adds to that module's global variables a `dict` named `__warningsgregistry__`, if that `dict` is not already present. Each key in the `dict` is a pair `(message, category)`, or a tuple with three items `(message, category, lineno)`; the corresponding value is **`True`** when further occurrences of that message are to be suppressed. Thus, for example, you can reset the suppression state of all warnings from a module *m* by executing `m.__warningsregistry__.clear()`: when you do that, all messages are allowed to get output again (once), even when, for example, they've previously triggered a filter with an *action* of `'module'`.

Functions

The `warnings` module supplies the functions listed in **Table 17-7**.

Table 17-7. *Functions of the `warnings` module*

`filter warnings`	`filterwarnings(action, message='.*', category=Warning, module='.*', lineno=0, append=False)` Adds a filter to the list of active filters. When `append` is **`True`**, `filterwarnings` adds the filter after all other exist filters (i.e., appends the filter to the list of existing filters otherwise, `filterwarnings` inserts the filter before any other existing filter. All components, save *action*, have default values that mean "match everything." As detaile above, `message` and `module` are pattern strings for regul expressions, `category` is some subclass of `Warning`, `line` is an integer, and *action* is a string that determines wha happens when a message matches this filter.

`formatwarning`	`formatwarning(`*`message`*`, `*`category`*`, `*`filename`*`, `*`lineno`*`)` Returns a string that represents the given warning with standard formatting.
`resetwarnings`	`resetwarnings()` Removes all filters from the list of filters. `resetwarnings` also discards any filters originally added with the **`-W`** command-line option.
`showwarning`	`showwarning(`*`message`*`, `*`category`*`, `*`filename`*`, `*`lineno`*`, file=sys.stderr)` Outputs the given warning to the given file object. Filter actions that output warnings call `showwarning`, letting th argument `file` default to `sys.stderr`. To change what happens when filter actions output warnings, code your own function with this signature and bind it to `warnings.showwarning`, thus overriding the default implementation.
`warn`	`warn(`*`message`*`, category=UserWarning, stacklevel=1` Sends a warning so that the filters examine and possibly output it. The location of the warning is the current function (caller of `warn`) if `stacklevel` is `1`, or the caller the current function if `stacklevel` is `2`. Thus, passing `2` a the value of `stacklevel` lets you write functions that sen warnings on their caller's behalf, such as:

```
def to_unicode(bytestr):
    try:
        return bytestr.decode()
    except UnicodeError:
        warnings.warn(f'Invalid characters in
                      {bytestr!r}',
                      stacklevel=2)
        return bytestr.decode(errors='ignore'
```

Thanks to the parameter `stacklevel=2`, the warning appears to come from the caller of `to_unicode`, rather th from `to_unicode` itself. This is very important when the *`action`* of the filter that matches this warning is `'defaul` or `'module'`, since these actions output a warning only t first time the warning occurs from a given location or module.

Optimization

"First make it work. Then make it right. Then make it fast." This quotation, often with slight variations, is widely known as "the golden rule of programming." As far as we've been able to ascertain, the source is Kent Beck, who credits his father with it. This principle is often quoted, but too rarely followed. A negative form, slightly exaggerated for emphasis, is a quotation by Don Knuth (who credits Sir Tony Hoare with it): "Premature optimization is the root of all evil in programming."

Optimization is premature if your code is not working yet, or if you're not sure what, precisely, your code should be doing (since then you cannot be sure if it's working). First make it work: ensure that your code is correctly performing exactly the tasks it is *meant* to perform.

Optimization is also premature if your code is working but you are not satisfied with the overall architecture and design. Remedy structural flaws before worrying about optimization: first make it work, then make it right. These steps are not optional; working, well-architected code is *always* a must.[6]

Having a good test suite is key before attempting any optimization. After all, the purpose of optimization is to increase speed or reduce memory consumption—or both—without changing the code's behavior.

In contrast, you don't always need to make it fast. Benchmarks may show that your code's performance is already acceptable after the first two steps. When performance is not acceptable, profiling often shows that all performance issues are in a small part of the code, with your program spending perhaps 80 or 90% of its time in 10 to 20% of the code.[7] Such performance-crucial regions of your code are known as *bottlenecks*, or *hot spots*. It's a waste of effort to optimize large portions of code that account for, say, 10 percent of your program's running time. Even if you made that part run 10 times as fast (a rare feat), your program's overall runtime would only decrease by 9%,[8] a speedup no user would likely even notice. If optimization is needed, focus your efforts where they matter: on bottlenecks. You can often optimize bottlenecks while keeping your code 100% pure Python, thus not preventing future porting to other Python implementations.

Developing a Fast-Enough Python Application

Start by designing, coding, and testing your application in Python, using available extension modules if they save you work. This takes much less time than it would with a classic compiled language. Then benchmark the application to find out if the resulting code is fast enough. Often it is, and you're done—congratulations! Ship it!

Since much of Python itself is coded in highly optimized C (as are many of its standard library and extension modules), your application may even turn out to already be faster than typical C code. However, if the application is too slow, you need, first and foremost, to rethink your algorithms and data structures. Check for bottlenecks due to application architecture, network traffic, database access, and operating system interactions. For many applications, each of these factors is more likely than language choice, or coding details, to cause slowdowns. Tinkering with large-scale architectural aspects can often dramatically speed up an application, and Python is an excellent medium for such experimentation. If you're using a version control system (and you ought to be!), it should be easy to create experimental branches or clones where you can try out different techniques to see which—if any—deliver significant improvements, all without jeopardizing your working code. You can then merge back any improvements that pass your tests.

If your program is still too slow, profile it: find out where the time is going! As we previously mentioned, applications often exhibit computational bottlenecks, with small areas of the source code accounting for the vast majority of the running time. Optimize the bottlenecks, applying the techniques suggested in the rest of this chapter.

If normal Python-level optimizations still leave some outstanding computational bottlenecks, you can recode those as Python extension modules, as covered in **Chapter 25**. In the end, your application will run at roughly the same speed as if you had coded it all in C, C++, or FORTRAN—or faster, when large-scale experimentation has let you find a better architecture. Your overall programming productivity with this process will not be much lower than if you had coded everything in Python. Future changes and maintenance are easy, since you use Python to express the overall structure of the program, and lower-level, harder-to-maintain languages for only a few specific computational bottlenecks.

