Things to Remember

- ✦ Python has built-in types and classes in modules that can represent practically every type of numerical value.
- ✦ The `Decimal` class is ideal for situations that require high precision and control over rounding behavior, such as computations of monetary values.
- ✦ Pass `str` instances to the `Decimal` constructor instead of `float` instances if it's important to compute exact answers and not floating point approximations.

Item 70: Profile Before Optimizing

The dynamic nature of Python causes surprising behaviors in its runtime performance. Operations you might assume would be slow are actually very fast (e.g., string manipulation, generators). Language features you might assume would be fast are actually very slow (e.g., attribute accesses, function calls). The true source of slowdowns in a Python program can be obscure.

The best approach is to ignore your intuition and directly measure the performance of a program before you try to optimize it. Python provides a built-in *profiler* for determining which parts of a program are responsible for its execution time. This means you can focus your optimization efforts on the biggest sources of trouble and ignore parts of the program that don't impact speed (i.e., follow Amdahl's law).

For example, say that I want to determine why an algorithm in a program is slow. Here, I define a function that sorts a `list` of data using an insertion sort:

```
def insertion_sort(data):
    result = []
    for value in data:
        insert_value(result, value)
    return result
```

The core mechanism of the insertion sort is the function that finds the insertion point for each piece of data. Here, I define an extremely inefficient version of the `insert_value` function that does a linear scan over the input array:

```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
```

```
            return
    array.append(value)
```

To profile insertion_sort and insert_value, I create a data set of random numbers and define a test function to pass to the profiler:

```
from random import randint

max_size = 10**4
data = [randint(0, max_size) for _ in range(max_size)]
test = lambda: insertion_sort(data)
```

Python provides two built-in profilers: one that is pure Python (profile) and another that is a C-extension module (cProfile). The cProfile built-in module is better because of its minimal impact on the performance of your program while it's being profiled. The pure-Python alternative imposes a high overhead that skews the results.

> **Note**
>
> When profiling a Python program, be sure that what you're measuring is the code itself and not external systems. Beware of functions that access the network or resources on disk. These may appear to have a large impact on your program's execution time because of the slowness of the underlying systems. If your program uses a cache to mask the latency of slow resources like these, you should ensure that it's properly warmed up before you start profiling.

Here, I instantiate a Profile object from the cProfile module and run the test function through it using the runcall method:

```
from cProfile import Profile

profiler = Profile()
profiler.runcall(test)
```

When the test function has finished running, I can extract statistics about its performance by using the pstats built-in module and its Stats class. Various methods on a Stats object adjust how to select and sort the profiling information to show only the things I care about:

```
from pstats import Stats

stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
```

The output is a table of information organized by function. The data sample is taken only from the time the profiler was active, during the `runcall` method above:

```
>>>
         20003 function calls in 1.320 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.320    1.320 main.py:35(<lambda>)
        1    0.003    0.003    1.320    1.320 main.py:10(insertion_sort)
    10000    1.306    0.000    1.317    0.000 main.py:20(insert_value)
     9992    0.011    0.000    0.011    0.000 {method 'insert' of 'list' objects}
        8    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
```

Here's a quick guide to what the profiler statistics columns mean:

- `ncalls`: The number of calls to the function during the profiling period.
- `tottime`: The number of seconds spent executing the function, excluding time spent executing other functions it calls.
- `tottime percall`: The average number of seconds spent in the function each time it is called, excluding time spent executing other functions it calls. This is `tottime` divided by `ncalls`.
- `cumtime`: The cumulative number of seconds spent executing the function, including time spent in all other functions it calls.
- `cumtime percall`: The average number of seconds spent in the function each time it is called, including time spent in all other functions it calls. This is `cumtime` divided by `ncalls`.

Looking at the profiler statistics table above, I can see that the biggest use of CPU in my test is the cumulative time spent in the `insert_value` function. Here, I redefine that function to use the `bisect` built-in module (see Item 72: "Consider Searching Sorted Sequences with `bisect`"):

```
from bisect import bisect_left

def insert_value(array, value):
    i = bisect_left(array, value)
    array.insert(i, value)
```

I can run the profiler again and generate a new table of profiler statistics. The new function is much faster, with a cumulative time spent that is nearly 100 times smaller than with the previous `insert_value` function:

```
>>>
         30003 function calls in 0.017 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.017    0.017 main.py:35(<lambda>)
        1    0.002    0.002    0.017    0.017 main.py:10(insertion_sort)
    10000    0.003    0.000    0.015    0.000 main.py:110(insert_value)
    10000    0.008    0.000    0.008    0.000 {method 'insert' of 'list' objects}
    10000    0.004    0.000    0.004    0.000 {built-in method _bisect.bisect_left}
```

Sometimes when you're profiling an entire program, you might find that a common utility function is responsible for the majority of execution time. The default output from the profiler makes such a situation difficult to understand because it doesn't show that the utility function is called by many different parts of your program.

