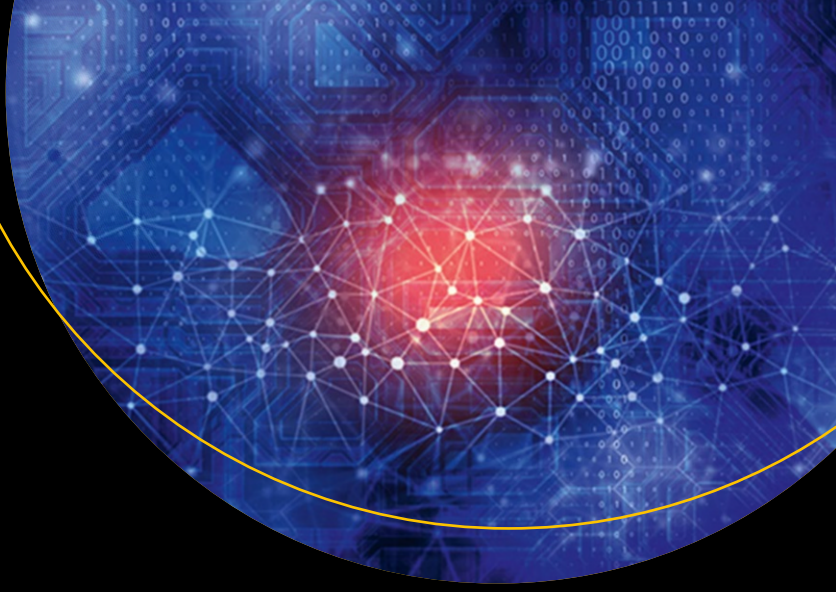

Good Habits for Great Coding

Improving Programming Skills with Examples in Python

Michael Stueben

Apress®

Good Habits for Great Coding

Improving Programming Skills with Examples in Python

Michael Stueben

Apress®

Good Habits for Great Coding

Michael Stueben
Falls Church, Virginia, USA

ISBN-13 (pbk): 978-1-4842-3458-7 ISBN-13 (electronic): 978-1-4842-3459-4
https://doi.org/10.1007/978-1-4842-3459-4

Library of Congress Control Number: 2018934317

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Todd Green
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484234587. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to the isolated C.S. teacher and student.

Table of Contents

About the Author

Michael Stueben started teaching Fortran at Fairfax High School in Virginia in 1977. Eventually the high school computer science curriculum changed from Fortran to BASIC, Pascal, C, C++, Java, and finally to Python. In the last five years, Stueben taught artificial intelligence at Thomas Jefferson High School for Science and Technology in Alexandria, VA. Along the way, he wrote a regular puzzle column for *Discover Magazine*, published articles in *Mathematics Teacher* and *Mathematics Magazine*, published a book on teaching high school mathematics: *Twenty Years Before the Blackboard* (Mathematical Association of America, 1998). In 2006 he received a Distinguished High School Mathematics Teaching / Edyth May Sliffe Award from the Mathematical Association of America.

About the Technical Reviewer

Michael Thomas has worked in software development for over 20 years as an individual contributor, team lead, program manager, and Vice President of Engineering. Michael has over 10 years experience working with mobile devices. His current focus is in the medical sector using mobile devices to accelerate information transfer between patients and health care providers.

Acknowledgments

Sincere thanks go to the following people: programmer Stephen Drodge for reviewing an earlier draft of this book and for making many useful suggestions; programmer Michael Ames for reviewing an earlier draft of this book; Dr. Stuart Dreyfus (University of California, Berkeley) for his personal thoughts on dynamic programming; Dr. Dana Richards (George Mason University) for mathematical advice on algorithms and puzzles; Dr. James Stanlaw of Illinois State University for discussions of signs, symbols, and semiotics; neurobiologist Paul Cammer (the best teacher I ever met) for years of discussions of effective teaching; hundreds of bright students who accepted my sarcasm and returned it right back to me; my wonderful wife of 40 years, Diane Sandford, for editing several versions of this book; Apress technical reviewer Michael Thomas; Apress editors James Markham, Jill Balzano, and Todd Green for their help in bringing this book into print; and the web site Stack Overflow. Finally, I must thank my two colleagues and master teachers: Dr. Peter Gabor (who reviewed this manuscript and made many suggestions) and Dr. Shane Torbert (who created the A.I. course I taught), both for five years of intense discussions about algorithms. Any mistakes in this book are due to the author, not those who gave me good advice.