As you build applications in a given area following this process, you will accumulate a library of reusable Python extension modules. You will therefore become more and more productive at developing other fast-running Python applications in the same field.

Even if external constraints eventually force you to recode your whole application in a lower-level language, you'll still be better off for having started in Python. Rapid prototyping has long been acknowledged as the best way to get software architecture right. A working prototype lets you check that you have identified the right problems and taken a good path to their solution. A prototype also affords the kind of large-scale architectural experimentation that can make a real difference in performance. You can migrate your code gradually to other languages by way of extension modules, if need be, and the application remains fully functional and testable at each stage. This ensures against the risk of compromising a design's architectural integrity in the coding stage.

Even if you are required to use a low-level language for the entire application, it can often be more productive to write it in Python first (especially if you are new to the application's domain). Once you have a working Python version, you can experiment with the user or network interface or library API, and with the architecture. Also, it is much easier to find and fix bugs and to make changes in Python code than in lower-level languages. At the end, you'll know the code so well that porting to a lower-level language should be very fast and straightforward, safe in the knowledge that most of the design mistakes were made and fixed in the Python implementation.

The resulting software will be faster and more robust than if all of the coding had been lower-level from the start, and your productivity—while not quite as good as with a pure Python application—will still be higher than if you had been coding at a lower level throughout.

Benchmarking

Benchmarking (also known as *load testing*) is similar to system testing: both activities are much like running the program for production purposes. In both cases, you need to have at least some subset of the program's intended functionality working, and you need to use known, reproducible inputs. For benchmarking, you don't need to capture and check your program's output: since you make it work and make it right before you make it fast, you're already fully confident about your program's correctness by the time you load test it. You do need inputs that are representative of typical system operation—ideally ones that are likely to pose the greatest challenges to your program's performance. If your program performs several kinds of operations, make sure you run some benchmarks for each different kind of operation.

Elapsed time as measured by your wristwatch is probably precise enough to benchmark most programs. A 5 or 10% difference in performance, except in programs with very peculiar constraints, makes no practical difference to a program's real-life usability. (Programs with hard real-time constraints are another matter, since they have needs very different from those of normal programs in most respects.)

When you benchmark "toy" programs or snippets in order to help you choose an algorithm or data structure, you may need more precision: the `timeit` module of Python's standard library (covered in **"The timeit module"**) is quite suitable for such tasks. The benchmarking discussed in this section is of a different kind: it is an approximation of real-life program operation for the sole purpose of checking whether the program's performance on each task is acceptable, before embarking on profiling and other optimization activities. For such "system" benchmarking, a situation that approximates the program's normal operating conditions is best, and high accuracy in timing is not all that important.

Large-Scale Optimization

The aspects of your program that are most important for performance are large-scale ones: your choice of overall architecture, algorithms, and data structures.

The performance issues that you must often take into account are those connected with the traditional big-O notation of computer science. Informally, if you call *`N`* the input size of an algorithm, big-O notation expresses algorithm performance, for large values of *`N`*, as proportional to some function of *`N`*. (In precise computer science lingo, this should be called big-Theta notation, but in real life, programmers always call it big-O, perhaps because an uppercase Theta looks a bit like an O with a dot in the center!)

An `O(1)` algorithm (also known as "constant time") is one that takes a certain amount of time not growing with *`N`*. An `O(N)` algorithm (also known as "linear time") is one where, for large enough *`N`*, handling twice as much data takes about twice as much time, three times as much data takes three times as much time, and so on, proportionally to *`N`*. An `O(N²)` algorithm (also known as a "quadratic time" algorithm) is one where, for large enough *`N`*, handling twice as much data takes about four times as much time, three times as much data takes nine times as much time, and so on, growing proportionally to *`N`* squared. Identical concepts and notation are used to describe a program's consumption of memory ("space") rather than of time.

To find more information on big-O notation, and about algorithms and their complexity, any good book about algorithms and data structures can help; we recommend Magnus Lie Hetland’s excellent book *Python Algorithms: Mastering Basic Algorithms in the Python Language*, 2nd edition (Apress).

To understand the practical importance of big-O considerations in your programs, consider two different ways to accept all items from an input iterable and accumulate them into a list in reverse order:

```python
def slow(it):
    result = []
    for item in it:
        result.insert(0, item)
    return result

def fast(it):
    result = []
    for item in it:
        result.append(item)
    result.reverse()
    return result
```

We could express each of these functions more concisely, but the key difference is best appreciated by presenting the functions in these elementary terms. The function `slow` builds the result list by inserting each input item *before* all previously received ones. The function `fast` appends each input item *after* all previously received ones, then reverses the result list at the end. Intuitively, one might think that the final reversing represents extra work, and therefore `slow` should be faster than `fast`. But that’s not the way things work out.

Each call to `result.append` takes roughly the same amount of time, independent of how many items are already in the list `result`, since there is (nearly) always a free slot for an extra item at the end of the list (in pedantic terms, `append` is *amortized* `O(1)`, but we don’t cover amortization in this book). The **`for`** loop in the function `fast` executes *N* times to receive `N` items. Since each iteration of the loop takes a constant time, overall loop time is `O(N)`. `result.reverse` also takes time `O(N)`, as it is directly proportional to the total number of items. Thus, the total running time of `fast` is `O(N)`. (If you don't understand why a sum of two quantities, each `O(N)`, is also `O(N)`, consider that the sum of any two linear functions of `N` is also a linear function of `N`—and "being `O(N)`" has exactly the same meaning as "consuming an amount of time that is a linear function of `N`.")

On the other hand, each call to `result.insert` makes space at slot `0` for the new item to insert, moving all items that are already in list `result` forward one slot. This takes time proportional to the number of items already in the list. The overall amount of time to receive `N` items is therefore proportional to `1+2+3+...N-1`, a sum whose value is `O(N²)`. Therefore, the total running time of `slow` is `O(N²)`.

It's almost always worth replacing an `O(N²)` solution with an `O(N)` one, unless you can somehow assign rigorous small limits to input size `N`. If `N` can grow without very strict bounds, the `O(N²)` solution turns out to be disastrously slower than the `O(N)` one for large values of `N`, no matter what the proportionality constants in each case may be (and no matter what profiling tells you). Unless you have other `O(N²)` or even worse bottlenecks elsewhere that you can't eliminate, a part of the program that is `O(N²)` turns into the program's bottleneck, dominating runtime for large values of `N`. Do yourself a favor and watch out for the big-O: all other performance issues, in comparison, are usually almost insignificant.

