Chapter 21. The Benchmarking Interlude

Now that we've fully explored function coding and iteration tools, we're going to take a short side trip to put both of them to work. This chapter closes out the function part of this book with a larger case study that times the relative performance of the iteration tools we've met so far, in both standard Python and one of its alternatives.

Along the way, this case study surveys Python's code-timing tools, discusses benchmarking techniques in general, and develops code that's more realistic and useful than most of what we've seen up to this point. We'll also measure the speed of code we've used—data points that may or may not be significant, depending on your programs' goals.

Finally, because this is the last chapter in this part of the book, we'll close with the usual sets of "gotchas" and exercises to help you start coding the ideas you've read about. First, though, let's have some fun with tangible Python code.

Benchmarking with Homegrown Tools

We've met quite a few iteration alternatives in this book. Like much in programming, they represent trade-offs—in terms of both subjective factors like expressiveness, and more objective criteria such as performance. Part of your job as a programmer and engineer is selecting tools based on factors like these.

In terms of performance, this book has mentioned a few times that list comprehensions and `map` calls sometimes have a speed advantage over `for` loop statements. It has also noted that sorting speed varies with ordering, and the generator functions and expressions of the preceding chapter tend to be slower than all the others, though they minimize memory space requirements and don't delay the caller for result generation when there are many results to generate.

All that is generally true today in common usage. That being said, benchmarking comes with some big caveats: both code structure and host architecture can influence speed arbitrarily; Python's performance can vary over time because its internals are constantly being changed and optimized; and the speed of alternative Pythons may differ widely. As noted in Chapter 2, for example, the standard *CPython* may adopt a standard JIT in the future which could change its speed in some contexts, and optimized Pythons like *PyPy* have very difference performance profiles.

In short, if you want to verify speed for yourself, you need to time your code, on your own device, with the Python or Pythons you plan to use, and in the here and now. That's a lot of qualifiers, but benchmarking is an empirical task.

Timer Module: Take 1

Luckily, Python makes it easy to time code with custom tools—though perhaps deceptively so. For example, to get the total time taken to run multiple calls to a function with arbitrary positional arguments, Example 21-1's function coded in a module file might suffice as a first cut.

Example 21-1. timer0.py

```
"Simplistic timing function"

import time
def timer(func, *args):                          # Any positi
    start = time.perf_counter()
    for i in range(100_000):                     # Hardcoded
        func(*args)
    return time.perf_counter() - start           # Total elap
```

This function fetches time values from Python's standard-library `time` module, and subtracts the system start time from the stop time after running 100,000 calls to the passed-in function with the passed-in arguments.

Time comes from `time.perf_counter`, which is defined to be a portable clock with the best resolution to measure a short duration. The difference between two calls is generally used for performance measurement, but includes time for sleeps and is system-wide time. The alternative `time.process_time` is per-process CPU time, and may also be useful in

some roles. See Python's manuals for more details; these calls replace the former and less portable `time.clock`, which was deprecated and dropped since this book's prior edition (breaking most examples here!).

Apart from its `time` calls, the code in Example 21-1 is straightforward and uses tools we've already met. When used on this book's main development computer using macOS and CPython 3.12, its results are these in a REPL:

```
>>> from timer0 import timer
>>> timer(pow, 2, 1000)                # Time to call pc
0.08591239107772708
>>> timer(str.upper, 'hack' * 100)     # Time to call 'h
0.04990812297910452
```

Though functional and simple, the preceding timer is also fairly limited, and deliberately exhibits some classic mistakes in both function design and benchmarking. Among these, it:

- Hardcodes the *repetitions* count at 100k
- Charges the cost of `range` to the tested function's time
- Doesn't support *keyword* arguments in the tested function
- Doesn't give callers a way to *verify* that the tested function actually worked
- Only gives *total* time, which might fluctuate on some busy host machines

In other words, timing code is more complex than you might expect!

Timer Module: Take 2

To be more general and accurate, let's rewrite the preceding section's code to define still simple but more useful timer utility functions we can both use to see how iteration alternative options stack up now, and apply to other timing needs in the future. These functions, in Example 21-2, are coded in a module file again so they can be used in a variety of programs, and have docstrings giving some basic usage details that `help` and PyDoc can display; see Chapter 15 for tips on viewing the in-code documentation of this chapter's timing modules.

Example 21-2. timer.py

```
"""
Homegrown timing tools for arbitrary function calls.
Times one call, total of N, best of N, and best of tota
Pass any number of positional and keyword arguments for
"""

import time
timer = time.perf_counter                          # See

def once(func, *pargs, **kargs):                   # Col
    """
    Time to run func(...) one time.
    Returns (time, result).
    """
    start   = timer()
    result  = func(*pargs, **kargs)                # Unp
    elapsed = timer() - start
    return (elapsed, result)                       # Ret

def total(reps, func, *pargs, **kargs):            # Col
    """
    Total time to run func(...) reps times.
    Returns (total-time, last-result).
    """
    total = 0                                      # Dor
    for i in range(reps):
        time, result = once(func, *pargs, **kargs)
        total += time
    return (total, result)                         # Ret

def bestof(reps, func, *pargs, **kargs):
    """
    Best time among reps runs of func(...).
    Returns (best-time, best-time-result).
    """
    return min(once(func, *pargs, **kargs) for i in rar

def bestoftotal(reps1, reps2, func, *pargs, **kargs):
    """
    Best total time among reps1 runs of [reps2 runs of
    Returns (best-total-time, best-total-time-last-resu
```

```
    """
    return min(total(reps2, func, *pargs, **kargs) for
```

Operationally, this module implements both *total* and *best* times, and a *best of totals* that combines the other two. In each mode, it times calls to any subject function that takes any positional and keyword arguments, by fetching the start time, calling the function with arbitrary arguments, and subtracting the start time from the stop time. Here are the salient points to notice about how this version addresses the shortcomings of its predecessor:

- The *repetitions* count is no longer hardcoded, but passed in as a required argument (or arguments) before the test function and its own arguments, to allow repetitions to vary per call.
- The `range` call's construction and iteration costs are no longer charged to timed functions, because timing has been factored out to the separate `once` function that times just the subject function.
- Any number of both positional and *keyword* arguments for the timed function are now collected and unpacked with starred-argument syntax. They must be sent individually, not in a sequence or dictionary, though callers can unpack argument collections into individual arguments with stars in the top-level call.
- All functions in this module return one of the timed function's *return values* so callers can verify that the function worked. The return value is provided in a two-item result tuple, along with the requested time result.
- New best-of modes return *minimum* times to address fluctuations on the host device, per the next paragraph.

This version's functions also support multiple use cases: `once` times a single call for simple cases; `total` runs many calls, to allow time to accumulate for short-lived functions; `bestof` returns the minimum time among all single calls, to filter out the impacts of other activity on the host; and `bestoftotal` selects the minimum of nested total-time tests, to both apply a best-of filter and run many calls for functions too fast to produce meaningful times.

Importantly, the `min` calls in the best-of variants work to select the best and lowest time, because Python compares collections recursively (from left to right) as we've learned, and result tuples begin with *time*: because it's first in

these `(time, result)` tuples, time dominates and determines the `min` calls' results:

```
>>> min(tup for tup in [(2.0, 3), (3.0, 3), (1.0, 3), (
(0.0, 3)
```

From a larger perspective, because these functions are coded in a module file, they become generally useful tools anywhere we wish to import them. Modules and imports were introduced in Chapter 3, and you'll learn more about them in the next part of this book; for now, simply import the module and call its function to use one of this file's timers. Its results on the same host and in a REPL are similar to its *timer0.py* predecessor, but are more robust:

```
>>> import timer                                  # Imp
>>> help(timer)                                   # Dis
>>> reps, text = 100_000, 'hack' * 100

>>> timer.once(pow, 2, 1000)[0]                   # Not
6.182119250297546e-06
>>> timer.once(str.upper, text)                   # (ti
(3.363005816936493e-06, 'HACKHACKHACK…etc…')

>>> timer.total(reps, pow, 2, 1000)[0]            # Com
0.08979405369609594
>>> timer.total(reps, str.upper, text)[0]         # (ti
0.050884191412478685

>>> timer.bestof(50, pow, 2, 1000)[0]             # Not
1.6265548765659332e-06
>>> timer.bestof(50, str.upper, text)[0]          # (be
8.619390428066254e-07

>>> timer.bestoftotal(50, reps, pow, 2, 1000)[0]
0.07521858718246222
>>> timer.bestoftotal(50, reps, str.upper, text)[0]
0.03947464330121875
```

The last two calls here calculate the *best-of-totals* times—the lowest time among 50 runs, each of which computes the total time to call a function 100k times—roughly corresponding to the `total` times earlier in this listing, but repeated for a minimum that filters out host fluctuations (and sans an extra

charge for `range`). The function used in these last two calls is really just a convenience that wraps `total` for better accuracy, but this is a common timing mode.