For example, here the `my_utility` function is called repeatedly by two different functions in the program:

```python
def my_utility(a, b):
    c = 1
    for i in range(100):
        c += a * b

def first_func():
    for _ in range(1000):
        my_utility(4, 5)

def second_func():
    for _ in range(10):
        my_utility(1, 3)

def my_program():
    for _ in range(20):
        first_func()
        second_func()
```

Profiling this code and using the default `print_stats` output generates statistics that are confusing:

```
>>>
         20242 function calls in 0.118 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.118    0.118 main.py:176(my_program)
       20    0.003    0.000    0.117    0.006 main.py:168(first_func)
    20200    0.115    0.000    0.115    0.000 main.py:161(my_utility)
       20    0.000    0.000    0.001    0.000 main.py:172(second_func)
```

The `my_utility` function is clearly the source of most execution time, but it's not immediately obvious why that function is called so much. If you search through the program's code, you'll find multiple call sites for `my_utility` and still be confused.

To deal with this, the Python profiler provides the `print_callers` method to show which callers contributed to the profiling information of each function:

```
stats.print_callers()
```

This profiler statistics table shows functions called on the left and which function was responsible for making the call on the right. Here, it's clear that `my_utility` is most used by `first_func`:

```
>>>
   Ordered by: cumulative time

Function                                was called by...
                                            ncalls  tottime  cumtime
main.py:176(my_program)                 <-
main.py:168(first_func)                 <-      20    0.003    0.117  main.py:176(my_program)
main.py:161(my_utility)                 <-   20000    0.114    0.114  main.py:168(first_func)
                                               200    0.001    0.001  main.py:172(second_func)
Profiling.md:172(second_func)           <-      20    0.000    0.001  main.py:176(my_program)
```

Things to Remember

- ✦ It's important to profile Python programs before optimizing because the sources of slowdowns are often obscure.
- ✦ Use the `cProfile` module instead of the `profile` module because it provides more accurate profiling information.
- ✦ The `Profile` object's `runcall` method provides everything you need to profile a tree of function calls in isolation.
- ✦ The `Stats` object lets you select and print the subset of profiling information you need to see to understand your program's performance.

Item 71: Prefer `deque` for Producer–Consumer Queues

A common need in writing programs is a first-in, first-out (FIFO) queue, which is also known as a producer–consumer queue. A FIFO queue is used when one function gathers values to process and another function handles them in the order in which they were received. Often, programmers use Python's built-in `list` type as a FIFO queue.

For example, say that I have a program that's processing incoming emails for long-term archival, and it's using a `list` for a producer–consumer queue. Here, I define a class to represent the messages:

```
class Email:
    def __init__(self, sender, receiver, message):
        self.sender = sender
        self.receiver = receiver
        self.message = message
    ...
```

I also define a placeholder function for receiving a single email, presumably from a socket, the file system, or some other type of I/O system. The implementation of this function doesn't matter; what's important is its interface: It will either return an `Email` instance or raise a `NoEmailError` exception:

```
class NoEmailError(Exception):
    pass

def try_receive_email():
    # Returns an Email instance or raises NoEmailError
    ...
```

The producing function receives emails and enqueues them to be consumed at a later time. This function uses the `append` method on the `list` to add new messages to the end of the queue so they are processed after all messages that were previously received:

```
def produce_emails(queue):
    while True:
        try:
            email = try_receive_email()
        except NoEmailError:
            return
        else:
            queue.append(email)  # Producer
```

The consuming function does something useful with the emails. This function calls `pop(0)` on the queue, which removes the very first item from the `list` and returns it to the caller. By always processing items from the beginning of the queue, the consumer ensures that the items are processed in the order in which they were received:

```
def consume_one_email(queue):
    if not queue:
        return
    email = queue.pop(0)  # Consumer
```

```
    # Index the message for long-term archival
    ...
```

Finally, I need a looping function that connects the pieces together. This function alternates between producing and consuming until the `keep_running` function returns `False` (see Item 60: "Achieve Highly Concurrent I/O with Coroutines" on how to do this concurrently):

```
def loop(queue, keep_running):
    while keep_running():
        produce_emails(queue)
        consume_one_email(queue)

def my_end_func():
    ...

loop([], my_end_func)
```

Why not process each `Email` message in `produce_emails` as it's returned by `try_receive_email`? It comes down to the trade-off between latency and throughput. When using producer–consumer queues, you often want to minimize the latency of accepting new items so they can be collected as fast as possible. The consumer can then process through the backlog of items at a consistent pace—one item per loop in this case—which provides a stable performance profile and consistent throughput at the cost of end-to-end latency (see Item 55: "Use `Queue` to Coordinate Work Between Threads" for related best practices).