Incidentally, you can make the function `fast` even faster by expressing it in more idiomatic Python. Just replace the first two lines with the following single statement:

```
result = list(it)
```

This change does not affect `fast`'s big-O character (`fast` is still `O(N)` after the change), but does speed things up by a large constant factor.

SIMPLE IS BETTER THAN COMPLEX, AND USUALLY FASTER!

More often than not, in Python, the simplest, clearest, most direct and idiomatic way to express something is also the fastest.

Choosing algorithms with good big-O performance is roughly the same task in Python as in any other language. You just need a few hints about the big-O performance of Python's elementary building blocks, and we provide them in the following sections.

List operations

Python lists are internally implemented as *vectors* (also known as *dynamic arrays*), not as "*linked* lists." This implementation choice determines the performance characteristics of Python lists, in big-O terms.

Chaining two lists `L1` and `L2`, of length `N1` and `N2` (i.e., `L1+L2`) is `O(N1+N2)`. Multiplying a list `L` of length `N` by integer `M` (i.e., `L*M`) is `O(N*M)`. Accessing or rebinding any list item is `O(1)`. `len()` on a list is also `O(1)`. Accessing any slice of length `M` is `O(M)`. Rebinding a slice of length `M` with one of identical length is also `O(M)`. Rebinding a slice of length `M1` with one of different length `M2` is `O(M1+M2+N1)`, where `N1` is the number of items *after* the slice in the target list (so, length-changing slice rebindings are relatively cheap when they occur at the *end* of a list, but costlier when they occur at the *beginning* or around the middle of a long list). If you need first-in, first-out operations, a list is probably not the fastest data structure for the purpose: instead, try the type `collections.deque`, covered in "deque".

Most list methods, as shown in Table 3-5, are equivalent to slice rebindings and have equivalent big-O performance. The methods `count`, `index`, `remove`, and `reverse`, and the operator **`in`**, are `O(N)`. The method `sort` is generally `O(N log N)`, but `sort` is highly optimized[9] to be `O(N)` in some important special cases, such as when the list is already sorted or reverse-sorted except for a few items. `range(a, b, c)` is `O(1)`, but looping on all items of the result is `O((b - a) // c)`.

String operations

Most methods on a string of length *N* (be it bytes or Unicode) are `O(N)`. `len(astring)` is `O(1)`. The fastest way to produce a copy of a string with transliterations and/or removal of specified characters is the string's method `translate`. The single most practically important big-O consideration involving strings is covered in **"Building up a string from pieces"**.

Dictionary operations

Python `dicts` are implemented with hash tables. This implementation choice determines all the performance characteristics of Python dictionaries, in big-O terms.

Accessing, rebinding, adding, or removing a dictionary item is `O(1)`, as are the methods `get`, `setdefault`, and `popitem`, and the operator **`in`**. `d1.update(d2)` is `O(len(d2))`. `len(adict)` is `O(1)`. The methods `keys`, `items`, and `values` are `O(1)`, but looping on all items of the iterators those methods return is `O(N)`, as is looping directly on a `dict`.

When the keys in a dictionary are instances of classes that define `__hash__` and equality comparison methods, dictionary performance is of course affected by those methods. The performance indications presented in this section hold when hashing and equality comparison on keys are `O(1)`.

Set operations

Python sets, like `dicts`, are implemented with hash tables. All performance characteristics of sets are, in big-O terms, the same as for dictionaries.

Adding or removing an item in a set is `O(1)`, as is the operator **`in`**. `len(aset)` is `O(1)`. Looping on a set is `O(N)`. When the items in a set are instances of classes that define `__hash__` and equality comparison methods, set performance is of course affected by those methods. The performance hints presented in this section hold when hashing and equality comparison on items are `O(1)`.

Summary of big-O times for operations on Python built-in types

Let *`L`* be any list, *`T`* any string (`str` or `bytes`), *`D`* any `dict`, *`S`* any `set` (with, say, numbers as items, just for the purpose of ensuring `O(1)` hashing and comparison), and *`x`* any number (ditto):

`O(1)`
: `len(`*`L`*`)`, `len(T)`, `len(`*`D`*`)`, `len(S)`, *`L`*`[i]`, *`T`*`[i]`, *`D`*`[i]`, `del D[i]`, **`if`** *`x`* **`in`** `D`, **`if`** *`x`* **`in`** *`S`*, *`S`*`.add(`*`x`*`)`, *`S`*`.remove(`*`x`*`)`, appends or removals to/from the very right end of *`L`*

`O(N)`
: Loops on *`L`*, *`T`*, *`D`*, *`S`*, general appends or removals to/from *`L`* (except at the very right end), all methods on *`T`*, **`if`** *`x`* **`in`** *`L`*, **`if`** *`x`* **`in`** *`T`*, most methods on *`L`*, all shallow copies

`O(N log N)`
: *`L`*`.sort()`, mostly (but `O(`*`N`*`)` if *`L`* is already nearly sorted or reverse sorted)

Profiling

As mentioned at the start of this section, most programs have *hot spots*: relatively small regions of source code that account for most of the time elapsed during a program run. Don’t try to guess where your program’s hot spots are: a programmer’s intuition is notoriously unreliable in this field. Instead, use the Python standard library module `profile` to collect profile data over one or more runs of your program, with known inputs. Then use the module `pstats` to collate, interpret, and display that profile data.

To gain accuracy, you can calibrate the Python profiler for your machine (i.e., determine what overhead profiling incurs on that machine). The `profile` module can then subtract this overhead from the times it measures, making profile data you collect closer to reality. The standard library module `cProfile` has similar functionality to `profile`; `cProfile` is preferable, since it’s faster, which means it imposes less overhead.

There are also many third-party profiling tools worth considering, such as **pyinstrument** and **Eliot**; an **excellent article** by Itamar Turner-Trauring explains the basics and advantages of each of these tools.

