Community Experience Distilled

Measure, optimize, and improve the performance of your Python code with this easy-to-follow guide

Mastering Python High Performance

Fernando Doglio

Every bit of memory and every CPU cycle counts, and knowing how to squash every bit of processing power out of your code is a crucial and sought-after skill. Nowadays, Python is used in many scientific projects, and the calculations done in these projects require serious fine-tuning. Profilers are tools designed to help you measure the performance of your code and help you during the optimization process, so knowing how to use them and read their output is very handy.

This book starts from the basics and advances on to increasingly complex topics, such as profiling, writing a real-life application, and applying a full set of tools designed to improve your application. Learn about the major profilers and graphic tools to help you make sense of their output. See generic optimization techniques and Python-specific optimizations and go through the main constructs of the language that will help you improve quickly with little change to the code. Finally, the book covers some tools for number crunching and how to use them properly to get the best results.

Who this book is written for
If you're a Python developer interested in improving the speed of your scripts or simply wanting to take your skills to the next level, then this book is perfect for you.

What you will learn from this book
- Master code optimization step by step and learn how to use different tools
- Understand what a profiler is and how to read its output
- Interpret visual output from profiling tools and improve the performance of your script
- Use Cython to create fast applications using Python and C
- Take advantage of PyPy to improve the performance of your Python code
- Optimize number-crunching code with Numba, Parakeet, and pandas

If you're a Python developer interested in improving the speed of your scripts or simply wanting to take your skills to the next level, then this book is perfect for you.

Who this book is written for
If you're a Python developer interested in improving the speed of your scripts or simply wanting to take your skills to the next level, then this book is perfect for you.

What you will learn from this book
- Master code optimization step by step and learn how to use different tools
- Understand what a profiler is and how to read its output
- Interpret visual output from profiling tools and improve the performance of your script
- Use Cython to create fast applications using Python and C
- Take advantage of PyPy to improve the performance of your Python code
- Optimize number-crunching code with Numba, Parakeet, and pandas
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Profiling 101'
- A synopsis of the book’s content
- More information on Mastering Python High Performance
About the Author

**Fernando Doglio** has been working as a web developer for the past 10 years.

During that time, he shifted his focus to the Web and grabbed the opportunity of working with most of the leading technologies, such as PHP, Ruby on Rails, MySQL, Python, Node.js, AngularJS, AJAX, REST APIs, and so on.

In his spare time, Fernando likes to tinker and learn new things. This is why his GitHub account keeps getting new repos every month. He's also a big open source supporter and tries to win the support of new people with the help of his website, [lookingforpullrequests.com](http://lookingforpullrequests.com).

You can reach him on Twitter at [@deleteman123](https://twitter.com/deleteman123).

When he is not programming, he spends time with his family.
Preface

The idea of this book came to me from the nice people at Packt Publishing. They wanted someone who could delve into the intricacies of high performance in Python and everything related to this subject, be it profiling, the available tools (such as profilers and other performance enhancement techniques), or even alternatives to the standard Python implementation.

Having said that, I welcome you to Mastering Python High Performance. In this book, we'll cover everything related to performance improvements. Knowledge about the subject is not strictly required (although it won't hurt), but knowledge of the Python programming language is required, especially in some of the Python-specific chapters.

We'll start by going through the basics of what profiling is, how it fits into the development cycle, and the benefits related to including this practice in it. Afterwards, we'll move on to the core tools required to get the job done (profilers and visual profilers). Then, we will take a look at a set of optimization techniques and finally arrive at a fully practical chapter that will provide a real-life optimization example.

What this book covers

Chapter 1, Profiling 101, provides information about the art of profiling to those who are not aware of it.

Chapter 2, The Profilers, tells you how to use the core tools that will be mentioned throughout the book.

Chapter 3, Going Visual – GUIs to Help Understand Profiler Output, covers how to use the pyprof2calltree and RunSnakeRun tools. It also helps the developer to understand the output of cProfile with different visualization techniques.
Chapter 4, *Optimize Everything*, talks about the basic process of optimization and a set of good/recommended practices that every Python developer should follow before considering other options.

Chapter 5, *Multithreading versus Multiprocessing*, discusses multithreading and multiprocessing and explains how and when to apply them.