Using a `list` for a producer–consumer queue like this works fine up to a point, but as the *cardinality*—the number of items in the list—increases, the `list` type's performance can degrade superlinearly. To analyze the performance of using `list` as a FIFO queue, I can run some micro-benchmarks using the `timeit` built-in module. Here, I define a benchmark for the performance of adding new items to the queue using the `append` method of `list` (matching the producer function's usage):

```
import timeit

def print_results(count, tests):
    avg_iteration = sum(tests) / len(tests)
    print(f'Count {count:>5,} takes {avg_iteration:.6f}s')
    return count, avg_iteration

def list_append_benchmark(count):
    def run(queue):
```

```
        for i in range(count):
            queue.append(i)

    tests = timeit.repeat(
        setup='queue = []',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)
```

Running this benchmark function with different levels of cardinality lets me compare its performance in relationship to data size:

```
def print_delta(before, after):
    before_count, before_time = before
    after_count, after_time = after
    growth = 1 + (after_count - before_count) / before_count
    slowdown = 1 + (after_time - before_time) / before_time
    print(f'{growth:>4.1f}x data size, {slowdown:>4.1f}x time')

baseline = list_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_append_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
Count   500 takes 0.000039s

Count 1,000 takes 0.000073s
 2.0x data size,  1.9x time

Count 2,000 takes 0.000121s
 4.0x data size,  3.1x time

Count 3,000 takes 0.000172s
 6.0x data size,  4.5x time

Count 4,000 takes 0.000240s
 8.0x data size,  6.2x time

Count 5,000 takes 0.000304s
10.0x data size,  7.9x time
```

This shows that the append method takes roughly constant time for the list type, and the total time for enqueueing scales linearly as the data size increases. There is overhead for the list type to increase its capacity under the covers as new items are added, but it's reasonably low and is amortized across repeated calls to append.

Here, I define a similar benchmark for the pop(0) call that removes items from the beginning of the queue (matching the consumer function's usage):

```
def list_pop_benchmark(count):
    def prepare():
        return list(range(count))

    def run(queue):
        while queue:
            queue.pop(0)

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)
```

I can similarly run this benchmark for queues of different sizes to see how performance is affected by cardinality:

```
baseline = list_pop_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_pop_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count   500 takes 0.000050s

Count 1,000 takes 0.000133s
 2.0x data size,  2.7x time

Count 2,000 takes 0.000347s
 4.0x data size,  6.9x time

Count 3,000 takes 0.000663s
 6.0x data size, 13.2x time
```

```
Count 4,000 takes 0.000943s
 8.0x data size, 18.8x time

Count 5,000 takes 0.001481s
10.0x data size, 29.5x time
```

Surprisingly, this shows that the total time for dequeuing items from a `list` with `pop(0)` scales quadratically as the length of the queue increases. The cause is that `pop(0)` needs to move every item in the `list` back an index, effectively reassigning the entire list's contents. I need to call `pop(0)` for every item in the `list`, and thus I end up doing roughly `len(queue) * len(queue)` operations to consume the queue. This doesn't scale.

Python provides the `deque` class from the `collections` built-in module to solve this problem. `deque` is a *double-ended queue* implementation. It provides constant time operations for inserting or removing items from its beginning or end. This makes it ideal for FIFO queues.

To use the `deque` class, the call to `append` in `produce_emails` can stay the same as it was when using a `list` for the queue. The `list.pop` method call in `consume_one_email` must change to call the `deque.popleft` method with no arguments instead. And the `loop` method must be called with a `deque` instance instead of a `list`. Everything else stays the same. Here, I redefine the one function affected to use the new method and run `loop` again:

```python
import collections

def consume_one_email(queue):
    if not queue:
        return
    email = queue.popleft()  # Consumer
    # Process the email message
    ...

def my_end_func():
    ...

loop(collections.deque(), my_end_func)
```

I can run another version of the benchmark to verify that `append` performance (matching the producer function's usage) has stayed roughly the same (modulo a constant factor):

```python
def deque_append_benchmark(count):
    def prepare():
        return collections.deque()
```

```
    def run(queue):
        for i in range(count):
            queue.append(i)

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)
    return print_results(count, tests)

baseline = deque_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = deque_append_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count   500 takes 0.000029s

Count 1,000 takes 0.000059s
 2.0x data size,  2.1x time

Count 2,000 takes 0.000121s
 4.0x data size,  4.2x time

Count 3,000 takes 0.000171s
 6.0x data size,  6.0x time

Count 4,000 takes 0.000243s
 8.0x data size,  8.5x time

Count 5,000 takes 0.000295s
10.0x data size, 10.3x time
```

And I can benchmark the performance of calling popleft to mimic the consumer function's usage of deque:

```
def dequeue_popleft_benchmark(count):
    def prepare():
        return collections.deque(range(count))

    def run(queue):
        while queue:
            queue.popleft()

    tests = timeit.repeat(
```

```
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)

baseline = dequeue_popleft_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = dequeue_popleft_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count   500 takes 0.000024s

Count 1,000 takes 0.000050s
 2.0x data size,  2.1x time

Count 2,000 takes 0.000100s
 4.0x data size,  4.2x time

Count 3,000 takes 0.000152s
 6.0x data size,  6.3x time

Count 4,000 takes 0.000207s
 8.0x data size,  8.6x time

Count 5,000 takes 0.000265s
10.0x data size, 11.0x time
```

The popleft usage scales linearly instead of displaying the superlinear behavior of pop(0) that I measured before—hooray! If you know that the performance of a program critically depends on the speed of producer–consumer queues, then deque is a great choice. If you're not sure, then you should instrument your program to find out (see Item 70: "Profile Before Optimizing").

Things to Remember

- ✦ The list type can be used as a FIFO queue by having the producer call append to add items and the consumer call pop(0) to receive items. However, this may cause problems because the performance of pop(0) degrades superlinearly as the queue length increases.