The profile module

The `profile` module supplies one often-used function:

`run`	`run(`*`code`*`, filename=`**`None`**`)` *`code`* is a string that is usable with `exec`, normally a call to the main function of the program you're profiling. `filename` is the path of a file that `run` creates or rewrites with profile data. Usually, you call `run` a few times, specifying different filenames and different arguments to your program's main function, in order to exercise various program parts in proportion to your expectations about their use "in real life." Then, you use the `pstats` module to display collated results across the various runs. You may call `run` without a `filename` to get a summary report, similar to the one the `pstats` module provides, on standard output. However, this approach gives you no control over the output format, nor any way to consolidate several runs into one report. In practice, you should rarely use this feature: it's best to collect profile data into files, then use `pstats`. The `profile` module also supplies the class `Profile` (discussed briefly in the next section). By instantiating `Profile` directly, you can access advanced functionality, such as the ability to run a command in specified local and global dictionaries. We do not cover such advanced functionality of the class `profile.Profile` further in this book.

Calibration

To calibrate `profile` for your machine, use the class `Profile`, which `profile` supplies and internally uses in the function `run`. An instance *`p`* of `Profile` supplies one method you use for calibration:

`calibrate` *`p`*`.calibrate(`*`N`*`)`

Loops *`N`* times, then returns a number that is the profiling
per call on your machine. *`N`* must be large if your machine
Call *`p`*`.calibrate(10000)` a few times and check that the va
numbers it returns are close to each other, then pick the sı
one of them. If the numbers vary a lot, try again with a lar
of *`N`*.

The calibration procedure can be time-consuming. Howev
need to perform it only once, repeating it only when you n
changes that could alter your machine's characteristics, su
applying patches to your operating system, adding memor
changing your Python version. Once you know your mach
overhead, you can tell `profile` about it each time you imp
right before using `profile.run`. The simplest way to do thi
follows:

```
import profile
profile.Profile.bias = ...the overhead you measu
profile.run('main()', 'somefile')
```

The pstats module

The `pstats` module supplies a single class, `Stats`, to analyze, consolidate, and report on the profile data contained in one or more files written by the function `profile.run`. Its constructor has the signature:

`Stats`	**class** `Stats(`*`filename`*`, *`*`filenames`*`, stream=`*`sys.stdout`*`)` Instantiates `Stats` with one or more filenames of files of profile data written by the function `profile.run`, with profiling output sent to `stream`.

An instance *s* of the class `Stats` provides methods to add profile data and sort and output results. Each method returns *s*, so you can chain many calls in the same expression. *s*'s main methods are listed in **Table 17-8**.

Table 17-8. *Methods of an instance s of class Stats*

`add`	`add(`*`filename`*`)` Adds another file of profile data to the set that *s* is holding for analysis.
`print_callees`, `print_callers`	`print_callees(*`*`restrictions`*`)`, `print_callers(*`*`restrictions`*`)` Outputs the list of functions in *s*'s profile data, sorted according to the latest call to *s*`.sort_stats` and subject to given restrictions, if any. You can call each printing method with zero or more *`restrictions`*, to be applied one after the other, in order, to reduce the number of output lines. A restriction that is an `int` *n* limits the output to the first *n* lines. A restriction that is a `float` *f* between `0.0` and `1.0` limits the output to a fraction *f* of the lines. A restriction that is a string is compiled as a regular expression pattern (covered in **"Regular Expressions and the re Module"**); only lines that satisfy a `search` method call on the regular expression are output. Restrictions are cumulative. For example, *s*`.print_callees(10, 0.5)` outputs the first 5 lines (half of 10). Restrictions apply only after the summary and header lines: the summary and header lines are output unconditionally. Each function *f* in the output is accompanied by the

	list of *f*’s callers (functions that called *f*) or *f*’s callees (functions that *f* called), according to the name of the method.
`print_stats`	`print_stats(*restrictions)` Outputs statistics about *s*’s profile data, sorted according to the latest call to `s.sort_stats` and subject to given restrictions, if any, as covered in `print_callees` and `print_callers`, above. After a few summary lines (date and time on which profile data was collected, number of function calls, and sort criteria used), the output—absent restrictions—is one line per function, with six fields per line, labeled in a header line. `print_stats` outputs the following fields for each function *f*: 1. Total number of calls to *f* 2. Total time spent in *f*, exclusive of other functions that *f* called 3. Total time per call to *f* (i.e., field 2 divided by field 1) 4. Cumulative time spent in *f*, and all functions directly or indirectly called from *f* 5. Cumulative time per call to *f* (i.e., field 4 divided by field 1) 6. The name of function *f*
`sort_stats`	`sort_stats(*keys)` Gives one or more keys on which to sort future output. Each key is either a string or a member of the enum `pstats.SortKey`. The sort is descending for keys that indicate times or numbers, and alphabetical for key `'nfl'`. The most frequently used keys when calling `sort_stats` are: `SortKey.CALLS` or `'calls'`

Number of calls to the function (like field 1 in the `print_stats` output)

`SortKey.CUMULATIVE` or `'cumulative'`

Cumulative time spent in the function and all functions it called (like field 4 in the `print_stats` output)

`SortKey.NFL` or `'nfl'`

Name of the function, its module, and the line number of the function in its file (like field 6 in the `print_stats` output)

`SortKey.TIME` or `'time'`

Total time spent in the function itself, exclusive of functions it called (like field 2 in the `print_stats` output)

`strip_dirs`	`strip_dirs()` Alters *s* by stripping directory names from all module names to make future output more compact. *s* is unsorted after *`s`*`.strip_dirs`, and therefore you normally call *`s`*`.sort_stats` right after calling *`s`*`.strip_dirs`.

Small-Scale Optimization

Fine-tuning of program operations is rarely important. It may make a small but meaningful difference in some particularly hot spot, but it is hardly ever a decisive factor. And yet, fine-tuning—in the pursuit of mostly irrelevant microefficiencies—is where a programmer's instincts are likely to lead them. It is in good part because of this that most optimization is premature and best avoided. The most that can be said in favor of fine-tuning is that, if one idiom is *always* speedier than another when the difference is measurable, then it's worth your while to get into the habit of always using the speedier way.[10]

In Python, if you do what comes naturally, choosing simplicity and elegance, you typically end up with code that has good performance and is clear and maintainable. In other words, *let Python do the work*: when

Python provides a simple, direct way to perform a task, chances are that it's also the fastest way. In a few cases, an approach that may not be intuitively preferable still offers performance advantages, as discussed in the rest of this section.