Chapter 6, *Generic Optimization Options*, describes and shows you how to install and use Cython and PyPy in order to improve code performance.

Chapter 7, *Lightning Fast Number Crunching with Numba, Parakeet, and pandas*, talks about tools that help optimize Python scripts that deal with numbers. These specific tools (Numba, Parakeet, and pandas) help make number crunching faster.

Chapter 8, *Putting It All into Practice*, provides a practical example of profilers, finds its bottlenecks, and removes them using the tools and techniques mentioned in this book. To conclude, we'll compare the results of using each technique.
Just like any infant needs to learn how to crawl before running 100 mts with obstacles in under 12 seconds, programmers need to understand the basics of profiling before trying to master that art. So, before we start delving into the mysteries of performance optimization and profiling on Python programs, we need to have a clear understanding of the basics.

Once you know the basics, you'll be able to learn about the tools and techniques. So, to start us off, this chapter will cover everything you need to know about profiling but were too afraid to ask. In this chapter we will do the following things:

• We will provide a clear definition of what profiling is and the different profiling techniques.
• We will explain the importance of profiling in the development cycle, because profiling is not something you do only once and then forget about it. Profiling should be an integral part of the development process, just like writing tests is.
• We will cover things we can profile. We'll go over the different types of resources we'll be able to measure and how they'll help us find our problems.
• We will discuss the risk of premature optimization, that is, why optimizing before profiling is generally a bad idea.
• You will learn about running time complexity. Understanding profiling techniques is one step into successful optimization, but we also need to understand how to measure the complexity of an algorithm in order to understand whether we need to improve it or not.
• We will also look at good practices. Finally, we'll go over some good practices to keep in mind when starting the profiling process of your project.
What is profiling?

A program that hasn't been optimized will normally spend most of its CPU cycles in some particular subroutines. Profiling is the analysis of how the code behaves in relation to the resources it's using. For instance, profiling will tell you how much CPU time an instruction is using or how much memory the full program is consuming. It is achieved by modifying either the source code of the program or the binary executable form (when possible) to use something called as a profiler.

Normally, developers profile their programs when they need to either optimize their performance or when those programs are suffering from some kind of weird bug, which can normally be associated with memory leaks. In such cases, profiling can help them get an in-depth understanding of how their code is using the computer's resources (that is, how many times a certain function is being called).

A developer can use this information, along with a working knowledge of the source code, to find the program's bottlenecks and memory leaks. The developer can then fix whatever is wrong with the code.

There are two main methodologies for profiling software: event-based profiling and statistical profiling. When using these types of software, you should keep in mind that they both have pros and cons.

Event-based profiling

Not every programming language supports this type of profiling. Here are some programming languages that support event-based profiling:

- **Java**: The JVM (JVM Tools Interface) provides hooks for profilers to trap events such as calls, thread-related events, class loads and so on.
- **.NET**: Just like with Java, the runtime provides events (http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing/Profiling#Methods_of_data_gathering)
- **Python**: Using the `sys.setprofile` function, a developer can trap events such as `python_[call|return|exception]` or `c_[call|return|exception]`
Event-based profilers (also known as tracing profilers) work by gathering data on specific events during the execution of our program. These profilers generate a large amount of data. Basically, the more events they listen to, the more data they will gather. This makes them somewhat impractical to use, and they are not the first choice when starting to profile a program. However, they are a good last resort when other profiling methods aren't enough or just aren't specific enough. Consider the case where you'd want to profile all the return statements. This type of profiler would give you the granularity you'd need for this task, while others would simply not allow you to execute this task.