Introduction

> For the player who wants to get ahead, he has only one piece of advice: get to work. Not with generalities taken from books, but in the struggle with concrete [chess] positions.—Willy Hendriks, *Move First, Think Later* (New in Chess, 2012), page 20.

THIS BOOK IS ABOUT improving coding skills and learning how to write readable code. It is written both for teachers and developing programmers. But I must immediately tell you that we learn how to write computer code only by trying to code many challenging problems, reflecting on the experience, and remembering the lessons we learned. Hence, you will find here more than twenty quizzes and problems. Chess coach Willy Hendriks is right: There is no other way.

I have spent more than 38 years both writing computer code and thinking about how to write code effectively.[1] I can't remember the last time I had a serious bug that I couldn't defeat—eventually. So what is the difference between myself and a novice? Part of it is that I notice details well, and I can stay focused for long periods of time. I can't pass that on to anyone, but I can show you some tricks gleaned from reading experts, talking to fellow programmers, and from analyzing my own mistakes. These tricks are guaranteed to reduce frustrations and failures. [2]

[1] I left Northern Illinois University in 1974 as a math major with just two C.S. courses behind me (COBOL and Fortran). Both classes employed optical card readers used with the University's IBM 360/370 computer. It often took 15 minutes of standing in line for the card-punch machine in order to change a single comma. At times there were no seats left in the computer room. I got tired of coding at midnight. The experience was so unpleasant that I vowed never to take another C.S. course. What got me interested in programming—actually the first time—was a TI programmable calculator with magnetic strips for memory. My first program would factor large integers, which I used for recreational mathematics. I started teaching Fortran at Fairfax High School (Virginia) in 1977. The school had three terminals (remotely connected) for the entire class. I tried to give each student about 10 minutes a week on a keyboard. Surprisingly, even under those conditions, some students became addicted to coding. Occasionally I found a student hiding under the tables after school so that he could program for hours after I locked up the room.

Later the H.S. curriculum changed from Fortran to BASIC, Pascal, C, C++, Java, and finally to Python. I discovered that I could not work in two languages at the same time. After six months with Python, I had forgotten my five years of Java. Python is definitely the most fun, with C/C++ second. COBOL was the worst, with Java the second worst. In fact, I think the Java language has discouraged many high school teachers from teaching C.S.

[2] "Essentially every approach works for a small project. Worse, it seems that essentially every approach—however ill-conceived and however cruel to the individuals involved—also works for a large project, provided you are willing to throw indecent amounts of time and money at the problem."—Bjarne Stroustrup, *The C++ Programming Language*, 2nd Ed., (Addison Wesley, 1991), page 385.

The computer code in this book is written in the Python language, which is almost executable pseudo-code. It comes with batteries, as they say. For example, consider the Python `min` function:

```
print(min(3,5)) # output: 3
```

Most languages have a `min` function. But look what Python's `min` can do:

```
paths = [[7,1,1], [5,1,3]]
print(min(paths))            # output: [5,1,3], because 5 < 7
print(min(paths, key = sum)) # output: [7,1,1], because 7+1+1 <
                                                 5+1+3
```

NOTE TO READER: The code examples appearing in this book (62 characters per line) were taken from programs with 80 characters per line. Consequently, some of the longer lines were broken into two (usually indented) lines. This affects their readability in a few cases, but has kept the type larger and easier to scan.