The simplest optimization is to run your Python programs using **`python -O`** or **`-OO`**. **`-OO`** makes little difference to performance compared to **`-O`** but may save some memory, as it removes docstrings from the bytecode, and memory is sometimes (indirectly) a performance bottleneck. The optimizer is not powerful in current releases of Python, but it may gain you performance advantages on the order of 5-10% (and potentially larger if you make use of **`assert`** statements and **`if`** `__debug__:` guards, as suggested in "The assert Statement"). The best aspect of **`-O`** is that it costs nothing—as long as your optimization isn't premature, of course (don't bother using **`-O`** on a program you're still developing).

The timeit module

The standard library module `timeit` is handy for measuring the precise performance of specific snippets of code. You can import `timeit` to use `timeit`'s functionality in your programs, but the simplest and most normal use is from the command line:

```
$ python -m timeit -s 'setup statement(s)' 'statement(s) to be timed'
```

The "setup statement" is executed only once, to set things up; the "statements to be timed" are executed repeatedly, to accurately measure the average time they take.

For example, say you're wondering about the performance of `x=x+1` versus `x+=1`, where `x` is an `int`. At a command prompt, you can easily try:

```
$ python -m timeit -s 'x=0' 'x=x+1'
```

```
1000000 loops, best of 3: 0.0416 usec per loop
```

```
$ python -m timeit -s 'x=0' 'x+=1'
```

```
1000000 loops, best of 3: 0.0406 usec per loop
```

and find out that performance is, for all intents and purposes, the same in both cases (a tiny difference, such as the 2.5% in this case, is best regarded as “noise”).

Memoizing

Memoizing is the technique of saving values returned from a function that is called repeatedly with the same argument values. When the function is called with arguments that have not been seen before, a memoizing function computes the result, and then saves the arguments used to call it and the corresponding result in a cache. When the function is called again later with the same arguments, the function just looks up the computed value in the cache instead of rerunning the function calculation logic. In this way, the calculation is performed just once for any particular argument or arguments.

Here is an example of a function for calculating the sine of a value given in degrees:

```
import math
def sin_degrees(x):
    return math.sin(math.radians(x))
```

If we determined that `sin_degrees` was a bottleneck, and was being repeatedly called with the same values for `x` (such as the integer values from `0` to `360`, as you might use when displaying an analog clock), we could add a memoizing cache:

```
_cached_values = {}
def sin_degrees(x):
    if x not in _cached_values:
        _cached_values[x] = math.sin(math.radians(x))
    return _cached_values[x]
```

For functions that take multiple arguments, the tuple of argument values would be used for the cache key.

We defined *_cached_values* outside the function, so that it is not reset each time we call the function. To explicitly associate the cache with the function, we can utilize Python’s object model, which allows us to treat functions as objects and assign attributes to them:

```
def sin_degrees(x):
    cache = sin_degrees._cached_values
    if x not in cache:
        cache[x] = math.sin(math.radians(x))
    return cache[x]
sin_degrees._cached_values = {}
```

Caching is a classic approach to gain performance at the expense of using memory (the *time–memory trade-off*). The cache in this example is unbounded, so, as `sin_degrees` is called with many different values of `x`, the cache will continue to grow, consuming more and more program memory. Caches are often configured with an *eviction policy*, which determines when values can be removed from the cache. Removing the oldest cached value is a common eviction policy. Since Python keeps `dict` entries in insertion order, the “oldest” key will be the first one found if we iterate over the `dict`:

```
def sin_degrees(x):
    cache = sin_degrees._cached_values
    if x not in cache:
        cache[x] = math.sin(math.radians(x))
        # remove oldest cache entry if exceed maxsize limit
        if len(cache) > sin_degrees._maxsize:
            oldest_key = next(iter(cache))
```

```
                del cache[oldest_key]
        return cache[x]
    sin_degrees._cached_values = {}
    sin_degrees._maxsize = 512
```

You can see that this starts to complicate the code, with the original logic for computing the sine of a value given in degrees hidden inside all the caching logic. The Python stdlib module `functools` includes caching decorators `lru_cache`, **3.9+** `cache`, and **3.8+** `cached_property` to perform memoization cleanly. For example:

```
import functools
@functools.lru_cache(maxsize=512)
def sin_degrees(x):
    return math.sin(math.radians(x))
```

The signatures for these decorators are described in detail in **"The functools Module"**.

CACHING FLOATING-POINT VALUES CAN GIVE UNDESIRABLE BEHAVIOR

As was described in **"Floating-Point Values"**, comparing `float` values for equality can return **`False`** when the values are actually within some expected tolerance for being considered equal. With an unbounded cache, a cache containing `float` keys may grow unexpectedly large by caching multiple values that differ only in the 18th decimal place. For a bounded cache, many `float` keys that are very nearly equal may cause the unwanted eviction of other values that are significantly different.

All the cache techniques listed here use equality matching, so code for memoizing a function with one or more `float` arguments should take extra steps to cache rounded values, or use `math.isclose` for matching.

Precomputing a lookup table

In some cases, you can predict all the values that your code will use when calling a particular function. This allows you to precompute the values and save them in a lookup table. For example, in our application that is going to compute the `sin` function for the integer degree values `0` to `360`, we can perform this work just once at program startup and keep the results in a Python `dict`:

```
_sin_degrees_lookup = {x: math.sin(math.radians(x))
                       for x in range(0, 360+1)}
sin_degrees = _sin_degrees_lookup.get
```

Binding `sin_degrees` to the `_sin_degrees_lookup` `dict`'s `get` method means the rest of our program can still call `sin_degrees` as a function, but now the value retrieval occurs at the speed of a `dict` lookup, with no additional function overhead.

Building up a string from pieces

The single Python "anti-idiom" that is most likely to damage your program's performance, to the point that you should *never* use it, is to build up a large string from pieces by looping on string concatenation statements such as `big_string += piece`. Python strings are immutable, so each such concatenation means that Python must free the M bytes previously allocated for `big_string`, and allocate and fill $M + K$ bytes for the new version. Doing this repeatedly in a loop, you end up with roughly $O(N^2)$ performance, where N is the total number of characters. More often than not, getting $O(N^2)$ performance where $O(N)$ is easily available is a disaster.[11] On some platforms, things may be even bleaker due to memory fragmentation effects caused by freeing many areas of progressively larger sizes.