Note that `bestoftotal` might replace its `min` of `total` with code like the following to nest `total` in `bestof`:

```
>>> timer.bestof(50, timer.total, reps, str.upper, text
(0.07037258706986904, (0.039281503297388554, 'HACKHACKH
```

But this isn't quite the same. For one thing, the result is nested tuples, reflecting the nested calls. For another, this really times the entire `total` function, not just the subject function it runs. The net effect charges an extra admin-code overhead that's enough to skew the best-of time up, and make it larger than the best total time shown in the nested tuple. By using `min` of `total` instead, `bestoftotal` avoids timing this overhead skew. Subtle but true!

Timing Runner and Script

Now, to time iteration tool speed (our original goal), we'll write a script that defines and submits test functions to the `timer` module. To make it easy to code a variety of tests, let's first define a utility module that does the heavy lifting as a reusable intermediary. Example 21-3 defines a function named `runner` that takes any number of test functions as arguments and passes them off for timing to the `bestoftotal` function imported from Example 21-2.

Example 21-3. timer_runner.py

```
"Run passed-in test functions with the timer.py module"

import timer, sys

def runner(*tests):
    results = []
    print('Python', sys.version.split()[0], 'on', sys.p

    # Time
    for test in tests:
```

```
            besttime, result = timer.bestoftotal(10, 1000,
            results.append(result)
            print(f'{test.__name__:<9}: '
                  f'{besttime:.5f} => [{result[0]}…{result[

    # Verify
    print('Results differ!'
           if any(result != results[0] for result in re
           else 'All results same.')
```

Fine points here: this module's `runner` function displays context with `sys` tools documented in Python's manuals, and steps through all the passed-in functions, printing the `__name__` of each (as we've seen, this is a built-in attribute that gives a function's name). The test-runner code also saves results to verify that they are all the same in a ternary expression at the very end of the process, to be sure we're comparing apples to apples.

Last but not least, the script in Example 21-4 defines the actual tests to be timed and passes them to the imported runner of Example 21-3, which hands them off to the imported timer of Example 21-2. We'll run the file in Example 21-4 as a top-level script to time the relative speeds of the various iteration codings we've studied in this book so far.

Example 21-4. timer_tests.py

```
"Test the relative speed of iteration coding alternativ

from timer_runner import runner
repslist = list(range(10_000))

def forLoop():
    res = []
    for x in repslist:
        res.append(abs(x))
    return res

def listComp():
    return [abs(x) for x in repslist]

def mapCall():
    return list(map(abs, repslist))              # Use

def genExpr():
```

```
    return list(abs(x) for x in repslist)        # Use

def genFunc():
    def gen():
        for x in repslist:
            yield abs(x)
    return list(gen())                           # Use

runner(forLoop, listComp, mapCall, genExpr, genFunc)
```

This script tests five alternative ways to build lists of results. In combination with the test runner, its reported times reflect some 100 million steps for each of the 5 test functions—each builds a list of 10,000 items 1,000 times, and this process is repeated 10 times to get the best-of times. Applying this for each of the 5 test functions yields a whopping 500 million total steps for the script at large (impressive but reasonable on most machines these days).

Notice how we have to run the results of the generator expression and function through the built-in `list` call to force them to yield all of their values; if we did not, we would just produce generators that never do any real work. We must do the same for the `map` result, since it is an iterable, on-demand object as well.

For similar reasons, the inner loops' `range` result is hoisted out to the top of the module to remove its construction cost from total time, and wrapped in a `list` call so that its traversal cost isn't skewed by being a generator. This may be overshadowed by the cost of the inner iterations' loops, but it's best to remove as many variables as we can. For example, though `range` supports multiple scans, tests' times inflate by some 25% if its `list` wrapper is removed.

Iteration Results

When the top-level script of the prior section is run under the standard CPython 3.12 on the same macOS host, it prints the following with total times in seconds—after cueing the *drum roll*, that is:

```
$ python3 timer_tests.py
Python 3.12.2 on darwin
forLoop  : 0.26035 => [0...9999]
```

```
listComp : 0.20781 => [0...9999]
mapCall  : 0.14399 => [0...9999]
genExpr  : 0.41133 => [0...9999]
genFunc  : 0.41203 => [0...9999]
All results same.
```

In short, `map` calls are faster than list comprehensions, which are quicker than `for` loops, and both generator expressions and functions come in last and roughly tied for slowest. Perhaps surprisingly, generator expressions run much slower than equivalent list comprehensions today. As warned in the prior chapter, although wrapping a generator expression in a `list` call makes it *functionally* equivalent to a list comprehension, the internal *implementations* of the two expressions appear to differ:

```
return [abs(x) for x in repslist]            # 0.20 sec
return list(abs(x) for x in repslist)        # 0.41 sec
```

We're also effectively timing the `list` call for the generator test, but given that this does not seem to hamper `map` , it's likely moot. Though the exact cause for the difference would require deeper analysis (and probably source code spelunking), this seems to make sense given that the generator expression must do extra work to save and restore its state during value production; the list comprehension does not, and runs quicker here and in other tests ahead.

Other Pythons' results

For comparison, following are the same tests' speed results on the same host using the current *PyPy*—the optimized Python implementation discussed in Chapter 2, whose current 7.3 release implements the Python 3.10 language. PyPy is roughly *5X* quicker than CPython here (and up to *10X*), though its timing results can vary widely if its JIT has not yet compiled code in full (and a future and currently hypothetical CPython JIT may or may not even the race):

```
$ pypy3 timer_tests.py
Python 3.10.14 on darwin
forLoop  : 0.04329 => [0...9999]
listComp : 0.01876 => [0...9999]
mapCall  : 0.04132 => [0...9999]
genExpr  : 0.08744 => [0...9999]
```

```
genFunc  : 0.08726 => [0...9999]
All results same.
```

On PyPy alone, list comprehensions win the title in this test today, but generators still lose soundly, and the fact that all of PyPy's results are so much quicker today seems the larger point here. On CPython, `map` is still quickest so far.

Interestingly, these results are very different than they were in this book's prior edition on Windows under *CPython 3.3*—when generators placed in the middle of the pack between list comprehensions and `for` loops, `for` loops fared substantially worse than they do today, and the script had a different name for historical reasons:

```
C:\code> c:\python33\python timeseqs.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC
forLoop  : 1.33290 => [0...9999]
listComp : 0.69658 => [0...9999]
mapCall  : 0.56483 => [0...9999]
genExpr  : 1.08457 => [0...9999]
genFunc  : 1.07623 => [0...9999]
```

While these absolute times naturally reflect older and slower test hosts, these tools' *relative* performance has clearly changed in the last 12 years—and probably should be expected to do so again in another dozen!

For more good times: Function calls and map

All of the foregoing is true as advertised, but watch what happens when Example 21-5 performs an *inline* operation on each iteration, such as a `+` expression, instead of calling a built-in function like `abs`. Only the test functions' operations (in bold) need to be modified here, and the imported and reused runner handles tests generically.