In Python, you can pass a function as a parameter to another function. Why would anyone want to do that? Imagine that you wrote several different functions, each using a different algorithm, to solve the same problem. Then you wrote a test function to test each algorithm. You could change the name of each of your functions, one at a time, to the name the test function expects to call. Or you could just pass the function name as a parameter and not have to change any code, which is much easier. Below is an example.[3]

```
def fn1():
    print('Hello:   ', end ='')
    return (1)
```

[3]Alas, if the different algorithms have different function signatures then this method fails—e.g., the bubble sort, the selection sort, and the insertion sort all pass just the array. But the recursive quick sort passes the array *and* the position of the first and last elements.

```
def fn2():
    print('Goodbye: ', end ='')
    return(2)

def test(func):
    print('testing', func.__name__, 'Output =', func())

def main():
    print('In program',__file__) # output: In program C:\test.py
    test(fn1)                    # output: Hello:   testing fn1
                                                    Output = 1
    test(fn2)                    # output: Goodbye: testing fn2
                                                    Output = 2
```

In Python, you can make multiple assignments in one line and can do a swap in one line.

```
a, b, c  = 1, 2, 3 # multiple assignment
a, b = b, a        # swap
```

A list (array) in Python can simultaneously hold different data types. A function can return more than one parameter. The last element in a list has index `-1`. The second-to-last element has index `-2`, etc. How convenient is that?[4] The extremely useful associative-array concept exists in Python as a built-in data structure called a "dictionary." The quick sort as shown below can be written in two logical lines. OK, five printed lines, but they are easy-to-understand lines.

```
def quickSort(array):
    if len(array) <= 1: return array
    return   quickSort([x for x in array[1:] if x <  array[0]]) \
           + [array[0]]                                         \
           + quickSort([x for x in array[1:] if x >= array[0]])
```

[4]This attribute can be a problem. I once ran a loop that moved backwards through a list. When it went past 0, the out-of-range error was not caught, because it just started at over at the end.

My point is that the language of Python is nearly ideal for developing algorithms. The main drawback is speed. The language is interpreted, not compiled. But only with graphics have I encountered a speed problem with Python.

You may not understand Python. I have looked at books written in languages that I didn't understand, and (if the code was not too long or too complex) I still understood the main ideas for algorithms. So I'm hoping you can do the same. Let's find out. Consider a function to raise a positive integer to a power. For example, `power(5, 23) = 5**23 = 11920928955078125`. Of course, there is a built-in function (`pow`) and a built-in operator (**) in Python to do this for us. But those won't help us in the application I have in mind. Can you understand the following code? I wrote it in two versions.

```
def power(base, exponent):
    product = 1
    for n in range(exponent):
        product *= base
    return product

def powr(b, exp):
    x = 1
    for n in range(exp):
        x *= b
    return x
```

If you can understand this code, then you can follow much of the code in this book. By the way, there are already a few lessons to be learned here.

1. In my opinion, the second version is easier to understand than the first version. Short identifiers—with obvious meaning—for short scope are more readable than longer identifiers, which are better for function names, class names, and variables in complex code.

2. The reason `pow` would be a poor function name is because the function would overwrite the built-in `pow` function of the same name, *for the entire program*. The reason `exp` is acceptable is because it overwrites the built-in `exp` function only for the short scope of the function. Good names are sometimes hard to find. Believe it or not, I have had students name their programs `random` and one student named his program `print`. Then they were confused when, under Linux, their `random` and `print` functions failed to work.

For almost any application, this four-line power function would be ideal. But it could be much more efficient. Again, consider `5**23 = 11920928955078125`. We don't need 22 multiplications to do the arithmetic. Notice that we can break up 5**23 like this:

```
5 * (5*5) * (5*5*5*5) * (5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5) = 5**23,
```

and like this:

```
5 *  5**2   *    5**4   *          5**16                  = 5**23,
```

where the exponents (1, 2, 4, and 16) add up to 23. (Recall $x^a \times x^b = x^{a+b}$.)

Once we calculate `a = 5*5`, it takes only one more multiplication to produce `b = a*a`. Then only one more multiplication to produce `c = b*b`. And only one more multiplication to produce `d = c*c`. Altogether we can produce 5**23 in only 7 multiplications: (`5*a*b*d`). The trick is to write the exponent 23 as a binary number: 23 (base 10) = 11101 (base 2). Then multiply each digit (in reverse order) by the base (5 here) raised to a power of 2:

```
1*(5) * 1*(5**2) * 1*(5**4) * 1*(5**16) = 5**23.
```