To achieve $O(N)$ performance, accumulate intermediate pieces in a list, rather than building up the string piece by piece. Lists, unlike strings, are mutable, so appending to a list is $O(1)$ (amortized). Change each occurrence of `big_string += piece` into `temp_list.append(piece)`. Then, when you're done accumulating, use the following code to build your desired string result in $O(N)$ time:

```
big_string = ''.join(temp_list)
```

Using a list comprehension, generator expression, or other direct means (such as a call to `map`, or use of the standard library module `itertools`) to build `temp_list` may often offer further (substantial, but not big-O) optimization over repeated calls to `temp_list.append`. Other `O(N)` ways to build up big strings, which a few Python programmers find more readable, are to concatenate the pieces to an instance of `array.array('u')` with the array's `extend` method, use a `bytearray`, or write the pieces to an instance of `io.TextIO` or `io.BytesIO`.

In the special case where you want to output the resulting string, you may gain a further small slice of performance by using `writelines` on `temp_list` (never building `big_string` in memory). When feasible (i.e., when you have the output file object open and available in the loop, and the file is buffered), it's just as effective to perform a `write` call for each `piece`, without any accumulation.

Although not nearly as crucial as `+=` on a big string in a loop, another case where removing string concatenation may give a slight performance improvement is when you're concatenating several values in an expression:

```
oneway = str(x) + ' eggs and ' + str(y) + ' slices of ' + k + ' ham'
another = '{} eggs and {} slices of {} ham'.format(x, y, k)
yetanother = f'{x} eggs and {y} slices of {k} ham'
```

Formatting strings using the `format` method or f-strings (discussed in Chapter 8) is often a good performance choice, as well as being more idiomatic and thereby clearer than concatenation approaches. On a sample run of the preceding example, the `format` approach is more than twice as fast as the (perhaps more intuitive) concatenation, and the f-string approach is more than twice as fast as `format`.

Searching and sorting

The operator **in**, the most natural tool for searching, is O(1) when the righthand-side operand is a set or dict, but O(*N*) when the righthand-side operand is a string, list, or tuple. If you must perform many such checks on a container, you're *much* better off using a set or dict, rather than a list or tuple, as the container. Python sets and dicts are highly optimized for searching and fetching items by key. Building the set or dict from other containers, however, is O(*N*), so for this crucial optimization to be worthwhile, you must be able to hold on to the set or dict over several searches, possibly altering it apace as the underlying sequence changes.

The sort method of Python lists is also a highly optimized and sophisticated tool. You can rely on sort's performance. Most functions and methods in the standard library that perform comparisons accept a key argument to determine how, exactly, to compare items. You provide a key function, which computes a key value for each element in the list. The list elements are sorted by their key values. For instance, you might write a key function for sorting objects based on an attribute *attr* as **lambda** ob: ob.*attr*, or one for sorting dicts by dict key '*attr*' as **lambda** d: d['*attr*']. (The attrgetter and itemgetter methods of the operator module are useful alternatives to these simple key functions; they're clearer and sharper than **lambda** and offer performance gains as well.)

Older versions of Python used a cmp function, which would take list elements in pairs (*A*, *B*) and return -1, 0, or 1 for each pair depending on which of *A* < *B*, *A* == *B*, or *A* > *B* is true. Sorting using a cmp function is very slow, as it may have to compare every element to every other element (potentially O(N^2) performance). The sort function in current Python versions no longer accepts a cmp function argument. If you are migrating ancient code and only have a function suitable as a cmp argument, you can use functools.cmp_to_key to build from it a key function suitable to pass as the new key argument.

However, several functions in the module heapq, covered in **"The heapq Module"**, do not accept a key argument. In such cases, you can use the

DSU idiom, covered in "The Decorate–Sort–Undecorate Idiom". (Heaps are well worth keeping in mind, since in some cases they can save you from having to perform sorting on all of your data.)

Avoid exec and from ... import *

Code in a function runs faster than code at the top level in a module, because access to a function's local variables is faster than access to globals. If a function contains an `exec` without explicit `dict`s, however, the function slows down. The presence of such an `exec` forces the Python compiler to avoid the modest but important optimization it normally performs regarding access to local variables, since the `exec` might alter the function's namespace. A **`from`** statement of the form:

```
from my_module import *
```

wastes performance too, since it also can alter a function's namespace unpredictably, and therefore inhibits Python's local-variable optimization.

`exec` itself is also quite slow, and even more so if you apply it to a string of source code rather than to a code object. By far the best approach—for performance, for correctness, and for clarity—is to avoid `exec` altogether. It's most often possible to find better (faster, more robust, and clearer) solutions. If you *must* use `exec`, *always* use it with explicit `dict`s, although avoiding `exec` altogether is *far* better, if at all feasible. If you need to `exec` a dynamically obtained string more than once, `compile` the string just once and then repeatedly `exec` the resulting code object.

`eval` works on expressions, not on statements; therefore, while still slow, it avoids some of the worst performance impacts of `exec`. With `eval`, too, you're best advised to use explicit `dict`s. As with `exec`, if you need multiple evaluations of the same dynamically obtained string, `compile` the string once and then repeatedly `eval` the resulting code object. Avoiding `eval` altogether is even better.

See **“Dynamic Execution and exec”** for more details and advice about `exec`, `eval`, and `compile`.

Short-circuiting of Boolean expressions

Python evaluates Boolean expressions from left to right according to the precedence of the operations **`not`**, **`and`**, and **`or`**. When, from evaluating just the leading terms, Python can determine that the overall expression must be **`True`** or **`False`**, it skips the rest of the expression. This feature is known as *short-circuiting*, so called because Python bypasses unneeded processing the same way an electrical short bypasses parts of an electrical circuit.

In this example, both conditions must be **`True`** to continue:

```
if slow_function() and fast_function():
    # ... proceed with processing ...
```

When `fast_function` is going to return **`False`**, it’s faster to evaluate it first, potentially avoiding the call to `slow_function` altogether:

```
if fast_function() and slow_function():
    # ... proceed with processing ...
```

This optimization also applies when the operator is **`or`**, when either case must be **`True`** to continue: when `fast_function` returns **`True`**, Python skips `slow_function` completely.

You can optimize these expressions by considering the order of the expressions’ operators and terms, and order them so that Python evaluates the faster subexpressions first.