Example 21-5. timer_tests2.py

```
from timer_runner import runner
repslist = list(range(10_000))

def forLoop():
    res = []
```

```
        for x in repslist:
            res.append(x + 10)
        return res

    def listComp():
        return [x + 10 for x in repslist]

    def mapCall():
        return list(map((lambda x: x + 10), repslist))

    def genExpr():
        return list(x + 10 for x in repslist)

    def genFunc():
        def gen():
            for x in repslist:
                yield x + 10
        return list(gen())

    runner(forLoop, listComp, mapCall, genExpr, genFunc)
```

Now, `map` is *slower* than the `for` loop statements, despite the fact that the looping-statements version is larger in terms of code. This could either mean that the need to *call* a user-defined function makes `map` slower—or equivalently, that the *lack* of function calls makes the others quicker. On CPython 3.12 and the same host as before:

```
$ python3 timer_tests2.py
Python 3.12.2 on darwin
forLoop  : 0.28682 => [10...10009]
listComp : 0.24389 => [10...10009]
mapCall  : 0.49622 => [10...10009]
genExpr  : 0.44047 => [10...10009]
genFunc  : 0.44476 => [10...10009]
All results same.
```

These results have also been consistent in CPython: the prior edition's Python 3.3 results on a slower machine were again relatively similar, discounting test machine differences. Because the interpreter optimizes so much internally, performance analysis of Python code like this is a very tricky affair. Without numbers, it's virtually impossible to guess which method will perform the best; again, the best you can do is time your code with your test parameters.

In this case, what we can say is that on this Python, using a user-defined `lambda` function in `map` calls seems to slow its performance disproportionately (though `+` is also slower than a trivial `abs` across the board), and that list comprehensions run quickest in this case (though slower than `map` in some others). List comprehensions seem consistently faster than `for` loops, but even this must be qualified—the list comprehension's relative speed might be affected by its extra syntax (e.g., `if` filters), Python changes, and usage modes we did not time here.

For deeper truth, Example 21-6 codes one last takeoff on our tests to apply a simple user-defined function in *all five* iterations timed. Again, we must only modify the relevant (and bold) bits of the test functions themselves.

Example 21-6. timer_tests3.py

```
from timer_runner import runner
repslist = list(range(10_000))

def F(x): return x

def forLoop():
    res = []
    for x in repslist:
        res.append(F(x))
    return res

def listComp():
    return [F(x) for x in repslist]

def mapCall():
    return list(map(F, repslist))

def genExpr():
    return list(F(x) for x in repslist)

def genFunc():
    def gen():
        for x in repslist:
            yield F(x)
    return list(gen())

runner(forLoop, listComp, mapCall, genExpr, genFunc)
```

When coded this way and run in CPython 3.12 on the same host again, `map` improves its relative times, and comes in second between list comprehensions and `for` loops—instead of being slower than all others as it was for `+`:

```
$ python3 timer_tests3.py
Python 3.12.2 on darwin
forLoop  : 0.36206 => [0...9999]
listComp : 0.31181 => [0...9999]
mapCall  : 0.35479 => [0...9999]
genExpr  : 0.50531 => [0...9999]
genFunc  : 0.50290 => [0...9999]
All results same.
```

That is, `map` may be slower simply *because it requires function calls*, and function calls are relatively slow in general. Since `map` can't avoid calling functions, it may lose by association, and the other iteration tools may win when they can use expressions instead. On the other hand, this hypothesis alone can't explain the better showing for `map` in the `abs` results of the first *timer_tests.py*: calling *built-in* functions may be a special and fast case for `map` in CPython (a theory supported by "Conclusion: Comparing tools" and `ord` at the end of this chapter's benchmarking safari).

All this being said, performance should not be your primary concern when writing Python code—the first thing you should do to optimize Python code is to *not optimize Python code*! Write for readability and simplicity first, then optimize later, if and only if needed. It could very well be that any of the five iteration alternatives we've timed is quick enough for the data sets your program needs to process; if so, program clarity should be the chief goal.

More Module Mods

Our *timer.py* module works as designed, but it could be a bit more user-friendly. Most obviously, its functions require passing in repetition counts as first arguments, and provide no defaults for them—a minor point, perhaps, but less than ideal in a general-purpose tool. To do better, it could allow repetition counts to be passed in as *keyword* arguments with *defaults*, much the same way we did for the `print` emulators of Chapter 18.

Here, though, these arguments' names would have to be distinct to avoid clashing with those of the timed function (e.g., `_reps` instead of `reps`). While we're at it, the function object could also be a *positional-only* argument to prevent name `func` from clashing with a keyword argument of the same name in the timed subject. Example 21-7 codes the required mods (sans docs for space). Review Chapter 18's argument-ordering coverage for more insight.

Example 21-7. timer2.py

```
"Use keyword-only arguments with defaults for reps, anc

import time
timer = time.perf_counter

def once(func, /, *pargs, **kargs):
    start   = timer()
    result  = func(*pargs, **kargs)
    elapsed = timer() - start
    return (elapsed, result)

def total(func, /, *pargs, _reps=100_000, **kargs):
    total = 0
    for i in range(_reps):
        time, result = once(func, *pargs, **kargs)
        total += time
    return (total, result)

def bestof(func, /, *pargs, _reps=5, **kargs):
    return min(once(func, *pargs, **kargs) for i in rar

def bestoftotal(func, /, *pargs, _reps1=50, **kargs):
    return min(total(func, *pargs, **kargs) for i in ra
```

This version is not backward compatible: it uses different names and modes for repetition arguments, which means the *timer_runner.py* we wrote earlier would require a minor edit to use it (see the examples package's *timer2_*.py*). Otherwise, it works the same—compare its output on the same host with *timer.py*'s results listed at Example 21-2:

```
$ python3
>>> import timer2
>>> timer2.total(pow, 2, 1000, _reps=100_000)[0]
```

```
0.0865794476121664
>>> timer2.total(str.upper, 'hack' * 100, _reps=100_000
0.04620114527642727

>>> timer2.bestoftotal(pow, 2, 1000, _reps1=50, _reps=1
0.07179990829899907
>>> timer2.bestoftotal(str.upper, 'hack' * 100, _reps1=
0.0393348871730268
```

This time, though, we can allow the functions' repetition defaults to apply or not:

```
>>> timer2.total(str.upper, 'hack' * 100)[0]
0.047992002684623
>>> timer2.bestoftotal(str.upper, 'hack' * 100)[0]
0.03935634717345238
>>> timer2.bestoftotal(str.upper, 'hack' * 100, _reps=1
0.00393257150426507
```

Notice how the `_reps` argument for `total`, if passed, in `bestoftotal` calls is propagated along in `**kargs` because it's not matched otherwise. For more vetting of this, time user-defined functions with richer argument headers as in the following. Per the first four timings, passing keyword arguments to a timed function adds a small time cost for unpacking—an unavoidable overhead shared by the original *timer.py*, but irrelevant when collecting relative times:

```
>>> def f(a, b, c=88, d=99): return(a, b, c, d)

>>> f(1, 2)
(1, 2, 88, 99)
>>> timer2.bestoftotal(f, 1, 2, _reps1=50, _reps=100_00
(0.014080120716243982, (1, 2, 88, 99))
>>> timer2.bestoftotal(f, 1, 2, c=66, d=77, _reps1=50,
(0.021575820166617632, (1, 2, 66, 77))

>>> timer2.bestoftotal(f, 1, 2, c=66, d=77, _reps1=50)
(0.021694606635719538, (1, 2, 66, 77))
>>> timer2.bestoftotal(f, 1, 2, c=66, d=77, _reps=100_0
(0.021556788589805365, (1, 2, 66, 77))

>>> def f(a, *b, c=88, **d): return(a, b, c, d)
```

```
>>> f(1, 2, 3, c=66, d=77, e=88)
(1, (2, 3), 66, {'d': 77, 'e': 88})
>>> timer2.bestoftotal(f, 1, 2, 3, c=66, d=77, e=88)
(0.030859854072332382, (1, (2, 3), 66, {'d': 77, 'e': 8

>>> timer2.bestoftotal(f, 1, 2, 3, c=66, d=77, e=88, _r
(0.030802161898463964, (1, (2, 3), 66, {'d': 77, 'e': 8
>>> timer2.bestoftotal(f, 1, 2, 3, c=66, d=77, e=88, _r
(0.0030745575204491615, (1, (2, 3), 66, {'d': 77, 'e':
```

Beyond this, custom benchmarking code is fun but open ended, and this section must stop here for space. As next steps, you might modify the timing script to measure the speed of *set and dictionary* comprehensions and their `for` equivalents (open questions we'll return to ahead); explore Python's `profile` and `cProfile` modules (tools that create full-program profiles instead of focused benchmarks); or explore Python's `timeit` module, which works much like some of this chapter's homegrown code, and offers extra options—as you'll learn in the next section.