You might try to write this function now, in your own preferred language. The hills make us strong, as they say in cycling. But you may be too busy, and the exercise probably seems both complex and pointless. Who would want such a function anyway? But in a situation we will see later, this binary-jumping type of multiplication will significantly speed up a function. So for the time being I will give you a pass in writing this algorithm. My eight-line solution follows. Python, of course, has built-in features to change an integer into a binary string and to reverse the characters in a string. No wonder people like to code in Python.

```
def powr(b, exp):
    binStng = str(bin(exp))[2:] # Change integer exp to a
    binary string.
    revStng = reversed(binStng) # Reverse the digits (alt. =
    binStng[::-1]).
    product  = 1
    for ch in revStng:
        if ch == '1':
           product *= b
        b *= b
    return product
```

If you don't know Python, the first two lines will be a mystery, hence the comments. If you do know Python, you still may learn something new and useful in those two lines. The loop should be clear to anyone who has worked with `for` loops.

I've been programming for nearly four decades, and had written a variation of this function about a month previously. Nevertheless, it took me nine runs to get this function working. (I had placed `b *= b` above the `if` statement, and it took me a while to realize the order mattered.) I mention this to make the point that most programmers, certainly the author, fail to write correct code in the first few attempts.

Before we leave this introduction, I want to give you three quizzes that will tell you what this book is all about.

QUIZ 1.

```
# If we optimize the one-line BODY of this for-loop, then
# what is the MINIMUM number of multiplications necessary?
# Do NOT use an exponential operator (**). Do NOT use a built-
# in power function. You MUST use the symbol * to indicate
# multiplication.
#
    total = 0
    for n in range(1, 3000000):
         total += (2*n*n*n + 3*n*n + 4*n) # <--Improve this line.
    print('total =', total)
#--------------------------------------------------------------
```

The answer is at the end of this chapter, but try to solve it now. Passive reading will not take you far. One of my colleagues (the amazing Ria Galanos) was asked the following question in a Google summer interview:

QUIZ 2. Given `x`, an *unsorted* list of the first 100 positive integers, one of the integers is replaced by 0: `x[randint(1,100)] = 0`. Write the code—any way you want—to print the missing (replaced) integer. A solution is in the footnote.[5]

[5]QUIZ 2 ANSWER: **`print(5050-sum(x))`**. Where did the 5050 come from? That is the sum of the first 100 positive integers. We can compute this number in our heads. 1+100 = 101, 2+99 = 101, 3+98 = 101, … 50+51 = 101 (a trick worth remembering). Then, 50* 101 = 5050. I later found this problem in Peter Winkler's *Mathematical Puzzles—A Connoisseur's Collection* (A.K. Peters, 2004), page 102. P.S. She got the job.

There are three main cultures of coding.[6] The people in these cultures all use computers, yet they rarely interact with each other. Perhaps you can tell now which one most interests you.

1. **The software developer (industry)**, who works with libraries of previously developed code to produce new software tools, who devises schemes to manage complexity in software programs, who determines what features make programming tools more useful, or more fun (games), etc.

2. **The computer scientist (theory)**, who develops and analyzes algorithms, who studies the syntax and semantics of computer languages, who designs efficient storage and retrieval strategies, who determines what can be computed efficiently, etc.

3. **The computational scientist (problem solving in other fields)**, who uses the computer as a scientific tool in modeling and simulations, as a way to visualize spatial and temporal patterns, as way to solve equations, as a way to efficiently organize, search, and find patterns in data, etc.

In this book there are references to all three cultures. Most beginners focus on just learning a language, learning data structures, and building coding-specific problem-solving skills. What is missing is learning to write readable code. In my experience, readability is difficult to teach well in both high school and college courses. There are reasons for this, which I will give you later. But I would like you to compare your ability to write

[6]Brian Hayes, "Cultures of Code", *American Scientist*, Vol. 103, No. 1, January–February, 2015, pages 10–13. This article is also on the Internet.

readable code with mine. Imagine we are the last two candidates for a summer coding job. The interviewer gives us the following assignment:

QUIZ 3. If I take a 52-card deck and I shuffle it well, then what is the probability that at least one card remains in place?[7] Solve this problem by computer simulation[8] (here, sampling) in your favorite language. That is, shuffle 1,000,000 sorted arrays and determine what percentage of them have at least one element remaining in place. Express this number as a probability. Be sure to make your code as readable as possible. Bring me your code tomorrow morning. I'll have one of our programmers look at your two programs and tell me whose code he prefers.