SHORT-CIRCUITING MAY BYPASS NEEDED FUNCTIONS

In the preceding examples, when `slow_function` performs some important “side effect” behavior (such as logging to an audit file, or notifying an administrator of a system condition), short-circuiting may unexpectedly skip that behavior. Take care when including necessary behavior as part of a Boolean expression, and do not overoptimize and remove important functionality.

Short-circuiting of iterators

Similarly to short-circuiting in Boolean expressions, you can short-circuit the evaluation of values in an iterator. Python’s built-in functions `all`, `any`, and `next` return after finding the first item in the iterator that meets the given condition, without generating further values:

```
any(x**2 > 100 for x in range(50))
# returns True once it reaches 10, skips the rest

odd_numbers_greater_than_1 = range(3, 100, 2)
all(is_prime(x) for x in odd_numbers_greater_than_1)
# returns False: 3, 5, and 7 are prime but 9 is not

next(c for c in string.ascii_uppercase if c in "AEIOU")
# returns 'A' without checking the remaining characters
```

Your code gains an added advantage when the iterator is specifically a generator, as shown in all three of these cases. When the sequence of items is expensive to produce (as might be the case with records fetched from a database, for example), retrieving those items with a generator and short-circuiting to retrieve only the minimum needed can provide significant performance benefits.

Optimizing loops

Most of your program’s bottlenecks will be in loops, particularly nested loops, because loop bodies execute repeatedly. Python does not implicitly perform any *code hoisting*: if you have any code inside a loop that you could execute just once by hoisting it out of the loop, and the loop is a bottleneck, hoist the code out yourself. Sometimes the presence of code to hoist may not be immediately obvious:

```
def slower(anobject, ahugenumber):
    for i in range(ahugenumber):
        anobject.amethod(i)

def faster(anobject, ahugenumber):
    themethod = anobject.amethod
    for i in range(ahugenumber):
        themethod(i)
```

In this case, the code that `faster` hoists out of the loop is the attribute lookup `anobject.amethod`. `slower` repeats the lookup every time, while `faster` performs it just once. The two functions are not 100% equivalent: it is (barely) conceivable that executing `amethod` might cause such changes on `anobject` that the next lookup for the same named attribute fetches a different method object. This is part of why Python doesn't perform such optimizations itself. In practice, such subtle, obscure, and tricky cases happen very rarely; it's almost invariably safe to perform such optimizations yourself, to squeeze the last drop of performance out of some bottleneck.

Python is faster with local variables than with global ones. If a loop repeatedly accesses a global whose value does not change between iterations, you can "cache" the value in a local variable, and access that instead. This also applies to built-ins:

```
def slightly_slower(asequence, adict):
    for x in asequence:
        adict[x] = hex(x)

def slightly_faster(asequence, adict):
    myhex = hex
    for x in asequence:
        adict[x] = myhex(x)
```

Here, the speedup is very modest.

Do not cache **None**, **True**, or **False**. Those constants are keywords: no further optimization is needed.

List comprehensions and generator expressions can be faster than loops, and, sometimes, so can `map` and `filter`. For optimization purposes, try changing loops into list comprehensions, generator expressions, or perhaps `map` and `filter` calls, where feasible. The performance advantage of `map` and `filter` is nullified, and worse, if you have to use a **lambda** or an extra level of function call. Only when the argument to `map` or `filter` is a built-in function, or a function you'd have to call anyway even from an explicit loop, list comprehension, or generator expression, do you stand to gain some tiny speedup.

The loops that you can replace most naturally with list comprehensions, or `map` and `filter` calls, are ones that build up a list by repeatedly calling `append` on the list. The following example shows this optimization in a microperformance benchmark script (the example includes a call to the the `timeit` convenience function `repeat`, which simply calls `timeit.timeit` the specified number of times):

```
import timeit, operator

def slow(asequence):
    result = []
    for x in asequence:
        result.append(-x)
    return result

def middling(asequence):
    return list(map(operator.neg, asequence))

def fast(asequence):
    return [-x for x in asequence]
```

```
for afunc in slow, middling, fast:
    timing = timeit.repeat('afunc(big_seq)',
                           setup='big_seq=range(500*1000)',
                           globals={'afunc': afunc},
                           repeat=5,
                           number=100)
    for t in timing:
        print(f'{afunc.__name__},{t}')
```

As we reported in the previous edition of this book (using a different set of test parameters):

> *Running this example in v2 on an old laptop shows that `fast` takes about 0.36 seconds, `middling` 0.43 seconds, and `slow` 0.77 seconds. In other words, on that machine, `slow` (the loop of `append` method calls) is about 80 percent slower than `middling` (the single `map` call), and `middling`, in turn, is about 20 percent slower than `fast` (the list comprehension).*
>
> *The list comprehension is the most direct way to express the task being microbenchmarked in this example, so, not surprisingly, it's also fastest—about two times faster than the loop of `append` method calls.*

At that time, using Python 2.7, there was a clear advantage to using the `middling` function over `slow`, and a modest speed increase resulted from using the `fast` function over `middling`. For the versions covered in this edition, the improvement of `fast` over `middling` is much less, if any. Of greater interest is that the `slow` function is now starting to approach the performance of the optimized functions. Also, it is easy to see the progressive performance improvements in successive versions of Python, especially Python 3.11 (see **Figure 17-3**).

The clear lesson is that performance tuning and optimization measures should be revisited when upgrading to newer Python versions.