NOTE

Decorator timers preview: Notice how we must pass functions into the timers manually here. In Chapter 39, we'll code *decorator*-based timer alternatives with which timed functions are called normally, but require extra `@` preamble syntax where defined. Decorators may be more useful to instrument functions with timing logic when they are already being used within a larger system, and don't as easily support the specific test-call patterns assumed here—when decorated, *every* call to the function runs the timing logic, which is either a plus or minus depending on your goals.

Benchmarking with Python's timeit

The preceding section used custom timing functions to compare code speed. As teased there, the Python standard library also ships with a module named `timeit` that can be used in similar ways, but offers added flexibility, with support for timing code strings in addition to functions, as well as a command-line mode.

As usual in Python, it's important to understand fundamental principles like those illustrated in the prior section. Python's "batteries included" approach

means you'll usually find precoded options for most goals, though you still need to know the ideas underlying them to choose and use them properly. Indeed, the `timeit` module is a prime example of this—it's had a history of being misused by newcomers who didn't yet understand the concepts it embodies. Now that we've learned the basics, though, let's move ahead to a tool that can automate much of our work.

Basic timeit Usage

Let's start with this module's fundamentals before leveraging them in larger scripts. With `timeit`, tests are specified by either *callable objects* or *statement strings*; the latter can hold multiple statements if they use `;` separators or `\n` characters for line breaks, and spaces or tabs to indent statements in nested blocks (e.g., `\n\t`). Tests may also give setup actions, and can be launched from both *command lines* and *API calls*, and from both scripts and the REPL.

API-calls mode

For example, the `timeit` module's `repeat` call returns a list giving the total time taken to run a test a `number` of times, for each of `repeat` runs—the `min` of this list yields the best time among the runs, and helps filter out system load fluctuations that can otherwise skew timing results artificially high (like our earlier `bestoftotal`). Code to be timed is passed to the `stmt` keyword (or first positional) argument, and may be a string or no-argument function.

The following shows this call in action in REPLs on macOS, timing a list comprehension on both *CPython 3.12* and the optimized *PyPy* implementation of Python described in Chapter 2 (as earlier, its tested 7.3 release implements the Python 3.10 language). The results here give the best total time in seconds among 5 runs that each execute the code string 1,000 times; the code string itself constructs a 1,000-item list of integers each time through (`timeit` also has reasonable defaults for repeat counts, but being explicit ensures comparability):

```
$ python3
>>> import timeit
>>> min(timeit.repeat(stmt='[x ** 2 for x in range(1000
0.05187674192711711
```

```
$ pypy3
>>>> import timeit
>>>> min(timeit.repeat(stmt='[x ** 2 for x in range(10
0.00252467580139637
```

You'll notice that PyPy checks in at *20X* faster than CPython 3.12. This is a small artificial benchmark, of course, but stunning nonetheless, and reflects a relative speed ranking that is generally supported by other tests run in this book (though CPython will still beat PyPy on some types of code ahead, and again, may improve with a future JIT). To be fair, this particular test measures the speed of both a list comprehension and integer math, but integer math is ubiquitous in Python code, and PyPy's win is larger on noninteger tests (try squaring a float to see for yourself).

These results also differ from the preceding section's relative version speeds, where PyPy was some *10X* quicker for list comprehensions. Apart from the different type of code being timed here, the different coding structure inside `timeit` may have an effect too—for code strings like those tested here, `timeit` builds, compiles, and executes a function `def` statement string that embeds the test string, thereby avoiding a function call per inner loop. This is irrelevant from a relative-speed perspective, though: times from the same given tool are comparable.

You'll also notice that code to be timed is a *string* here. As you'll see ahead, this requires manually coded line breaks and indentation in some usage modes, and all code to be timed this way must conform to Python's rules for string literals, quotes, and escapes. For instance, a code string cannot embed the same quotes used to enclose it without also escaping them. You can avoid this potential downside by timing callables, though they're limited to zero arguments.

Command-line mode

The `timeit` module also can be run as a script from a command line—either by explicit pathname, or, more portably, automatically located on the module search path with Python's `-m` switch (we used this earlier to launch PyDoc in Chapter 15 and IDLE in Chapter 3). In this mode, you'll need to *quote or escape* Python code in the command line per your console shell's rules;

double quotes are generally portable, but this also qualifies as a potential downside.

Also in this mode, `timeit` reports the average time for a *single* `-n` loop, in either "usec" microseconds (millionths), "msec" milliseconds (thousandths), "sec" seconds, or "nsec" nanoseconds; to compare results here to the total time values reported by API calls, multiply by the number of loops run—51 usec here * 1,000 loops is 51k usec, 51 msec, and 0.051 seconds in total time. For CPython 3.12 on macOS:

```
$ python3 -m timeit -n 1000 "[x ** 2 for x in range(100
1000 loops, best of 5: 51.9 usec per loop

$ python3 -m timeit -n 1000 -r 50 "[x ** 2 for x in rar
1000 loops, best of 50: 51.8 usec per loop
```

As another example, we can use command lines to verify that choice of timer call doesn't impact speed comparisons run in this chapter so far. `timeit` uses `time.perf_counter` by default but its `-p` flag instructs it to instead use `time.process_time`, both discussed earlier (spoiler: as tested, there's no discernible difference):

```
$ python3 -m timeit -p -n 1000 -r 50 "[x ** 2 for x in
1000 loops, best of 50: 51.8 usec per loop
```

We can also use `timeit` to see how PyPy stacks up again—but we're in for a bit of a surprise:

```
$ pypy3 -m timeit -n 1000 -r 50 "[x ** 2 for x in range
WARNING: timeit is a very unreliable tool. use pyperf o
else for real measurements
pypy3 -m pip install pyperf
pypy3 -m pyperf timeit -n '1000' -r '50' '[x ** 2 for x
----------------------------------------------------
1000 loops, average of 50: 1.54 +- 0.568 usec per loop
```

As you can see, PyPy has customized command-line mode in its version of `timeit` to both emit a subjective redirect to an alternative timer, as well as

modify the result display to show averages instead of minimums. These are both opinionated mods made since this book's prior edition, though perhaps understandable for a tool so focused on (and sensitive to) benchmark results. See the web for more on the suggested *pyperf*; it's not part of Python's standard library and requires a separate install, but may offer more advanced timing options that may or may not apply to your goals.

Handling multiline statements

Happily, `timeit` 's API mode is still opinion-free today. To time larger blocks of code in *API-call* mode, either load code from a file, use a triple-quoted block string, or use newlines and tabs or spaces to satisfy Python's syntax. Because you pass Python string objects to a Python function in this mode, there are no shell considerations, though be careful to handle nested quotes with escapes or mixed quotes if needed. The following, for instance, times Chapter 13 loop alternatives in CPython 3.12; you can use the same pattern to time the file-line-reader alternatives in Chapter 14:

```
$ python3
>>> import timeit
>>> min(timeit.repeat(number=100_000, repeat=5,
        stmt='L = [1, 2, 3, 4, 5]\nfor i in range(len(L
0.028153221122920513

>>> min(timeit.repeat(number=100_000, repeat=5,
        stmt='L = [1, 2, 3, 4, 5]\ni=0\nwhile i < len(L
0.03337613819167018

>>> min(timeit.repeat(number=100_000, repeat=5,
        stmt='L = [1, 2, 3, 4, 5]\nM = [x + 1 for x in
0.021507765166461468
```

To run multiline statements like these in *command-line* mode, appease your shell by passing each statement line as a separate argument, with whitespace for indentation— `timeit` concatenates all the lines together with a newline character between them, and later re-indents for its own statement-nesting purposes. Leading spaces may work better for indentation than tabs in this mode due to shell variability, and be sure to quote the code arguments if required by your shell (the first of the following was line-split here to fit, but must be input on a single line in some shells):

```
$ python3 -m timeit -n 100000 -r 5 "L = [1,2,3,4,5]" "i
 "    L[i] += 1" "    i += 1"
100000 loops, best of 5: 332 nsec per loop

$ python3 -m timeit -n 100000 -r 5 "L = [1,2,3,4,5]" "M
100000 loops, best of 5: 218 nsec per loop
```

Other timeit usage modes

The `timeit` module also allows you to provide *setup* code that is run in the main statement's scope, but whose time is not charged to the main statement's total—useful for initialization code you wish to exclude from total time, such as imports of required modules and builds of test data. Because they're run in the same scope, any names created by setup code are available to the main test statement; names defined in the interactive shell generally are not.