- ✦ The deque class from the collections built-in module takes constant time—regardless of length—for append and popleft, making it ideal for FIFO queues.

Item 72: Consider Searching Sorted Sequences with bisect

It's common to find yourself with a large amount of data in memory as a sorted list that you then want to search. For example, you may have loaded an English language dictionary to use for spell checking, or perhaps a list of dated financial transactions to audit for correctness.

Regardless of the data your specific program needs to process, searching for a specific value in a list takes linear time proportional to the list's length when you call the index method:

```
data = list(range(10**5))
index = data.index(91234)
assert index == 91234
```

If you're not sure whether the exact value you're searching for is in the list, then you may want to search for the closest index that is equal to or exceeds your goal value. The simplest way to do this is to linearly scan the list and compare each item to your goal value:

```
def find_closest(sequence, goal):
    for index, value in enumerate(sequence):
        if goal < value:
            return index
    raise ValueError(f'{goal} is out of bounds')

index = find_closest(data, 91234.56)
assert index == 91235
```

Python's built-in bisect module provides better ways to accomplish these types of searches through ordered lists. You can use the bisect_left function to do an efficient binary search through any sequence of sorted items. The index it returns will either be where the item is already present in the list or where you'd want to insert the item in the list to keep it in sorted order:

```
from bisect import bisect_left

index = bisect_left(data, 91234)     # Exact match
assert index == 91234
```

```
index = bisect_left(data, 91234.56)  # Closest match
assert index == 91235
```

The complexity of the binary search algorithm used by the bisect module is logarithmic. This means searching in a list of length 1 million takes roughly the same amount of time with bisect as linearly searching a list of length 20 using the list.index method (math.log2(10**6) == 19.93...). It's way faster!

I can verify this speed improvement for the example from above by using the timeit built-in module to run a micro-benchmark:

```
import random
import timeit

size = 10**5
iterations = 1000

data = list(range(size))
to_lookup = [random.randint(0, size)
             for _ in range(iterations)]

def run_linear(data, to_lookup):
    for index in to_lookup:
        data.index(index)

def run_bisect(data, to_lookup):
    for index in to_lookup:
        bisect_left(data, index)

baseline = timeit.timeit(
    stmt='run_linear(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Linear search takes {baseline:.6f}s')

comparison = timeit.timeit(
    stmt='run_bisect(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Bisect search takes {comparison:.6f}s')

slowdown = 1 + ((baseline - comparison) / comparison)
print(f'{slowdown:.1f}x time')
```

```
>>>
Linear search takes 5.370117s
Bisect search takes 0.005220s
1028.7x time
```

The best part about bisect is that it's not limited to the list type; you can use it with any Python object that acts like a sequence (see Item 43: "Inherit from collections.abc for Custom Container Types" for how to do that). The module also provides additional features for more advanced situations (see help(bisect)).

Things to Remember

- ✦ Searching sorted data contained in a list takes linear time using the index method or a for loop with simple comparisons.
- ✦ The bisect built-in module's bisect_left function takes logarithmic time to search for values in sorted lists, which can be orders of magnitude faster than other approaches.

Item 73: Know How to Use heapq for Priority Queues

One of the limitations of Python's other queue implementations (see Item 71: "Prefer deque for Producer–Consumer Queues" and Item 55: "Use Queue to Coordinate Work Between Threads") is that they are first-in, first-out (FIFO) queues: Their contents are sorted by the order in which they were received. Often, you need a program to process items in order of relative importance instead. To accomplish this, a *priority queue* is the right tool for the job.

For example, say that I'm writing a program to manage books borrowed from a library. There are people constantly borrowing new books. There are people returning their borrowed books on time. And there are people who need to be reminded to return their overdue books. Here, I define a class to represent a book that's been borrowed:

```
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
```

I need a system that will send reminder messages when each book passes its due date. Unfortunately, I can't use a FIFO queue for this because the amount of time each book is allowed to be borrowed varies based on its recency, popularity, and other factors. For example, a book that is borrowed today may be due back later than a book that's borrowed tomorrow. Here, I achieve this behavior by using a standard list and sorting it by due_date each time a new Book is added:

```
def add_book(queue, book):
    queue.append(book)
    queue.sort(key=lambda x: x.due_date, reverse=True)

queue = []
add_book(queue, Book('Don Quixote', '2019-06-07'))
add_book(queue, Book('Frankenstein', '2019-06-05'))
add_book(queue, Book('Les Misérables', '2019-06-08'))
add_book(queue, Book('War and Peace', '2019-06-03'))
```

If I can assume that the queue of borrowed books is always in sorted order, then all I need to do to check for overdue books is to inspect the final element in the list. Here, I define a function to return the next overdue book, if any, and remove it from the queue:

```
class NoOverdueBooks(Exception):
    pass

def next_overdue_book(queue, now):
    if queue:
        book = queue[-1]
        if book.due_date < now:
            queue.pop()
            return book

    raise NoOverdueBooks
```