A simple example of an event-based profiler on Python could be the following code (we'll understand this topic better once we reach the upcoming chapters):

```python
import sys

def profiler(frame, event, arg):
    print 'PROFILER: %r %r' % (event, arg)

sys.setprofile(profiler)

# simple (and very inefficient) example of how to calculate the Fibonacci sequence for a number.
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n-1))
        seq.append(fib(n))
    return seq

print fib_seq(2)
```
The preceding code contributes to the following output:

```
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'return' 0
PROFILER: 'c_call' <built-in method append of list object at 0x7f570ca215f0>
PROFILER: 'c_return' <built-in method append of list object at 0x7f570ca215f0>
PROFILER: 'return' [0]
PROFILER: 'c_call' <built-in method extend of list object at 0x7f570ca21bd8>
PROFILER: 'c_return' <built-in method extend of list object at 0x7f570ca21bd8>
PROFILER: 'call' None
PROFILER: 'return' 1
PROFILER: 'c_call' <built-in method append of list object at 0x7f570ca21bd8>
PROFILER: 'c_return' <built-in method append of list object at 0x7f570ca21bd8>
PROFILER: 'return' [0, 1]
PROFILER: 'c_call' <built-in method extend of list object at 0x7f570ca55bd8>
PROFILER: 'c_return' <built-in method extend of list object at 0x7f570ca55bd8>
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'return' 1
PROFILER: 'call' None
PROFILER: 'return' 0
PROFILER: 'return' 1
PROFILER: 'c_call' <built-in method append of list object at 0x7f570ca55bd8>
PROFILER: 'c_return' <built-in method append of list object at 0x7f570ca55bd8>
PROFILER: 'return' [0, 1, 1]
[0, 1, 1]
PROFILER: 'return' None
PROFILER: 'call' None
PROFILER: 'c_call' <built-in method discard of set object at 0x7f570ca8a960>
PROFILER: 'c_return' <built-in method discard of set object at 0x7f570ca8a960>
PROFILER: 'return' None
```
As you can see, PROFILER is called on every event. We can print/gather the information we deem relevant inside the PROFILER function. The last line on the sample code shows that the simple execution of `fib_seq(2)` generates a lot of output data. If we were dealing with a real-world program, this output would be several orders of magnitude bigger. This is why event-based profiling is normally the last option when it comes to profiling. There are other alternatives out there (as we’ll see) that generate much less output, but, of course, have a lower accuracy rate.

### Statistical profiling

Statistical profilers work by sampling the program counter at regular intervals. This in turn allows the developer to get an idea of how much time the target program is spending on each function. Since it works by sampling the PC, the resulting numbers will be a statistical approximation of reality instead of exact numbers. Still, it should be enough to get a glimpse of what the profiled program is doing and where the bottlenecks are.

Some advantages of this type of profiling are as follows:

- **Less data to analyze**: Since we’re only sampling the program’s execution instead of saving every little piece of data, the amount of information to analyze will be significantly smaller.

- **Smaller profiling footprint**: Due to the way the sampling is made (using OS interrupts), the target program suffers a smaller hit on its performance. Although the presence of the profiler is not 100 percent unnoticed, statistical profiling does less damage than the event-based one.

Here is an example of the output of **OProfile** ([http://oprofile.sourceforge.net/news/](http://oprofile.sourceforge.net/news/)), a Linux statistical profiler:

```
Function name,File name,Times Encountered,Percentage
"func80000","statistical_profiling.c",30760,48.96%
"func40000","statistical_profiling.c",17515,27.88%
"func20000","static_functions.c",7141,11.37%
"func10000","static_functions.c",3572,5.69%
"func5000","static_functions.c",1787,2.84%
"func2000","static_functions.c",768,1.22%
```
Here is the output of profiling the same Fibonacci code from the preceding code using a statistical profiler for Python called statprof:

```
%   cumulative      self
time    seconds   seconds  name
100.00      0.01      0.01  B02088_01_03.py:11:fib
 0.00      0.01      0.00  B02088_01_03.py:17:fib_seq
 0.00      0.01      0.00  B02088_01_03.py:21:<module>
---
Sample count: 1
Total time: 0.010000 seconds
```

As you can see, there is quite a difference between the output of both profilers for the same code.

**The importance of profiling**

Now that we know what profiling means, it is also important to understand how important and relevant it is to actually do it during the development cycle of our applications.

Profiling is not something everyone is used to do, especially with non-critical software (unlike peace maker embedded software or any other type of execution-critical example). Profiling takes time and is normally useful only after we've detected that something is wrong with our program. However, it could still be performed before that even happens to catch possible unseen bugs, which would, in turn, help chip away the time spent debugging the application at a later stage.