To specify setup code, use a `-s` in command-line mode (or many of these for multiline setups) and a `setup` argument string in API-call mode. This can focus tests more sharply, as in the following, whose second command splits list initialization off to a setup statement to time just iteration in command-line mode. As a general rule of thumb, though, the more code you include in a test statement, the more applicable its results will generally be to realistic code:

```
$ python3 -m timeit -n 100000 -r 5 "L = [1,2,3,4,5]" "M
100000 loops, best of 5: 211 nsec per loop

$ python3 -m timeit -n 100000 -r 5 -s "L = [1,2,3,4,5]"
100000 loops, best of 5: 175 nsec per loop
```

Timing sort speed

To demo setup code in API-call mode, the interaction that follows times a sort-based option in Chapter 18's minimum-value example. To avoid page flipping, here's the function it times, in the module *mins.py* (Example 18-2):

```
def min4(*args):
    return sorted(args)[0]
```

And here are the results—proving indirectly that sequences *sort much faster* when they are ordered than they do with randomly ordered contents (to see for yourself, run this in Chapter 18's code folder):

```
>>> from timeit import repeat                          # Stanc

>>> min(repeat(number=1000, repeat=5,                  # Sort
        setup='from mins import min4\n'                # Adjac
              'vals=list(range(1000))',                # Code
        stmt= 'min4(*vals)'))                          # First
0.011066518723964691

>>> min(repeat(number=1000, repeat=5,                  # Sort
        setup='from mins import min4\n'
              'import random\n'
              'vals=[random.random() for i in range(100
        stmt= 'min4(*vals)'))
0.068130933213979
```

See Python's manuals for more on the `random` module used here and in earlier chapters, as well as more on `timeit`. Not shown here, `timeit` also lets you ask for just total time, time callable objects instead of strings, use a class-based API, and leverage additional command-line switches and API-call arguments we don't have space to cover. Instead, the next section codes a `timeit` utility that goes beyond manual command lines and REPL calls.

Automating timeit Benchmarking

Rather than going into more `timeit` details, let's study a program that deploys it to time both coding alternatives and Python versions. This will let us easily define a set of tests to time in a separate file, and will allow us collect data from multiple Pythons, both individually and grouped (though grouped mode comes with limits today, as you'll see).

Benchmark module

To get started, the module file in Example 21-8, *pybench.py*, is set up to time a sequence of statements coded in scripts that import and use it, with either the Python running its code or all Python versions named in a list. It uses some application-level tools described ahead. Because it mostly applies ideas we've

already explored and is amply documented, though, it's listed as mostly self-study material, and an exercise in reading Python code.

Example 21-8. pybench.py

```
r"""
Time the speed of one or more Pythons on multiple code-
benchmarks with timeit.  This is a function, to allow t
to vary.  It times all code strings in a passed list, i

1) The Python running this script, by timeit API calls
2) Multiple Pythons whose paths are passed in a list, t
   the output of timeit command lines run by os.popen t
   Python's -m switch to find timeit on the module sear

In command-line mode (2) only, this replaces all " in t
with ', to avoid clashes with argument quoting; splits
statements into one quoted argument per line so all wil
and replaces all \t in indentation with 4 spaces for ur

Caveats:
- Command-line mode (only) uses naive quoting and MAY F
  embeds and requires double quotes; quoted code is inc
  with the host shell; or command length exceeds shell
- PyPy is largely unusable in command-line mode (2) toc
  modified timeit output in this mode is jarring in the
- This does not (yet?) support a setup statement in any
  time of all code in the test stmt is charged to its t

As fallbacks on fails, use either this module's API-cal
test one Python at a time, or the homegrown timer.py mo
"""

import sys, os, time, timeit
defnum, defrep= 1000, 5    # May vary per stmt

def show_context():
    """
    Show run's context using an arguably gratuitous f-s
    that fails on 3.10 PyPy without "..." for nested '
    """
    print(f"Python {'.'.join(str(x) for x in sys.versio
          f' on {sys.platform}'
          f" at {time.strftime('%b-%d-%Y, %H:%M:%S')}")
```

```
def runner(stmts, pythons=None, tracecmd=False):
    """
    Main logic: run tests per input lists which determi
    stmts:   [(number?, repeat?, stmt-string)]
    pythons: None=host python only, or [python-executab
    """
    show_context()
    for (number, repeat, stmt) in stmts:
        number = number or defnum
        repeat = repeat or defrep    # 0=default

        if not pythons:
            # Run stmt on this python: API call
            # No need to split lines or quote here
            best = min(timeit.repeat(stmt=stmt, number=
            print(f'{best:.4f}  {stmt[:70]!r}')

        else:
            # Run stmt on all pythons: command line
            # Split lines into quoted arguments
            print('-' * 80)
            print(repr(stmt))
            for python in pythons:
                stmt  = stmt.replace('"', "'")
                stmt  = stmt.replace('\t', ' ' * 4)
                lines = stmt.split('\n')
                args  = ' '.join(f'"{line}"' for line i

                oscmd = f'{python} -m timeit -n {number
                print(oscmd if tracecmd else python)
                print('\t' + os.popen(oscmd).read().rst
```

Benchmark script

This preceding file is really only half the picture. Testing scripts use this module's function, passing in concrete though variable lists of statements and Pythons to be tested, as appropriate for the usage mode desired. For example, the script in Example 21-9, *pybench_tests.py*, tests a handful of statements and Pythons by importing and using Example 21-8, and allows command-line arguments to determine part of its operation: `-a` tests all listed Pythons instead of just one, and an added `-t` traces constructed command lines in full so you can see how quotes, multiline statements, and tabs are handled per

`timeit` 's command-line formats shown earlier (see both files' docstrings for more details).

Example 21-9. pybench_tests.py

```
"""
Run pybench.py to time one or more Pythons on multiple
Use command-line arguments (which appear in sys.argv) t

<python> pybench_tests.py
    times just the hosting Python on all code listed ir
python3 pybench_tests.py -a
    times all stmts in all pythons whose paths are list
python3 pybench_tests.py -a -t
    same as -a, but also traces command lines in full

Edit stms below to change tested code, and edit pythons
paths of Python executables to be tested in -a mode.  T
path, start its REPL, run "import sys", and inspect "sy
"""

import pybench, sys

pythons = [
    '/Library/Frameworks/Python.framework/Versions/3.12
    '/Users/me/Downloads/pypy3.10-v7.3.16-macos_x86_64/
]

stmts = [
# Iterations
    (0, 0, '[x ** 2 for x in range(1000)]'),
    (0, 0, 'res=[]\nfor x in range(1000): res.append(x
    (0, 0, 'list(map(lambda x: x ** 2, range(1000)))'),
    (0, 0, 'list(x ** 2 for x in range(1000))'),
# String ops
    (0, 0, "s = 'hack' * 2500\nx = [s[i] for i in range
    (0, 0, "s = '?'\nfor i in range(10_000): s += '?'")
]

tracecmd = '-t' in sys.argv                             #
pythons  = pythons if '-a' in sys.argv else None        #
pybench.runner(stmts, pythons, tracecmd)                #
```