I can call this function repeatedly to get overdue books to remind people about in the order of most overdue to least overdue:

```
now = '2019-06-10'

found = next_overdue_book(queue, now)
print(found.title)

found = next_overdue_book(queue, now)
print(found.title)

>>>
War and Peace
Frankenstein
```

If a book is returned before the due date, I can remove the scheduled reminder message by removing the Book from the list:

```
def return_book(queue, book):
    queue.remove(book)

queue = []
book = Book('Treasure Island', '2019-06-04')

add_book(queue, book)
print('Before return:', [x.title for x in queue])

return_book(queue, book)
print('After return: ', [x.title for x in queue])

>>>
Before return: ['Treasure Island']
After return:  []
```

And I can confirm that when all books are returned, the return_book function will raise the right exception (see Item 20: "Prefer Raising Exceptions to Returning None"):

```
try:
    next_overdue_book(queue, now)
except NoOverdueBooks:
    pass          # Expected
else:
    assert False  # Doesn't happen
```

However, the computational complexity of this solution isn't ideal. Although checking for and removing an overdue book has a constant cost, every time I add a book, I pay the cost of sorting the whole list again. If I have len(queue) books to add, and the cost of sorting them is roughly len(queue) * math.log(len(queue)), the time it takes to add books will grow superlinearly (len(queue) * len(queue) * math.log(len(queue))).

Here, I define a micro-benchmark to measure this performance behavior experimentally by using the timeit built-in module (see Item 71: "Prefer deque for Producer–Consumer Queues" for the implementation of print_results and print_delta):

```
import random
import timeit

def print_results(count, tests):
    ...
```

```
def print_delta(before, after):
    ...

def list_overdue_benchmark(count):
    def prepare():
        to_add = list(range(count))
        random.shuffle(to_add)
        return [], to_add

    def run(queue, to_add):
        for i in to_add:
            queue.append(i)
            queue.sort(reverse=True)

        while queue:
            queue.pop()

    tests = timeit.repeat(
        setup='queue, to_add = prepare()',
        stmt=f'run(queue, to_add)',
        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)
```

I can verify that the runtime of adding and removing books from the queue scales superlinearly as the number of books being borrowed increases:

```
baseline = list_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_overdue_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count   500 takes 0.001138s

Count 1,000 takes 0.003317s
 2.0x data size,  2.9x time

Count 1,500 takes 0.007744s
 3.0x data size,  6.8x time

Count 2,000 takes 0.014739s
 4.0x data size, 13.0x time
```

When a book is returned before the due date, I need to do a linear scan in order to find the book in the queue and remove it. Removing a book causes all subsequent items in the `list` to be shifted back an index, which has a high cost that also scales superlinearly. Here, I define another micro-benchmark to test the performance of returning a book using this function:

```
def list_return_benchmark(count):
    def prepare():
        queue = list(range(count))
        random.shuffle(queue)

        to_return = list(range(count))
        random.shuffle(to_return)

        return queue, to_return

    def run(queue, to_return):
        for i in to_return:
            queue.remove(i)

    tests = timeit.repeat(
        setup='queue, to_return = prepare()',
        stmt=f'run(queue, to_return)',
        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)
```

And again, I can verify that indeed the performance degrades superlinearly as the number of books increases:

```
baseline = list_return_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_return_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count   500 takes 0.000898s

Count 1,000 takes 0.003331s
 2.0x data size,  3.7x time

Count 1,500 takes 0.007674s
 3.0x data size,  8.5x time
```

```
Count 2,000 takes 0.013721s
 4.0x data size, 15.3x time
```

Using the methods of `list` may work for a tiny library, but it certainly won't scale to the size of the Great Library of Alexandria, as I want it to!

Fortunately, Python has the built-in `heapq` module that solves this problem by implementing priority queues efficiently. A *heap* is a data structure that allows for a `list` of items to be maintained where the computational complexity of adding a new item or removing the smallest item has logarithmic computational complexity (i.e., even better than linear scaling). In this library example, smallest means the book with the earliest due date. The best part about this module is that you don't have to understand how heaps are implemented in order to use its functions correctly.

Here, I reimplement the `add_book` function using the `heapq` module. The queue is still a plain `list`. The `heappush` function replaces the `list.append` call from before. And I no longer have to call `list.sort` on the queue:

```
from heapq import heappush

def add_book(queue, book):
    heappush(queue, book)
```

If I try to use this with the `Book` class as previously defined, I get this somewhat cryptic error:

```
queue = []
add_book(queue, Book('Little Women', '2019-06-05'))
add_book(queue, Book('The Time Machine', '2019-05-30'))
```

```
>>>
Traceback ...
TypeError: '<' not supported between instances of 'Book' and
➥'Book'
```

The `heapq` module requires items in the priority queue to be comparable and have a natural sort order (see Item 14: "Sort by Complex Criteria Using the `key` Parameter" for details). You can quickly give the `Book` class this behavior by using the `total_ordering` class decorator from the `functools` built-in module (see Item 51: "Prefer Class Decorators Over Metaclasses for Composable Class Extensions" for background) and implementing the `__lt__` special method (see Item 43: "Inherit from `collections.abc` for Custom Container Types" for background). Here, I redefine the class with a less-than method that simply compares the `due_date` fields between two `Book` instances:

```
import functools

@functools.total_ordering
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date

    def __lt__(self, other):
        return self.due_date < other.due_date
```