As hardware keeps advancing, getting faster and cheaper, it is increasingly hard to understand why we, as developers, should spend resources (mainly time) on profiling our creations. After all, we have practices such as test-driven development, code review, pair programming and others that assure us our code is solid and that it'll work as we want it. Right?
However, what we sometimes fail to realize is that the higher level our languages become (we've gone from assembler to JavaScript in just a few years), the less we think about CPU cycles, memory allocation, CPU registries, and so on. New generations of programmers learn their craft using higher level languages because they're easier to understand and provide more power out of the box. However, they also abstract the hardware and our interaction with it. As this tendency keeps growing, the chances that new developers will even consider profiling their software as another step on its development grows weaker by the second.

Let's look at the following scenario:

As we know, profiling measures the resources our program uses. As I've stated earlier, they keep getting cheaper and cheaper. So, the cost of getting our software out and the cost of making it available to a higher number of users is also getting cheaper.

These days, it is increasingly easy to create and publish an application that will be reached by thousands of people. If they like it and spread the word through social media, that number can blow up exponentially. Once that happens, something that is very common is that the software will crash, or it'll become impossibly slow and the users will just go away.

A possible explanation for the preceding scenario is, of course, a badly thought and non-scalable architecture. After all, one single server with a limited amount of RAM and processing power will get you so far until it becomes your bottleneck. However, another possible explanation, one that proves to be true many times, is that we failed to stress test our application. We didn't think about resource consumption; we just made sure our tests passed, and we were happy with that. In other words, we failed to go that extra mile, and as a result, our project crashed and burned.

Profiling can help avoid that crash and burn outcome, since it provides a fairly accurate view of what our program is doing, no matter the load. So, if we profile it with a very light load, and the result is that we're spending 80 percent of our time doing some kind of I/O operation, it might raise a flag for us. Even if, during our test, the application performed correctly, it might not do so under heavy stress. Think of a memory leak-type scenario. In those cases, small tests might not generate a big enough problem for us to detect it. However, a production deployment under heavy stress will. Profiling can provide enough evidence for us to detect this problem before it even turns into one.
What can we profile?
Going deeper into profiling, it is very important to understand what we can actually profile. Measuring is the core of profiling, so let's take a detailed look at the things we can measure during a program's execution.

Execution time
The most basic of the numbers we can gather when profiling is the execution time. The execution time of the entire process or just of a particular portion of the code will shed some light on its own. If you have experience in the area your program is running (that is, you're a web developer and you're working on a web framework), you probably already know what it means for your system to take too much time. For instance, a simple web server might take up to 100 milliseconds when querying the database, rendering the response, and sending it back to the client. However, if the same piece of code starts to slow down and now it takes 60 seconds to do the same task, then you should start thinking about profiling. You also have to consider that numbers here are relative. Let's assume another process: a MapReduce job that is meant to process 2 TB of information stored on a set of text files takes 20 minutes. In this case, you might not consider it as a slow process, even when it takes considerably more time than the slow web server mentioned earlier.

To get this type of information, you don't really need a lot of profiling experience or even complex tools to get the numbers. Just add the required lines into your code and run the program.

For instance, the following code will calculate the Fibonacci sequence for the number 30:

```python
import datetime

tstart = None
tend = None

def start_time():
    global tstart
    tstart = datetime.datetime.now()

def get_delta():
    global tstart
    tend = datetime.datetime.now()
    return tend - tstart

def fib(n):
```
return n if n == 0 or n == 1 else fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

start_time()
print "About to calculate the fibonacci sequence for the number 30"
delta1 = get_delta()

start_time()
seq = fib_seq(30)
delta2 = get_delta()

print "Now we print the numbers: 
start_time()
for n in seq:
    print n
delta3 = get_delta()

print "====== Profiling results ======
print "Time required to print a simple message: %(delta1)s" % locals()
print "Time required to calculate fibonacci: %(delta2)s" % locals()
print "Time required to iterate and print the numbers: %(delta3)s" % locals()
print "====== ======

Now, the code will produce the following output:

About to calculate the Fibonacci sequence for the number 30
Now we print the numbers:
0
1
1
2
3
5
8
13
21
#...more numbers
4181
Based on the last three lines, we see the obvious results: the most expensive part of the code is the actual calculation of the Fibonacci sequence.