Timing individual Pythons

The following is the preceding script's output when run to test a *specific Python*—the one running the script. This mode uses direct `timeit` API calls, not command lines, with total time listed in the left column, the statement tested quoted on the right, and a context line at the top courtesy of Python's `sys` and `time` modules (see its manuals for more info). These two runs again use CPython 3.12 and PyPy 7.3 (i.e., 3.10), respectively, on the same host:

```
$ python3 pybench_tests.py
Python 3.12.2 on darwin at Jun-27-2024, 15:21:02
0.0533  '[x ** 2 for x in range(1000)]'
0.0605  'res=[]\nfor x in range(1000): res.append(x **
0.0804  'list(map(lambda x: x ** 2, range(1000)))'
0.0759  'list(x ** 2 for x in range(1000))'
0.4042  "s = 'hack' * 2500\nx = [s[i] for i in range(10
0.8061  "s = '?'\nfor i in range(10_000): s += '?'"

$ pypy3 pybench_tests.py
Python 3.10.14 on darwin at Jun-27-2024, 15:22:14
0.0020  '[x ** 2 for x in range(1000)]'
0.0040  'res=[]\nfor x in range(1000): res.append(x **
0.0030  'list(map(lambda x: x ** 2, range(1000)))'
0.0077  'list(x ** 2 for x in range(1000))'
0.0529  "s = 'hack' * 2500\nx = [s[i] for i in range(10
1.2942  "s = '?'\nfor i in range(10_000): s += '?'"
```

Drawing conclusions from this is left as a suggested exercise, but notice that PyPy actually *loses* to CPython on the very last test run. Again, performance can vary per code, and absolutes in benchmarking are perilous at best.

Timing multiple Pythons

As noted, this script can also test multiple Pythons for each statement string, by adding a `-a` to the command line. In this mode the script itself is run by CPython 3.12, which launches shell command lines that start other Pythons to run the `timeit` module on the statement strings. To make this work, this mode must split, format, and quote multiline statements for use in command lines according to both `timeit` expectations and shell requirements.

This mode also relies on the `-m` Python command-line flag to locate `timeit` on the module search path and run it as a script, as well as the `os.popen` and `sys.argv` standard-library tools to run a shell command and inspect command-line arguments, respectively. Add a `-t` to trace commands run, and see Python manuals and other resources for more on these tools; `os.popen` is also mentioned briefly in Chapters 3, 9, and 13. Here is this mode's result:

```
$ python3 pybench_tests.py -a
Python 3.12.2 on darwin at Jun-27-2024, 16:12:39
------------------------------------------------------------
'[x ** 2 for x in range(1000)]'
/Library/Frameworks/Python.framework/Versions/3.12/bin/
    1000 loops, best of 5: 52.7 usec per loop
/Users/me/Downloads/pypy3.10-v7.3.16-macos_x86_64/bin/p
    WARNING: timeit is a very unreliable tool. use pype
    else for real measurements
pypy3 -m pip install pyperf
pypy3 -m pyperf timeit -n '1000' -r '5' '[x ** 2 for x
------------------------------------------------------------
1000 loops, average of 5: 2.66 +- 1.55 usec per loop (u
…more output clipped…
```

Regrettably, this all-Pythons mode is nearly unusable today: the opinionated inserts and mods made to `timeit` by PyPy render its command-line output very different from the norm, and very difficult to read or list here. Moreover, there's no way to disable these (short of dropping CPython's *timeit.py* into PyPy's library folder, which seems too much for this demo to ask). Hence, while this script's `-a` mode still works in this edition, it may be best used to time a set of Pythons that do not subjectively mod the behavior of a widely used standard-library module like `timeit`.

Timing set and dictionary iterations

The good news is that *single-Python* mode still allows us to easily script a set of benchmarks—even on PyPy. The script in Example 21-10, for instance, uses the driver module in Example 21-8 to see how sets and dictionaries fare.

Example 21-10. pybench_tests2.py

```
import pybench

stmts = [
# Sets
    (0, 0, '{x ** 2 for x in range(1000)}'),
    (0, 0, 'set(x ** 2 for x in range(1000))'),
    (0, 0, 's=set()\nfor x in range(1000): s.add(x ** 2
# Dicts
    (0, 0, '{x: x ** 2 for x in range(1000)}'),
    (0, 0, 'dict((x, x ** 2) for x in range(1000))'),
    (0, 0, 'd={}\nfor x in range(1000): d[x] = x ** 2')
]

pybench.runner(stmts, None, False)    # No -a mode in t
```

As suggested in the preceding chapter, passing a generator to a type name is indeed substantially slower than other construction schemes—as this script's output on both CPython 3.12 and PyPy 7.3 (3.10) finally proves:

```
$ python3 pybench_tests2.py
Python 3.12.2 on darwin at Jun-27-2024, 16:20:19
0.0746  '{x ** 2 for x in range(1000)}'
0.0947  'set(x ** 2 for x in range(1000))'
0.0834  's=set()\nfor x in range(1000): s.add(x ** 2)'
0.0745  '{x: x ** 2 for x in range(1000)}'
0.1174  'dict((x, x ** 2) for x in range(1000))'
0.0754  'd={}\nfor x in range(1000): d[x] = x ** 2'

$ pypy3 pybench_tests2.py
Python 3.10.14 on darwin at Jun-27-2024, 16:22:18
0.0191  '{x ** 2 for x in range(1000)}'
0.0222  'set(x ** 2 for x in range(1000))'
0.0186  's=set()\nfor x in range(1000): s.add(x ** 2)'
0.0188  '{x: x ** 2 for x in range(1000)}'
0.0398  'dict((x, x ** 2) for x in range(1000))'
0.0185  'd={}\nfor x in range(1000): d[x] = x ** 2'
```

Conclusion: Comparing tools

If you have time, check out *pybench_tests3.py* in the book example's package (omitted here for space) for another test that verifies that this section's `timeit` tools turn in results similar to the earlier *timer.py* homegrown tools. Its findings generally jive with those in "Iteration Results", though they arrive at them by very different means:

```
$ python3 pybench_tests3.py
Python 3.12.2 on darwin at Jun-27-2024, 16:26:44
0.2766  "res=[]\nfor x in 'hack' * 2500: res.append(orc
0.2173  "[ord(x) for x in 'hack' * 2500]"
0.1430  "list(map(ord, 'hack' * 2500))"
0.4172  "list(ord(x) for x in 'hack' * 2500)"
```

For more fidelity, study this chapter's code and run more tests on your own. Benchmarks can be great sport, but we'll have to leave further excursions as suggested exercises. Here, it's time to wrap up this chapter and part, so we can move on to learn more about the modules we've been using informally since Chapter 16.

NOTE

Cross-platform results: As a bonus, folder `_benchmark-platform-results` in the book's examples package collects CPython's results for this chapter's *timer* and *pybench* tests on multiple platforms—macOS, Windows, Android, and Linux. Due to Python and host differences, not all its results are directly comparable, but they are relatively similar. Surprisingly, an Android foldable phone beats all the PCs, though these tests are CPU bound, and operations like file access may fare differently. Caveat: these results *will* change; retest with your Pythons and devices for current data.

Function Gotchas

Now that we've reached the end of the function story, let's review some common pitfalls. Functions have some jagged edges that you might not expect. They're all relatively obscure, and a few have started to fall away from the language completely in recent releases, but most have been known to trip up new users.

Local Names Are Detected Statically

As you've learned, Python classifies names assigned in a function as *locals* by default; they live in the function's scope and exist only while the function is running. What you may not realize is that Python detects locals statically, when it compiles the `def`'s code, rather than by noticing assignments as they happen at runtime. This leads to one of the most common oddities posted on the Python newsgroup by beginners.

Normally, a name that isn't assigned in a function is looked up in the enclosing module:

```
>>> X = 99

>>> def selector():        # X used but not assigned
        print(X)           # X found in global scope

>>> selector()
99
```

Here, the `X` in the function resolves to the `X` in the module. But watch what happens if you add an assignment to `X` after the reference:

```
>>> def selector():
        print(X)           # Does not yet exist!
        X = 88             # X classified as a local nam
                           # Can also happen for "import
>>> selector()
UnboundLocalError: cannot access local variable 'X' whe
```

You get the name-usage error shown here, but the reason is subtle. Python reads and compiles this code when it's typed interactively or imported from a module. While compiling, Python sees the assignment to `X` and decides that `X` will be a local name everywhere in the function. But when the function is actually run, because the assignment hasn't yet happened when the `print` executes, Python says you're using an undefined name. According to its name rules, it should say this; the local `X` is used before being assigned. In fact, any assignment in a function body makes a name local. Imports, `=`, nested `def`s, nested classes, and so on are all susceptible to this behavior.