Now, I can add books to the priority queue by using the `heapq.heappush` function without issues:

```
queue = []
add_book(queue, Book('Pride and Prejudice', '2019-06-01'))
add_book(queue, Book('The Time Machine', '2019-05-30'))
add_book(queue, Book('Crime and Punishment', '2019-06-06'))
add_book(queue, Book('Wuthering Heights', '2019-06-12'))
```

Alternatively, I can create a `list` with all of the books in any order and then use the `sort` method of `list` to produce the heap:

```
queue = [
    Book('Pride and Prejudice', '2019-06-01'),
    Book('The Time Machine', '2019-05-30'),
    Book('Crime and Punishment', '2019-06-06'),
    Book('Wuthering Heights', '2019-06-12'),
]
queue.sort()
```

Or I can use the `heapq.heapify` function to create a heap in linear time (as opposed to the `sort` method's `len(queue) * log(len(queue))` complexity):

```
from heapq import heapify

queue = [
    Book('Pride and Prejudice', '2019-06-01'),
    Book('The Time Machine', '2019-05-30'),
    Book('Crime and Punishment', '2019-06-06'),
    Book('Wuthering Heights', '2019-06-12'),
]
heapify(queue)
```

To check for overdue books, I inspect the first element in the list instead of the last, and then I use the heapq.heappop function instead of the list.pop function:

```
from heapq import heappop

def next_overdue_book(queue, now):
    if queue:
        book = queue[0]           # Most overdue first
        if book.due_date < now:
            heappop(queue)        # Remove the overdue book
            return book

    raise NoOverdueBooks
```

Now, I can find and remove overdue books in order until there are none left for the current time:

```
now = '2019-06-02'

book = next_overdue_book(queue, now)
print(book.title)

book = next_overdue_book(queue, now)
print(book.title)

try:
    next_overdue_book(queue, now)
except NoOverdueBooks:
    pass          # Expected
else:
    assert False  # Doesn't happen

>>>
The Time Machine
Pride and Prejudice
```

I can write another micro-benchmark to test the performance of this implementation that uses the heapq module:

```
def heap_overdue_benchmark(count):
    def prepare():
        to_add = list(range(count))
        random.shuffle(to_add)
        return [], to_add

    def run(queue, to_add):
        for i in to_add:
```

```
            heappush(queue, i)
        while queue:
            heappop(queue)

    tests = timeit.repeat(
        setup='queue, to_add = prepare()',
        stmt=f'run(queue, to_add)',
        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)
```

This benchmark experimentally verifies that the heap-based priority queue implementation scales much better (roughly `len(queue) * math.log(len(queue))`), without superlinearly degrading performance:

```
baseline = heap_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = heap_overdue_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count   500 takes 0.000150s

Count 1,000 takes 0.000325s
 2.0x data size,  2.2x time

Count 1,500 takes 0.000528s
 3.0x data size,  3.5x time

Count 2,000 takes 0.000658s
 4.0x data size,  4.4x time
```

With the `heapq` implementation, one question remains: How should I handle returns that are on time? The solution is to never remove a book from the priority queue until its due date. At that time, it will be the first item in the `list`, and I can simply ignore the book if it's already been returned. Here, I implement this behavior by adding a new field to track the book's return status:

```
@functools.total_ordering
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
```

```
        self.returned = False  # New field

    ...
```

Then, I change the next_overdue_book function to repeatedly ignore any book that's already been returned:

```
def next_overdue_book(queue, now):
    while queue:
        book = queue[0]
        if book.returned:
            heappop(queue)
            continue

        if book.due_date < now:
            heappop(queue)
            return book

        break

    raise NoOverdueBooks
```

This approach makes the return_book function extremely fast because it makes no modifications to the priority queue:

```
def return_book(queue, book):
    book.returned = True
```

The downside of this solution for returns is that the priority queue may grow to the maximum size it would have needed if all books from the library were checked out and went overdue. Although the queue operations will be fast thanks to heapq, this storage overhead may take significant memory (see Item 81: "Use tracemalloc to Understand Memory Usage and Leaks" for how to debug such usage).

That said, if you're trying to build a robust system, you need to plan for the worst-case scenario; thus, you should expect that it's possible for every library book to go overdue for some reason (e.g., a natural disaster closes the road to the library). This memory cost is a design consideration that you should have already planned for and mitigated through additional constraints (e.g., imposing a maximum number of simultaneously lent books).

Beyond the priority queue primitives that I've used in this example, the heapq module provides additional functionality for advanced use cases (see help(heapq)). The module is a great choice when its functionality matches the problem you're facing (see the queue.PriorityQueue class for another thread-safe option).

Things to Remember

- ✦ Priority queues allow you to process items in order of importance instead of in first-in, first-out order.
- ✦ If you try to use `list` operations to implement a priority queue, your program's performance will degrade superlinearly as the queue grows.
- ✦ The `heapq` built-in module provides all of the functions you need to implement a priority queue that scales efficiently.
- ✦ To use `heapq`, the items being prioritized must have a natural sort order, which requires special methods like `__lt__` to be defined for classes.