**Where are the bottlenecks?**

Once you’ve measured how much time your code needs to execute, you can profile it by paying special attention to the slow sections. These are the bottlenecks, and normally, they are related to one or a combination of the following reasons:

- Heavy I/O operations, such as reading and parsing big files, executing long-running database queries, calling external services (such as HTTP requests), and so on
- Unexpected memory leaks that start building up until there is no memory left for the rest of the program to execute properly
- Unoptimized code that gets executed frequently
- Intensive operations that are not cached when they could be
I/O-bound code (file reads/write, database queries, and so on) is usually harder to optimize, because that would imply changing the way the program is dealing with that I/O (normally using core functions from the language). Instead, when optimizing compute-bound code (like a function that is using a badly implemented algorithm), getting a performance improvement is easier (although not necessarily easy). This is because it just implies rewriting it.

A general indicator that you’re near the end of a performance optimization process is when most of the bottlenecks left are due to I/O-bound code.

**Memory consumption and memory leaks**

Another very important resource to consider when developing software is memory. Regular software developers don’t really care much about it, since the era of the 640 KB of RAM PC is long dead. However, a memory leak on a long-running program can turn any server into a 640 KB computer. Memory consumption is not just about having enough memory for your program to run; it’s also about having control over the memory that your programs use.

There are some developments, such as embedded systems, that actually require developers to pay extra attention to the amount of memory they use, because it is a limited resource in those systems. However, an average developer can expect their target system to have the amount of RAM they require.

With RAM and higher level languages that come with automatic memory management (like garbage collection), the developer is less likely to pay much attention to memory utilization, trusting the platform to do it for them.
Keeping track of memory consumption is relatively straightforward. At least for a basic approach, just use your OS's task manager. It'll display, among other things, the amount of memory used or at least the percentage of total memory used by your program. The task manager is also a great tool to check your CPU time consumption. As you can see in the next screenshot, a simple Python program (the preceding one) is taking up almost the entire CPU power (99.8 percent), and barely 0.1 percent of the total memory that is available:

With a tool like that (the `top` command line tool from Linux), spotting memory leaks can be easy, but that will depend on the type of software you're monitoring. If your program is constantly loading data, its memory consumption rate will be different from another program that doesn't have to deal much with external resources.
For instance, if we were to chart the memory consumption over time of a program dealing with lots of external data, it would look like the following chart:

There will be peaks, when these resources get fully loaded into memory, but there will also be some drops, when those resources are released. Although the memory consumption numbers fluctuate quite a bit, it's still possible to estimate the average amount of memory that the program will use when no resources are loaded. Once you define that area (marked as a green box in the preceding chart), you can spot memory leaks.

Let's look at how the same chart would look with bad resource handling (not fully releasing allocated memory):
In the preceding chart, you can clearly see that not all memory is released when a resource is no longer used, which is causing the line to move out of the green box. This means the program is consuming more and more memory every second, even when the resources loaded are released.

The same can be done with programs that aren't resource heavy, for instance, scripts that execute a particular processing task for a considerable period of time. In those cases, the memory consumption and the leaks should be easier to spot.

Let's take a look at an example:

When the processing stage starts, the memory consumption should stabilize within a clearly defined range. If we spot numbers outside that range, especially if it goes out of it and never comes back, we're looking at another example of a memory leak.

Let's look at an example of such a case:
The risk of premature optimization

Optimization is normally considered a good practice. However, this doesn't hold true when the act of optimization ends up driving the design decisions of the software solution.

A very common pitfall developers face while starting to code a new piece of software is premature optimization.

When this happens, the end result ends up being quite the opposite of the intended optimized code. It can contain an incomplete version of the required solution, or it can even contain errors derived from the optimization-driven design decisions.

As a normal rule of thumb, if you haven't measured (profiled) your code, optimizing it might not be the best idea. First, focus on readable code. Then, profile it and find out where the real bottlenecks are, and as a final step, perform the actual optimization.

Running time complexity

When profiling and optimizing code, it's really important to understand what Running time complexity (RTC) is and how we can use that knowledge to properly optimize our code.

RTC helps quantify the execution time of a given algorithm. It does so by providing a mathematical approximation of the time a piece of code will take to execute for any given input. It is an approximation, because that way, we're able to group similar algorithms using that value.

RTC is expressed using something called Big O notation. In mathematics, Big O notation is used to express the limiting behavior of a given function when the terms tend to infinity. If I apply that concept in computer science, we can use Big O notation to express the limiting behavior of the function describing the execution time.