The problem occurs because assigned names are treated as locals everywhere in a function, not just after the statements where they're assigned. Really, the previous example is ambiguous: was the intention to print the global `X` and create a local `X`, or is this a real programming error? Because Python treats `X` as a local everywhere, it's seen as an error; if you mean to print the global `X`, you need to declare it in a `global` statement:

```
>>> def selector():
        global X                # Force X to be global
        print(X)
        X = 88

>>> selector()
99
```

Remember, though, that this means the assignment also changes the global `X`, not a local `X`. Within a function, you can't use both local and global versions of the same simple name. If you really meant to print the global and then set a local of the same name, you'd need to import the enclosing module and use module attribute notation to get to the global version:

```
>>> X = 99
>>> def selector():
        import __main__         # Import enclosing modu
        print(__main__.X)       # Qualify to get to glc
        X = 88                  # Unqualified X classif
        print(X)                # Prints local version

>>> selector()
99
88
```

Qualification (the `.X` part) fetches a value from a namespace object. The interactive namespace is a module called `__main__`, so `__main__.X` reaches the global version of `X`. If that isn't clear, review Chapter 17.

In recent versions, Python has improved on this story somewhat by issuing for this case the more specific "unbound local" error message shown in the example listing (it used to simply raise a generic name error); this gotcha is still present in general, though.

Defaults and Mutable Objects

As noted briefly in Chapters 17 and 18, mutable values for default arguments can retain state between calls, though this is often unexpected. In general, default argument values are evaluated and saved *once* when a `def` statement is run, not each time the resulting function is later called. Internally, Python saves one object per default argument, attached to the function itself.

That's usually what you want—because defaults are evaluated at `def` time, they let you save values from the enclosing scope, if needed (functions defined within loops by factories may even depend on this behavior—see ahead). But because a default retains an object between calls, you have to be careful about changing mutable defaults. For instance, the following function uses an empty list as a default value, and then changes it in place each time the function is called:

```
>>> def saver(x=[]):                    # Saves away a list
        x.append(1)                      # Changes same objec
        print(x)

>>> saver([2])                          # Default not used
[2, 1]
>>> saver()                             # Default used
[1]
>>> saver()                             # Grows on each call
[1, 1]
>>> saver()
[1, 1, 1]
```

Some see this behavior as a feature—because mutable default arguments retain their state between function calls, they can serve some of the same roles as "static" local function variables in the C language. In a sense, they work much like global variables, but their names are local to the functions and so will not clash with names elsewhere in a program.

To other observers, though, this seems like a gotcha—especially the first time they run into it. There are better ways to retain state between calls in Python (e.g., using the nested scope closures and function attributes we met in this part, and the classes we will study in Part VI).

Moreover, mutable defaults can be overridden by real values, and are tricky to remember (and to understand at all). They depend upon the timing of default object construction. In the prior example, there is just one list object for the default value—the one created when the `def` is executed. You don't get a new list every time the function is called, so the list grows with each new append; it is not reset to empty each time.

If that's not the behavior you want, simply make a copy of the default at the start of the function body, or move the default value expression into the function body. As long as the value resides in code that's actually executed each time the function is *called*, you'll get a new object each time through:

```
>>> def saver(x=None):
        if x is None:                # No argument passed:
            x = []                   # Run code to make a
        x.append(1)                  # Changes new list ob
        print(x)

>>> saver([2])
[2, 1]
>>> saver()                          # Doesn't grow here
[1]
>>> saver()
[1]
```

By the way, the `if` statement in this example could *almost* be replaced by the assignment `x = x or []`, which takes advantage of the fact that Python's `or` returns one of its operand objects: if no argument was passed, `x` would default to `None`, so the `or` would return the new empty list on the right.

However, this isn't exactly the same. If an empty list were passed in, the `or` expression would cause the function to extend and return a newly created list, rather than extending and returning the passed-in list like the `if` version. (The expression becomes `[] or []`, which evaluates to the new empty list on the right; see Chapter 12's "Truth Values Revisited" if you don't recall why.) Real program requirements may call for either behavior.

Today, another way to achieve the value retention effect of mutable defaults in a possibly less confusing way is to use the *function attributes* we discussed in Chapter 19:

```
>>> def saver():
        saver.x.append(1)
        print(saver.x)

>>> saver.x = []
>>> saver()
[1]
>>> saver()
[1, 1]
>>> saver()
[1, 1, 1]
```

The function name is global to the function itself, but it need not be declared because it isn't changed directly within the function. This isn't used in exactly the same way, but when coded like this, the attachment of an object to the function is much more explicit (and arguably less magical).

Functions Without returns

In Python functions, `return` (and `yield`) statements are optional. When a function doesn't return a value explicitly, the function exits when control falls off the end of the function body. Technically, all functions return a value; if you don't provide a `return` statement, your function returns the `None` object automatically:

```
>>> def proc(x):
        print(x)                    # No return is a None

>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

Functions such as this without a `return` are Python's equivalent of what are called "procedures" in some languages. They're usually invoked as statements, and the `None` results are ignored, as they do their business without computing a useful result.

This is worth remembering, because Python won't tell you if you try to use the result of a function that doesn't return one. As we noted in Chapter 11, for

instance, assigning the result of a list `append` method won't raise an error, but you'll get back `None`, not the modified list:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)          # append is a "procedu
>>> print(list)                    # append changes list
None
```

"Common Coding Gotchas" discusses this more broadly. In general, any functions that do their business as a side effect are usually designed to be run as statements, not expressions.

Miscellaneous Function Gotchas

Here are two additional function-related gotchas—mostly reviews, but common enough to reiterate.

Enclosing scopes and loop variables

We met this gotcha in Chapter 17's discussion of enclosing function scopes, but as a reminder: when coding factory functions (a.k.a. closures), be careful about relying on enclosing function scope lookup for variables that are changed by enclosing loops. When a generated function is later called, all such references will remember the value of the *last* loop iteration in the enclosing function's scope. In this case, you must use *defaults* to save loop variable values instead of relying on automatic lookup in enclosing scopes. See "Loops Require Defaults, Not Scopes" in Chapter 17 for more details on this topic.

Hiding built-ins by assignment

Also in Chapter 17, we saw how it's possible to reassign built-in names in a closer local or global scope; the reassignment effectively hides ("shadows") and replaces that built-in's name for the remainder of the scope where the assignment occurs. This means you won't be able to use the original built-in value for the name. As long as you don't need the built-in value of the name you're assigning, this isn't an issue—many names are built in, and they may be freely reused. However, if you reassign a built-in name your code relies on, you may have problems. So don't do that, unless you really mean to. The good

news is that the built-ins you commonly use will soon become second nature, and Python's error trapping will alert you early in testing if your built-in name is not what you think it is.

Chapter Summary

This chapter rounded out our look at functions and built-in iteration tools with a larger case study that measured the performance of iteration alternatives and other tools we've met along the way, as well as one alternative Python implementation. We used both custom timer code as well as Python's `timeit` with a helper to time code in a variety of modes. We also reviewed common function-related mistakes to help you avoid pitfalls.

This concludes the functions part of this book. The next part expands on what we already know about *modules*—files of tools that form the topmost organizational unit in Python programs, and the structure in which functions reside. After that, we will explore *classes*, which are largely packages of functions with special first arguments. As you'll see, classes can implement objects that tap into the iteration protocol, just like the generators and iterables we used here. In fact, everything we have learned in this part of the book will apply when functions take the guise of class methods.

Before moving on to modules, though, be sure to work through this chapter's quiz, as well as the exercises for this part of the book, to practice what you've learned about functions here.

Test Your Knowledge: Quiz

1. What conclusions can you draw from this chapter about the relative speed of Python iteration tools?
2. What conclusions can you draw from this chapter about the relative speed of the Pythons timed?