Quiz 3 is the most important quiz in this book. If you attempt no other problem in this book, try to write this short program—and a complete program is expected, not just a function. My code follows, with notes as to why I made some design decisions. Before you compare your code to mine, what can you tell me about the programmer who will judge our code? My answer is in the footnote.[9]

How would you answer this interview question: "So, what will you do this summer if you don't get this job?" My suggested answer-to-impress is in the footnote.[10]

---

[7] This is known as Montmort's Matching Problem. See Isaac Todhunter, *Theory of Probability* (London: Macmillan, 1865), (Chelsea Reprint, 1965), page 91 (online). Curiously, for a deck of any number of cards greater than 5 the answer is almost the same.

[8] *Tech. Note. Wikipedia* states that a **computer model** is the set of algorithms capturing the essence of a process or system, and that a **computer simulation** is the running of those algorithms. That being said, the terms *simulation* and *model* are often interchanged in both writing and speaking. Random sampling to obtain numerical approximations by ratios is called the **Monte Carlo Method**.

[9] He wants someone who pays attention to detail, who has some maturity in his/her coding skills, and who wants this job so much that the candidate will try to impress the code reviewer. Will your code show this?

[10] "I'll have to go back to reading computer books and working problems on my own. I would much rather gain some experience this summer by working in industry."

QUIZ 3 ANSWER.

```
"""+==========+========-========*========-========+==========+
   ||                   A SHUFFLING PROBLEM                   ||
   ||             by M. Stueben (October 8, 2017)             ||
   ||     Interview Question, Mr. Jones, XYZ Corporation      ||
   ||                                                         ||
   || Description: By computer stimulation this program       ||
   ||              determines the probability of a deck of    ||
   ||              52 cards having at least one unmoved card  ||
   ||              element after shuffling. (Answer: 0.63,    ||
   ||              rounded.)                                  ||
   ||                                                         ||
   || Language:  Python Ver. 3.4                              ||
   || Graphics:  None                                         ||
   || Downloads: None                                         ||
   || Run time:  Approx. 43 seconds for 1,000,000 runs of a   ||
   || 52-element array.                                       ||
   +=============================================================+
"""
########################<START OF PROGRAM>#####################
def printHeading():
    print('                      A SHUFFLING PROBLEM')
    print('                    (currently calculating)')
#-------------------------------------------------------------

def shuffleArrays():
    totalArrays = 0 # Arrays with at least one unmoved element
    after shuffling.
    for trial in range(TRIAL_RUNS):
        array = list(range(LIST_SIZE))
        shuffle(array)
```

```
        for num in range(LIST_SIZE):
            if array[num] == num:
                totalArrays += 1
                break
    probability = round(totalArrays/TRIAL_RUNS, 2)
    return probability
#----------------------------------------------------------------

def printResult(probability):
    print('   Result:', probability ,'is the probability of an
    array having at')
    print('   least one unmoved element after shuffling. This
    is based')
    print('   on a computer simulation with an array size =',
    LIST_SIZE, 'and')
    print('  ', TRIAL_RUNS, 'trial runs.')
#============<GLOBAL CONSTANTS and GLOBAL IMPORTS>=============

from random import shuffle
TRIAL_RUNS = 1000000
LIST_SIZE  =      52
assert LIST_SIZE > 1, 'LIST_SIZE must be greater than 1.'
#================================================================

def main():
    printHeading()
    probability = shuffleArrays()
    printResult(probability)
#----------------------------------------------------------------
```

```
if __name__ == '__main__':
    from time import clock; START_TIME = clock();main();
    print('\n   '+'- '*12);
    print('   PROGRAM RUN TIME:%6.2f'%(clock()-START_TIME),
    'seconds.');
##################<END OF PROGRAM>##############################
```

Output:

```
                    A SHUFFLING PROBLEM
                  (currently calculating)
Result: 0.63 is the probability of an array having at
least one unmoved element after shuffling. This is based
on a computer simulation with an array size = 52 and
1000000 trial runs.

- - - - - - - - - - - -
PROGRAM RUN TIME:  43 seconds.
```

What was I thinking when I wrote this code?