In other words, this notation will give us a broad idea of how long our algorithm will take to process an arbitrarily large input. It will not, however, give us a precise number for the time of execution, which would require a more in-depth analysis of the source code.

As I've said earlier, we can use this tendency to group algorithms. Here are some of the most common groups:
Constant time – O(1)
This is the simplest of them all. This notation basically means that the action we’re measuring will always take a constant amount of time, and this time is not dependent on the size of the input.

Here are some examples of code that have O(1) execution time:

- Determining whether a number is odd or even:
  ```python
  if number % 2:
    odd = True
  else:
    odd = False
  ```

- Printing a message into standard output:
  ```python
  print "Hello world!"
  ```

Even something more conceptually complex, like finding the value of a key inside a dictionary (or hash table), if implemented correctly, can be done in constant time. Technically speaking, accessing an element on the hash takes O(1) amortized time, which roughly means that the average time each operation takes (without taking into account edge cases) is a constant O(1) time.

Linear time – O(n)
Linear time dictates that for a given input of arbitrary length n, the amount of time required for the execution of the algorithm is linearly proportional to n, for instance, 3n, 4n + 5, and so on.
The preceding chart clearly shows that both the blue \((3n)\) line and the red one \((4n + 5)\) have the same upper limit as the black line \((n)\) when \(x\) tends to infinity. So, to simplify, we can just say that all three functions are \(O(n)\).

Examples of algorithms with this execution order are:

- Finding the smallest value in an unsorted list
- Comparing two strings
- Deleting the last item inside a linked list

**Logarithmic time – \(O(\log n)\)**

An algorithm with logarithmic execution time is one that will have a very determined upper limit time. A logarithmic function grows quickly at first, but it'll slow down as the input size gets bigger. It will never stop growing, but the amount it grows by will be so small that it will be irrelevant.

The preceding chart shows three different logarithmic functions. You can clearly see that they all possess a similar shape, including the upper limit \(x\), which keeps increasing to infinity.

Some examples of algorithms that have logarithmic execution time are:

- Binary search
- Calculating Fibonacci numbers (using matrix multiplications)
Linearithmic time – $\mathcal{O}(n \log n)$
A particular combination of the previous two orders of execution is the linearithmic time. It grows quickly as soon as the value of $x$ starts increasing.

Here are some examples of algorithms that have this order of execution:

- Merge sort
- Heap sort
- Quick sort (at least its average time complexity)

Let's see a few examples of plotted linearithmic functions to understand them better:

Factorial time – $\mathcal{O}(n!)$
Factorial time is one of the worst execution times we might get out of an algorithm. It grows so quickly that it's hard to plot.
Here is a rough approximation of how the execution time of our algorithm would look with factorial time:

![Graph showing factorial time growth]

An example of an algorithm with factorial execution time is the solution for the traveling salesman using brute force search (basically checking every single possible solution).

**Quadratic time – $O(n^2)$**

Quadratic execution time is another example of a fast growing algorithm. The bigger the input size, the longer it's going to take (this is true for most complexities, but then again, specially true for this one). Quadratic execution time is even less efficient than linearithmic time.

Some examples of algorithms having this order of execution are:

- Bubble sort
- Traversing a 2D array
- Insertion sort
Here are some examples of plotted exponential functions:

Finally, let's look at all examples plotted together to get a clear idea of algorithm efficiency:
Leaving aside constant execution time, which is clearly faster but most of the time impossible to achieve in complex algorithms, the order or preference should be:

- Logarithmic
- Linear
- Linearithmic
- Quadratic
- Factorial

Obviously, there are cases when you'll have no choice but to get a quadratic execution time as the best possible result. The idea is to always aim for the faster algorithms, but the limitations of your problems and technology will affect the actual result.

Note that between quadratic and factorial times, there are several other alternatives (cubic, \( n^4 \), and so on).

Another important consideration is that most algorithms don't have only a single order of execution time. They can have up to three orders of execution time: for the best case, normal case, and worst case scenarios. The scenario is determined by the properties of the input data. For instance, the insertion sort algorithm will run much faster if the input is already sorted (best case), and it will be worst (exponential order) for other types of input.