Item 74: Consider `memoryview` and `bytearray` for Zero-Copy Interactions with `bytes`

Although Python isn't able to parallelize CPU-bound computation without extra effort (see Item 64: "Consider `concurrent.futures` for True Parallelism"), it is able to support high-throughput, parallel I/O in a variety of ways (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism" and Item 60: "Achieve Highly Concurrent I/O with Coroutines"). That said, it's surprisingly easy to use these I/O tools the wrong way and reach the conclusion that the language is too slow for even I/O-bound workloads.

For example, say that I'm building a media server to stream television or movies over a network to users so they can watch without having to download the video data in advance. One of the key features of such a system is the ability for users to move forward or backward in the video playback so they can skip or repeat parts. In the client program, I can implement this by requesting a chunk of data from the server corresponding to the new time index selected by the user:

```
def timecode_to_index(video_id, timecode):
    ...
    # Returns the byte offset in the video data

def request_chunk(video_id, byte_offset, size):
    ...
    # Returns size bytes of video_id's data from the offset

video_id = ...
timecode = '01:09:14:28'
byte_offset = timecode_to_index(video_id, timecode)
```

```
size = 20 * 1024 * 1024
video_data = request_chunk(video_id, byte_offset, size)
```

How would you implement the server-side handler that receives the `request_chunk` request and returns the corresponding 20 MB chunk of video data? For the sake of this example, I assume that the command and control parts of the server have already been hooked up (see Item 61: “Know How to Port Threaded I/O to `asyncio`” for what that requires). I focus here on the last steps where the requested chunk is extracted from gigabytes of video data that’s cached in memory and is then sent over a socket back to the client. Here’s what the implementation would look like:

```
socket = ...                # socket connection to client
video_data = ...            # bytes containing data for video_id
byte_offset = ...           # Requested starting position
size = 20 * 1024 * 1024     # Requested chunk size

chunk = video_data[byte_offset:byte_offset + size]
socket.send(chunk)
```

The latency and throughput of this code will come down to two factors: how much time it takes to slice the 20 MB video chunk from `video_data`, and how much time the socket takes to transmit that data to the client. If I assume that the socket is infinitely fast, I can run a micro-benchmark by using the `timeit` built-in module to understand the performance characteristics of slicing `bytes` instances this way to create chunks (see Item 11: “Know How to Slice Sequences” for background):

```
import timeit

def run_test():
    chunk = video_data[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')
```

```
>>>
0.004925669 seconds
```

It took roughly 5 milliseconds to extract the 20 MB slice of data to transmit to the client. That means the overall throughput of my server is limited to a theoretical maximum of 20 MB / 5 milliseconds = 7.3 GB / second, since that's the fastest I can extract the video data from memory. My server will also be limited to 1 CPU-second / 5 milliseconds = 200 clients requesting new chunks in parallel, which is tiny compared to the tens of thousands of simultaneous connections that tools like the `asyncio` built-in module can support. The problem is that slicing a `bytes` instance causes the underlying data to be copied, which takes CPU time.

A better way to write this code is by using Python's built-in `memoryview` type, which exposes CPython's high-performance *buffer protocol* to programs. The buffer protocol is a low-level C API that allows the Python runtime and C extensions to access the underlying data buffers that are behind objects like `bytes` instances. The best part about `memoryview` instances is that slicing them results in another `memoryview` instance without copying the underlying data. Here, I create a `memoryview` wrapping a `bytes` instance and inspect a slice of it:

```
data = b'shave and a haircut, two bits'
view = memoryview(data)
chunk = view[12:19]
print(chunk)
print('Size:           ', chunk.nbytes)
print('Data in view:   ', chunk.tobytes())
print('Underlying data:', chunk.obj)

>>>
<memory at 0x10951fb80>
Size:            7
Data in view:    b'haircut'
Underlying data: b'shave and a haircut, two bits'
```

By enabling *zero-copy* operations, `memoryview` can provide enormous speedups for code that needs to quickly process large amounts of memory, such as numerical C extensions like NumPy and I/O-bound programs like this one. Here, I replace the simple `bytes` slicing from above with `memoryview` slicing instead and repeat the same micro-benchmark:

```
video_view = memoryview(video_data)

def run_test():
    chunk = video_view[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark
```

```
result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')
```

```
>>>
0.000000250 seconds
```

The result is 250 nanoseconds. Now the theoretical maximum throughput of my server is 20 MB / 250 nanoseconds = 164 TB / second. For parallel clients, I can theoretically support up to 1 CPU-second / 250 nanoseconds = 4 million. That's more like it! This means that now my program is entirely bound by the underlying performance of the socket connection to the client, not by CPU constraints.