Test Your Knowledge: Answers

1. In CPython today, list comprehensions are often the quickest of the bunch; `map` beats list comprehensions in Python when all tools must call built-in

functions; `for` loops tend to be slower than comprehensions; and generator functions and expressions are slower than all other options. Under PyPy, some of these findings differ; `map` often turns in a different relative performance, for example, and list comprehensions seem always quickest, perhaps due to function-level optimizations.
At least that's the case today on the Python versions tested, on the test machine used, and for the type of code timed—these results may vary if any of these three variables differ. Use the homegrown `timer` or standard library `timeit` to test your use cases for more relevant results. Also keep in mind that iteration is just one component of a program's time: more code gives a more complete picture.

2. In general, PyPy 7.3 (implementing Python 3.10) is substantially faster than CPython 3.12. In some cases timed, PyPy was 5X–20X faster than CPython, though in isolated cases (e.g., some string operations), PyPy was slower than CPython, and PyPy's use of a JIT can impact benchmark results.
At least that's the case today on the Python versions tested, on the test machine used, and for the type of code timed—these results may vary if any of these three variables differ. Use the homegrown `timer` or standard library `timeit` to test your use cases for more relevant results. This is especially true when timing Python implementations, which may be arbitrarily optimized in each new release—in fact, CPython may soon adopt a JIT like PyPy, which could invalidate results here. We also didn't test any of the many other Python implementations; see Chapter 2 for other options to time on your own.

Test Your Knowledge: Part IV Exercises

In these exercises, you're going to start coding more sophisticated programs. Be sure to check the solutions in "Part IV, Functions and Generators" in Appendix B, and be sure to start writing your code in module files. You won't want to retype these exercises in a REPL if you make a mistake.

1. *The basics*: At the Python interactive prompt, write a function named `echo` that prints its single argument to the screen and call it interactively, passing a variety of object types: string, integer, list, dictionary. Then, try calling it without passing any argument. What happens? What happens when you pass two arguments?

2. *Arguments*: Write a function called `adder` in a Python module file named *adder1.py*. The function should accept two arguments and return the sum (or concatenation) of the two. Then, add code at the bottom of the file to call the `adder` function with a variety of object types (two strings, two lists, two floating points), and run this file as a script from the system command line. Do you have to print the call statement results to see results on your screen?
3. *Arbitrary arguments*: Copy the file you wrote in the last exercise to *adder2.py*, generalize its `adder` function to compute the sum of an arbitrary number of arguments, and change the calls to pass more or fewer than two arguments. What type is the returned sum? (Hints: a slice such as `S[:0]` returns an empty sequence of the same type as `S`, and the `type` built-in function can test types; but see the manually coded `min` examples in Chapter 18 for a simpler approach.) What happens if you pass in arguments of different types? What about passing in dictionaries?
4. *Keywords*: In an *adder3.py*, change the `adder` function from exercise 2 to accept and sum/concatenate three arguments: `def adder(red, green, blue)`. Now, provide *default* values for each argument, and experiment with calling the function interactively or code tests in the file. Try passing one, two, three, and four arguments. Then, try passing *keyword* arguments. Does the call `adder(blue=1, red=2)` work? Why? Finally, copy and generalize the new `adder` to accept and sum/concatenate an *arbitrary* number of keyword arguments in an *adder4.py*. This is similar to what you did in exercise 3, but you'll need to iterate over a dictionary, not a tuple. (Hint: the `dict.keys` method returns an iterable you can step through with a `for` or `while`, but be sure to wrap it in a `list` call to index it; `dict.values` may help here too.)
5. *Dictionary tools*: Write a function called `copyDict(dict)` that copies its dictionary argument. It should return a new dictionary containing all the items in its argument. Use the dictionary `keys` method to iterate (or step over a dictionary's keys without calling `keys`). Copying sequences is easy (`X[:]` makes a top-level copy); does this work for dictionaries, too? As explained in this exercise's solution, because dictionaries come with similar tools, this and the next exercise are just coding exercises but still serve as representative functions.
6. *Dictionary tools*: Write a function called `addDict(dict1, dict2)` that computes the union (i.e., merge) of two dictionaries. It should return a new dictionary containing all the items in both its arguments (which are

assumed to be dictionaries). If the same key appears in both arguments, feel free to pick a value from either. Test your function by writing it in a file and running the file as a script. What happens if you pass lists instead of dictionaries? How could you generalize your function to handle this case, too? (Hint: see the `type` built-in function used earlier.) Does the order of the arguments passed in matter? Dictionary merge is also a built-in today (actually, several), but you're trying to stretch yourself a bit by coding it manually.

7. *More argument-matching examples*: First, define the following six functions (either interactively or in a module file that can be imported):

```
def f1(a, b): print(a, b)              # Normal args

def f2(a, *b): print(a, b)             # Positional co

def f3(a, **b): print(a, b)            # Keyword colle

def f4(a, *b, **c): print(a, b, c)     # Mixed modes

def f5(a, b=2, c=3): print(a, b, c)    # Defaults

def f6(a, b=2, *c): print(a, b, c)     # Defaults and
```

Now, test the following calls interactively, and try to explain each result; in some cases, you'll probably need to fall back on the matching rules covered in Chapter 18. Do you think mixing matching modes is a good idea in general? Can you think of cases where it would be useful?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, **dict(x=2, y=3))

>>> f5(1)
>>> f5(1, 4)
>>> f5(1, c=4)

>>> f6(1)
>>> f6(1, *[3, 4])
```

8. *Primes revisited*: Recall the following code snippet from Chapter 13, which simplistically determines whether a positive integer is prime:

```
x = num // 2                                # For some
while x > 1:
    if num % x == 0:                        # Remainde
        print(num, 'has factor', x)
        break                               # Exit now
    x -= 1
else:                                       # Normal e
    print(num, 'is prime')
```

Package this code as a reusable function in a module file (`num` should be a passed-in argument), and add some calls to the function at the bottom of your file to test. While you're at it, experiment with replacing the first line's `//` operator with `/` to see how *true* division breaks this code (refer back to Chapter 5 if you need a reminder). What can you do about negatives, and the values `0` and `1` ? How about speeding this up? Your outputs should look something like this:

```
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
```

9. *Iterations and comprehensions*: Write code to build a new list containing the square roots of all the numbers in this list: `[2, 4, 9, 16, 25]` . Code this as a `for` loop first, then as a `map` call, then as a list comprehension, and finally as a generator expression. Use the `sqrt` function in the built-in `math` module to do the calculation (i.e., import `math` and say `math.sqrt(X)`). Of the four, which approach do you like best?
10. *Timing tools*: In Chapter 5, we saw three ways to compute square roots: `math.sqrt(X)` , `X ** .5` , and `pow(X, .5)` . If your programs run a lot of these, their relative performance might become important. To see which is quickest, use the *timer2.py* module (Example 21-7) we wrote in this chapter to time each of these three tools. Use its `bestoftotal` function to test. Which of the three square root tools seems to run fastest on your device and Python in general? Finally, how might you use

timer2.py to interactively time the speed of dictionary comprehensions versus `for` loops? What about comprehensions with `if` clauses and nested `for` loops?

11. *Recursive functions*: Write a simple recursion function named `countdown` that prints numbers as it counts down to zero. For example, a call `countdown(5)` will print: `5 4 3 2 1 stop`. There's no obvious reason to code this with an explicit stack or queue, but what about a nonfunction approach? Would a generator make sense here?
12. *Computing factorials*: Finally, a computer-science classic (but demonstrative nonetheless). We employed the notion of factorials in Chapter 20's coverage of permutations: `N!`, computed as `N*(N-1)*(N-2)*...1`. For instance, `6!` is `6*5*4*3*2*1`, or `720`. Code and time four functions that, for a call `fact(N)`, each return `N!`. Code these four functions (1) as a recursive countdown per Chapter 19; (2) using the functional `reduce` call per Chapter 19; (3) with a simple iterative counter loop per Chapter 13; and (4) using the `math.factorial` library tool per Chapter 20. Use this chapter's `timeit` to time each of your functions. What conclusions can you draw from your results?