1. The pretty box, the centering, the vertical alignment are all just window dressing. Is this fancy stuff necessary? Like it or not, looks matter.
2. The minimum information is a title, your name, the date, and a program description. The other information in the box is optional, but shows attention to detail. I want to look like I am trying to impress the interviewer.
3. There are no spelling, punctuation, or grammatical errors (important). I used complete sentences in both the description and the program output.

4. This program is so simple, why not place all of the code in the main function? Two reasons: 1) major code chunks need descriptive, self-documenting names, and 2) the main function is expected to be mostly a list of calls to other functions (stepwise refinement).
5. Comments are almost unnecessary, because the code is self-documented and well organized. Docstrings in a program this short are unnecessary. Still, some interviewers may expect them in an interview program.
6. The variable names are descriptive. Over-abbreviation, to save a few keystrokes, was avoided. In particular `array` was used, not `a` or `arr`.
7. The output is well-labeled.
8. The two constants are in all caps.
9. The import and global constants are placed above the main function. It is usual to place them at the top of the program in large commercial programs. In extremely small programs I think they are better placed above the main function.
10. An `assert` is used to catch ridiculous cases. Error traps should be common in student code.
11. The indenting is everywhere consistent: 4 spaces.
12. Some output is printed immediately. I do not want the user to stare at an empty screen for 43 seconds and wonder if the program is running.

13. The following two lines could have been combined into one line, but then the descriptive variable `probability` would not be part of the code.

```
probability = round(totalArrays/TRIAL_RUNS, 2)
return probability
```

14. The run-time is printed. (Every program I write prints its run time.) Some text editors and IDEs automatically print the runtime. That is how important this statistic is.

15. The answer is correct.

Did you learn anything? Readability is a hot topic with conflicting opinions. What you are used to seeing will look more readable than what I am used to seeing. But it is always good to know how other people think, even if we disagree. I hope that I have interested you in reading the rest of the book. If not, at least start the next chapter. Much of this book is text, not code. Good luck.

> Documentation and readability are as important to software quality in the long run as speed of creation, correct functioning, and performance are in the short run.—L. Peter Deutsch, (ACM Fellow), Found on the Internet (ACM SigSoft, Software Engineering Notes, Vol. 24, Issue 1, January, 1999).

* * *

QUIZ 1 ANSWER: `total += ((n+n + 3)*n + 4)*n.` (Only two multiplications are necessary.) Here are some running times (of repeated calls) for six different versions:

1. `2*n*n*n + 3*n*n + 4*n -->. . . . . .1.09 seconds  (original)`

2. ((n+n + 3)*n + 4)*n -->. 0.76
   seconds (fastest)
3. n*((n + n + 1) * (n + 1) + 3)-->....0.86
   seconds. (2nd fastest)
4. ((n+n)*n + (n+n+n))*n + n+n+n+n --> 1.39
   seconds. (also two stars)
5. 2*n**3 + 3*n**2 + 4*n -->. 2.85
   seconds. (poor)
6. 2*pow(n,3) + 3*pow(n,2) + 4*n --> . 3.23
   seconds. (worst)

For non-Python people, please excuse this digression into the Python language. When I showed my powr function to my colleague Peter Gabor, he suggested the following improvements:

```
def powr1(b, exp):
    myPowr = 1
    while exp > 0:
        myPowr *= ((~exp)&1) or b
        b *= b
        exp >>= 1  # Shift one bit right.
    return myPowr
```

Explanation: The Python tilde (pronounced TIL-da) operator (~) is a bit-wise operator. The expression ~x is the same as -(x+1). It is only employed here because it will flip the right-most bit. The expression (~exp)&1 is equivalent to the rightmost bit of ~exp. The expression ((~exp)&1) or b

will be either `1` or `b`. In Python the `or` operator returns the value of the last expression evaluated, not `True` or `False`. His `powr1` function can also be written like this:

```
def powr2(b, exp):
    myPowr = 1
    while exp > 0:
        if exp%2 == 1:   # An odd exp means right-most bit is 1.
            myPowr *= b
        b *= b
        exp //= 2
    return myPowr
```

Or even like this (a form I would not want to debug):

```
def powr3(b, exp):
    return (not exp) or ((powr3(b, exp >> 1)**2) * (((~exp)&1)
    or b))
```

PART I

Not Learned in School

CHAPTER 1

A Coding Fantasy

ONCE UPON A TIME, a talented young programmer was in a situation where he did not have the resources to seek more education. He had a dead-end job that would never allow any promotion. Further, his family could not help him, and he lived in a decaying and unsafe part of town. Our programmer had four friends who had developed similar programming skills and who also felt limited by their opportunities. They were all slightly depressed and worried about their futures.