Other interesting cases to look at are the data types used. They inherently come with execution time that is associated with actions you can perform on them (lookup, insert, search, and so on). Let's look at some of the most common data types and their associated actions:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td>Indexing</td>
</tr>
<tr>
<td>List</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Linked list</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Doubly linked list</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Dictionary</td>
<td>-</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>( O(\log(n)) )</td>
</tr>
</tbody>
</table>
Profiling best practices

Profiling is a repetitive task. You'll do it several times inside the same project in order to get the best results, and you'll do it again on the next project. Just like with any other repetitive task in software development, there is a set of best practices you can follow to ensure that you get the most out of the process. Let's look at some of them:

Build a regression-test suite

Before starting any kind of optimization process, you need to make sure that the changes you make to the code will not affect its functioning in a bad way. The best way to do this, especially when it's a big code base, is to create a test suite. Make sure that your code coverage is high enough to provide the confidence you need to make the changes. A test suite with 60 percent code coverage can lead to very bad results.

A regression-test suite will allow you to make as many optimization tries as you need to without fear of breaking the code.

Mind your code

Functional code tends to be easier to refactor, mainly because the functions structured that way tend to avoid side effects. This reduces any risk of affecting unwanted parts of your system. If your functions avoid a local mutable state, that's another winning point for you. This is because the code should be pretty straightforward for you to understand and change. Functions that don't follow the previously mentioned guidelines will require more work and care while refactoring.

Be patient

Profiling is not fast, not easy, and not an exact process. What this means is that you should not expect to just run the profiler and expect the data from it to point directly to your problem. That could happen, yes. However, most of the time, the problems you're trying to solve are the ones that simple debugging couldn't fix. This means you'll be browsing through data, plotting it to try to make sense of it, and narrowing down the source of your problem until you either need to start again, or you find it.

Keep in mind that the deeper you get into the profiled data, the deeper into the rabbit hole you get. Numbers will stop making sense right away, so make sure you know what you're doing and that you have the right tools for the job before you start. Otherwise, you'll waste your time and end up with nothing but frustration.
Gather as much data as you can

Depending on the type and size of software you're dealing with, you might want to get as much data as you can before you start analyzing it. Profilers are a great source for this. However, there are other sources, such as server logs from web applications, custom logs, system resources snapshots (like from the OS task manager), and so on.

Preprocess your data

After you have all the information from your profilers, your logs, and other sources, you will probably need to preprocess the data before analyzing it. Don't shy away from unstructured data just because a profiler can't understand it. Your analysis of the data will benefit from the extra numbers.

For instance, getting the web server logs is a great idea if you're profiling a web application, but those files are normally just text files with one line per request. By parsing it and getting the data into some kind of database system (like MongoDB, MySQL, or the like), you'll be able to give that data meaning (by parsing the dates, doing geolocation by source IP address, and so on) and query that information afterwards.

The formal name for the stage is ETL, which stands for extracting the data from it's sources, transforming it into something with meaning, and loading it into another system that you can later query.

Visualize your data

If you don't know exactly what it is that you're looking for and you're just looking for ways to optimize your code before something goes wrong, a great idea to get some insight into the data you've already preprocessed is to visualize it. Computers are great with numbers, but humans, on the other hand, are great with images when we want to find patterns and understand what kind of insight we can gather from the information we have.
For instance, to continue with the web server logs example, a simple plot (such as the ones you can do with MS Excel) for the requests by hour can provide some insight into the behavior of your users:

The preceding chart clearly shows that the majority of requests are done during late afternoon and continue into the night. You can use this insight later on for further profiling. For instance, an optional improvement of your setup here would be to provide more resources for your infrastructure during that time (something that can be done with service providers such as Amazon Web Services).
Another example, using custom profiling data, could be the following chart:

It uses data from the first code example of this chapter by counting the number of each event that triggers the `profile` function. We can then plot it and get an idea of the most common events. In our case, the `call` and `return` events are definitely taking up most of our program's time.

**Summary**

In this chapter, we've covered the basics of profiling. You understood profiling and its importance. You also learned how we can leverage it in order to get the most out of our code.

In the next chapter, we'll start getting our hands dirty by looking at some Python profilers and how we can use them on our applications.
Where to buy this book
You can buy Mastering Python High Performance from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.