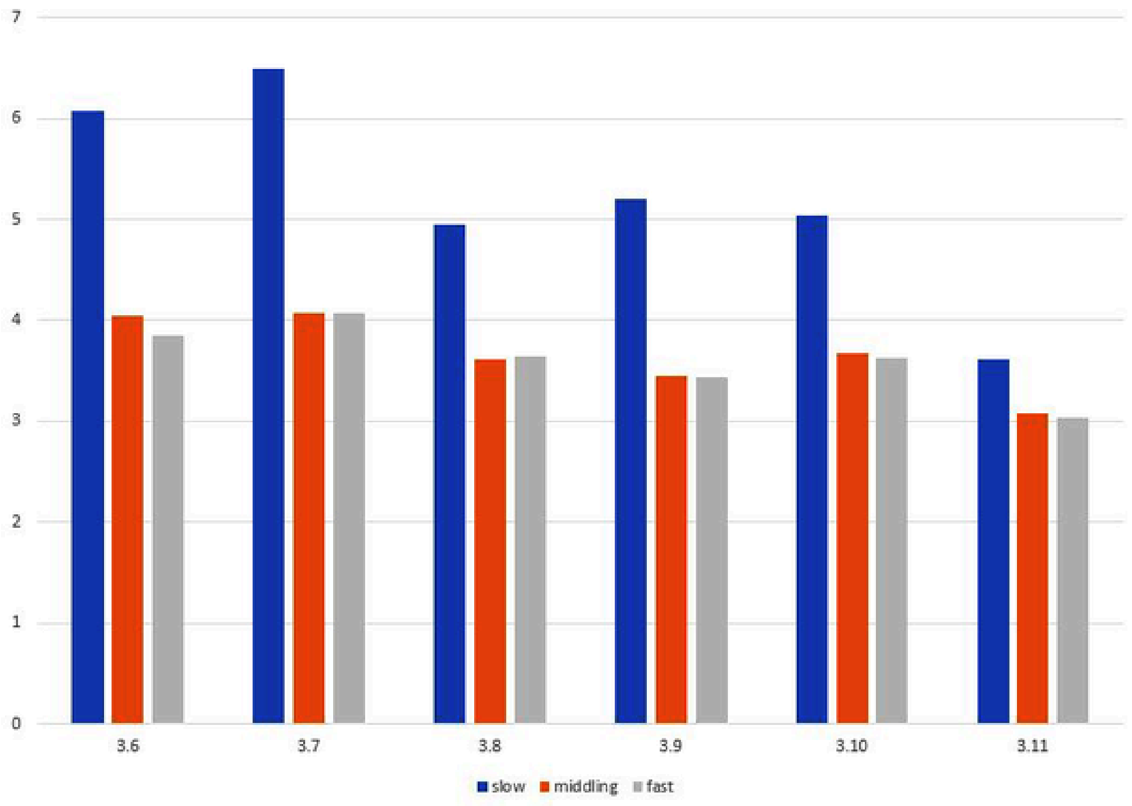


Figure 17-3. *Performance of the example on various Python versions*

Using multiprocessing for heavy CPU work

If you have heavily CPU-bound processing that can be done in independent pieces, then one important way to optimize is to use multiprocessing, as described in **Chapter 15**. You should also consider whether using one of the numeric packages described in **Chapter 16**, capable of applying vector processing to large data sets, is applicable.

Optimizing I/O

If your program does substantial amounts of I/O, it's quite likely that performance bottlenecks are due to I/O, rather than to computation. Such programs are said to be I/O-bound, rather than CPU-bound. Your operating system tries to optimize I/O performance, but you can help it in a couple of ways.

From the point of view of a program's convenience and simplicity, the ideal amount of data to read or write at a time is often small (one character or one line) or very large (an entire file at a time). That's often fine: Python and your operating system work behind the scenes to let your program use convenient logical chunks for I/O, while arranging for physical I/O operations to use chunk sizes more attuned to performance. Reading and writing a whole file at a time is quite likely to be OK for performance as long as the file is not *very* large. Specifically, file-at-a-time I/O is fine as long as the file's data fits very comfortably in physical RAM, leaving ample memory available for your program and operating system to perform whatever other tasks they're doing at the same time. The hard problems of I/O-bound performance come with huge files.

If performance is an issue, *never* use a file's `readline` method, which is limited in the amount of chunking and buffering it can perform. (Using `writelines`, on the other hand, causes no performance problems when that method is convenient for your program.) When reading a text file, loop directly on the file object to get one line at a time with best performance. If the file isn't too huge, and so can conveniently fit in memory, time two versions of your program: one looping directly on the file object, the other reading the whole file into memory. Either may prove faster by a little.

For binary files, particularly large binary files whose contents you need just a part of on each given run of your program, the module `mmap` (covered in **"The mmap Module"**) can sometimes help keep your program simple and boost performance.

Making an I/O-bound program multithreaded sometimes affords substantial performance gains, if you can arrange your architecture accordingly. Start a few worker threads devoted to I/O, have the computational threads request I/O operations from the I/O threads via `Queue` instances, and post the request for each input operation as soon as you know you'll eventually need that data. Performance increases only if there are other tasks your computational threads can perform while I/O threads are blocked waiting for data. You get better performance this way only if you can manage to overlap computation and waiting for data by having different threads do the computing and the waiting. (See **"Threads in Python"** for detailed coverage of Python threading and a suggested architecture.)

On the other hand, a possibly even faster and more scalable approach is to eschew threads in favor of asynchronous (event-driven) architectures, as mentioned in Chapter 15.

[1] This issue is related to "technical debt" and other topics covered in the "'Good enough' is good enough" tech talk by one of this book's authors (that author's favorite tech talk out of the many he delivered!), excellently summarized and discussed by Martin Michlmayr on LWN.net.

[2] The language used in this area is confused and confusing: terms like *dummies*, *fakes*, *spies*, *mocks*, *stubs*, and *test doubles* are utilized by different people to mean slightly different things. For an authoritative approach to terminology and concepts (though not the exact one we use), see the essay "Mocks Aren't Stubs" by Martin Fowler.

[3] That's partly because the structure of the system tends to mirror the structure of the organization, per Conway's law.

[4] However, be sure you know exactly what you're using `doctest` for in any given case: to quote Peter Norvig, writing precisely on this subject: "Know what you're aiming for; if you aim at two targets at once you usually miss them both."

[5] When evaluating `assert` *a* `==` *b*, `pytest` interprets *a* as the observed value and *b* as the expected value (the reverse of `unittest`).

[6] "Oh, but I'll only be running this code for a short time!" is *not* an excuse to get sloppy: the Russian proverb "nothing is more permanent than a temporary solution" is particularly applicable in software. All over the world, plenty of "temporary" code performing crucial tasks is over 50 years old.

[7] A typical case of the Pareto principle in action.

[8] Per Amdahl's law.

[9] Using the invented-for-Python adaptive sorting algorithm Timsort.

[10] A once-slower idiom may be optimized in some future version of Python, so it's worth redoing `timeit` measurements to check for this when you upgrade to newer versions of Python.

11 Even though current Python implementations bend over backward to help reduce the performance hit of this specific, terrible, but common anti-pattern, they can’t catch every occurrence, so don’t count on that!