Suddenly, the five programmers discovered an amazing opportunity. If they could team up and write a particular computer application, then the attention they would receive would immediately open doors for better jobs.

Of course, anyone in this situation would want to attempt to write the application. But it was not so simple. Previously, the most challenging program each of them had written took three weeks of time at 1-2 hours a day. Most of the time was spent in debugging. Some of those bugs were so difficult to track down that they had twice given up on their programs, only to come back to them out of curiosity. In fact, those three weeks of time were actually spread over six weeks. They all had the same experience.

Upon reviewing the work for this new project, it appeared that the job naturally could be divided into five equal parts. The problem was that each part was five times longer than anything any one of them had worked on before. They had 40 weeks to finish the project. In theory, if all could stay focused, that was more than enough time to finish. But in practice, the complexity was beyond what anyone thought he or she could

M. Stueben, *Good Habits for Great Coding*, https://doi.org/10.1007/978-1-4842-3459-4_1

do. The tantalizing prize was also an invitation to failure. Briefly each thought that the quiet go-nowhere life they were currently living might be preferable to 40 weeks of misery that almost certainly would lead to failure. Who needed that? Maybe something else would come along. Eventually in conversation, the five friends realized that this defeatist thinking is a common reason why people do not climb out of their poor situations in life. Yet, as each one currently understood the project, it was too difficult for them to complete. If they could increase the likelihood of success, then it might be worth a try. So, what to do?

First, the five programmers had to accept the fact that they would have to turn themselves into programming robots. Many of the pleasures that were part of their everyday lives would have to be replaced with hours of coding. This would require a change of both habits and perspective on life. Could they do this? The prize dangling in front of them just might be enough.

The real problem was debugging. Although all parts of the code seemed reasonable enough, there were so many parts that debugging problems would arise en masse. They didn't see how any one of them could be successful. Then someone suggested a solution: For almost every key function written, a companion function could be written to test that function. After each session of coding, the testing functions would be run. Another program would import most of the important functions and run several sets of data through each function. The data would test, for example, almost every **if** statement in a function.

This meant that if a redesign occurred, the functions adversely affected would be flagged immediately. Writing two functions for every one function needed in the application would be extra work, but the testing functions would be simple to write, and mostly similar to each other. This scheme, called *unit testing*, seemed to offer hope.

Another member suggested that the group get together every week to read each other's code, to discuss problems, and to suggest solutions coming from fresh eyes. In these *code reviews* they would

share both problems and hard-learned solutions. Another suggestion was to document almost every key function with an English description (*docstrings*), so that any of the other members could more easily follow the code. Another suggestion was that they should occasionally try to work in pairs (*pair programming*): one typing and the other thinking about what is being typed.

The group felt that their only chance at success was to adopt these conventions. One of the members later described working with these conventions as writing code in a straightjacket.

Soon after the coding began, the members noticed that progress was slow but steady. The inevitable redesigns, usually based on overlooked special cases and poorly chosen data structures, almost always caused a domino effect of other changes. These changes were all quickly noticed and located by unit testing.

The members also began discussing small differences in coding styles—e.g., should one write

```
if (x and y) == True: print(x)
```

or

```
if x and y: print(x)?
```

Because of differing opinions, they decided to vote on a group style and stick to the group's decision. Eventually, their conventions, which were often arbitrary, began to look correct and any different convention looked wrong. Because the same style was used by everyone, they all became efficient at reading code written in their *shop style.*

To make a long story short, their combination of sacrifices, commitment, and good decisions about writing code enabled them to complete the project and win a better life. Eventually, they were hired by employers seeking expert programmers.

Their new employers appreciated the members of this group for several reasons. First, the programmers had put so much of their lives into writing