Now, imagine that the data must flow in the other direction, where some clients are sending live video streams to the server in order to broadcast them to other users. In order to do this, I need to store the latest video data from the user in a cache that other clients can read from. Here's what the implementation of reading 1 MB of new data from the incoming client would look like:

```
socket = ...          # socket connection to the client
video_cache = ...     # Cache of incoming video stream
byte_offset = ...     # Incoming buffer position
size = 1024 * 1024    # Incoming chunk size

chunk = socket.recv(size)
video_view = memoryview(video_cache)
before = video_view[:byte_offset]
after = video_view[byte_offset + size:]
new_cache = b''.join([before, chunk, after])
```

The socket.recv method returns a bytes instance. I can splice the new data with the existing cache at the current byte_offset by using simple slicing operations and the bytes.join method. To understand the performance of this, I can run another micro-benchmark. I'm using a dummy socket, so the performance test is only for the memory operations, not the I/O interaction:

```
def run_test():
    chunk = socket.recv(size)
    before = video_view[:byte_offset]
    after = video_view[byte_offset + size:]
    new_cache = b''.join([before, chunk, after])
```

```
result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.033520550 seconds
```

It takes 33 milliseconds to receive 1 MB and update the video cache. This means my maximum receive throughput is 1 MB / 33 milliseconds = 31 MB / second, and I'm limited to 31 MB / 1 MB = 31 simultaneous clients streaming in video data this way. This doesn't scale.

A better way to write this code is to use Python's built-in `bytearray` type in conjunction with `memoryview`. One limitation with `bytes` instances is that they are read-only and don't allow for individual indexes to be updated:

```
my_bytes = b'hello'
my_bytes[0] = b'\x79'

>>>
Traceback ...
TypeError: 'bytes' object does not support item assignment
```

The `bytearray` type is like a mutable version of `bytes` that allows for arbitrary positions to be overwritten. `bytearray` uses integers for its values instead of `bytes`:

```
my_array = bytearray(b'hello')
my_array[0] = 0x79
print(my_array)

>>>
bytearray(b'yello')
```

A `memoryview` can also be used to wrap a `bytearray`. When you slice such a `memoryview`, the resulting object can be used to assign data to a particular portion of the underlying buffer. This eliminates the copying costs from above that were required to splice the `bytes` instances back together after data was received from the client:

```
my_array = bytearray(b'row, row, row your boat')
my_view = memoryview(my_array)
write_view = my_view[3:13]
write_view[:] = b'-10 bytes-'
print(my_array)
```

```
>>>
bytearray(b'row-10 bytes- your boat')
```

Many library methods in Python, such as `socket.recv_into` and `RawIOBase.readinto`, use the buffer protocol to receive or read data quickly. The benefit of these methods is that they avoid allocating memory and creating another copy of the data; what's received goes straight into an existing buffer. Here, I use `socket.recv_into` along with a `memoryview` slice to receive data into an underlying `bytearray` without the need for splicing:

```
video_array = bytearray(video_cache)
write_view = memoryview(video_array)
chunk = write_view[byte_offset:byte_offset + size]
socket.recv_into(chunk)
```

I can run another micro-benchmark to compare the performance of this approach to the earlier example that used `socket.recv`:

```
def run_test():
    chunk = write_view[byte_offset:byte_offset + size]
    socket.recv_into(chunk)

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')
```

```
>>>
0.000033925 seconds
```

It took 33 microseconds to receive a 1 MB video transmission. This means my server can support 1 MB / 33 microseconds = 31 GB / second of max throughput, and 31 GB / 1 MB = 31,000 parallel streaming clients. That's the type of scalability that I'm looking for!

Things to Remember

- ✦ The `memoryview` built-in type provides a zero-copy interface for reading and writing slices of objects that support Python's high-performance buffer protocol.
- ✦ The `bytearray` built-in type provides a mutable `bytes`-like type that can be used for zero-copy data reads with functions like `socket.recv_from`.
- ✦ A `memoryview` can wrap a `bytearray`, allowing for received data to be spliced into an arbitrary buffer location without copying costs.

This page intentionally left blank

9 Testing and Debugging

Python doesn't have compile-time static type checking. There's nothing in the interpreter that will ensure that your program will work correctly when you run it. Python does support optional type annotations that can be used in static analysis to detect many kinds of bugs (see Item 90: "Consider Static Analysis via `typing` to Obviate Bugs" for details). However, it's still fundamentally a dynamic language, and anything is possible. With Python, you ultimately don't know if the functions your program calls will be defined at runtime, even when their existence is evident in the source code. This dynamic behavior is both a blessing and a curse.

The large numbers of Python programmers out there say it's worth going without compile-time static type checking because of the productivity gained from the resulting brevity and simplicity. But most people using Python have at least one horror story about a program encountering a boneheaded error at runtime. One of the worst examples I've heard of involved a `SyntaxError` being raised in production as a side effect of a dynamic import (see Item 88: "Know How to Break Circular Dependencies"), resulting in a crashed server process. The programmer I know who was hit by this surprising occurrence has since ruled out using Python ever again.

But I have to wonder, why wasn't the code more well tested before the program was deployed to production? Compile-time static type safety isn't everything. You should always test your code, regardless of what language it's written in. However, I'll admit that in Python it may be more important to write tests to verify correctness than in other languages. Luckily, the same dynamic features that create risks also make it extremely easy to write tests for your code and to debug malfunctioning programs. You can use Python's dynamic nature and easily overridable behaviors to implement tests and ensure that your programs work as expected.