CODING
Best Practices

Essential Tips and Techniques for Writing Clean, Correct, Sustainable Code

by
Michael Marsh
Evan Golub
Anwar Mamat

Coding Best Practices

Essential Tips and Techniques for Writing Clean, Correct, Sustainable Code

Michael Marsh Evan Golub Anwar Mamat

Contents

Introduction

This book covers some topics from the introductory Computer Science sequence that are less directly assessed via tools such as exams and style grading, so end up being easier to forget. It might also touch on topics that were not covered in the specific sections of courses you took, but they are still good to know if you are going to write real-world code. This text is short, by design, so that you can use it as a quick reference or refresher. We provide examples in Java, Python, and C, to better illustrate the practices we're presenting.

The material is divided into the following chapters:

- *Coding Basics*: These are simple principles that typically apply to either a single line or block of code.
- *Larger Software Structure*: This addresses interactions between multiple blocks or code, methods, or classes, and focuses on making your code easier to read and maintain.
- *Testing Your Code*: Here we discuss details of how you can test your code to make sure it behaves in the way you expect, including how to test for corner cases in the logic.

Coding Basics

We begin with some basics that apply to single expressions, statements, or blocks of code. The first important thing to understand is that compilers are *very good* at what they do. The compiler's main job is to take (hopefully) readable, maintainable code written by people and turn it into efficient bytecode or machine code that the CPU can run.

Our goal, therefore, should primarily be to write accurate code that we or someone else can read and maintain. With this in mind, we will consider several general principles that fit into typically one to ten lines of code. While comments are a useful tool, the following examples are in the context of improving the readability of the code itself.

Make Your Variables Names Expressive

There are few things as difficult when trying to understand someone else's code as figuring out what a single-character variable name represents. Unless the code is computing acceleration, a variable named `a` conveys no information to the reader. Even then, `acceleration` (or even `accel`) do a much better job of informing the reader about what they are reading.

While our simple examples in this book will often ignore this (they are simple templates, after all), as a general rule you should *only* use single-character names for loop variables, and even then these should typically be for simple iterative (integer range) loops. A `while` or `foreach` loop should have a more meaningful variable name.

Some important things to keep in mind:

- If you are working on a team, other people will need to understand your code.
- If you are asking a TA for help with a project, they will need to understand your code (and might send you away to re-write it more clearly first).
- In six months' time, *you* are one of those other people, as far as your code is concerned.

Assignments Might Have a Value

Depending on the programming language, an expression like `a = 1` might be a simple *assignment statement*, or it might also return a *value*.

Normally, it doesn't matter if you are unaware of whether it returns a value, since we can easily have the following in Java or C with no consequence of not considering that a value is returned:

```
a = 1;
```

or, in Python:

```
a = 1
```

Of course, in C (and some similar languages), we can have *chained assignments* like:

```
b = a = 1;
```

Why does this work? Because first we assign `1` to `a`, which has a *return value* of `1`, which we then assign to `b`. This is very handy when, for example, initializing several variables to `0`.

While this allows you to write more compact code in C (which is fine, as long as you aren't putting anything too long or complex in a single statement), it can cause problems when you mis-type something. For example:

```
if ( a = 1 ) { ... }
```

What you *probably* meant to type was `a == 1` (a comparison), not `a = 1` (an assignment). In Java or Python this would fail with a syntax error since a Boolean result is needed. In C, however, this would first assign `1` to `a`, and then evaluate to `1`. In C a non-zero integer is treated as a Boolean true, which means *the result of this expression is always* ***`true`*** so rather than testing whether `a` was `1` you are assigning `a` to be `1` and also getting `true` for the conditional statement.

You might not write C code very often, but you likely *will* be programming in a number of different languages in your career, so here's a useful tip to convert a valid typo into a syntax error that the compiler will catch in almost any language:

When comparing for equality, use a non-lvalue on the left-hand side.

What is an *lvalue*? It's anything that is legal to put on the left-hand side of an assignment, like a variable. A constant or literal is not an lvalue, so if we were to write

```
if ( 1 = a ) { ... }
```

the compiler would flag this as a syntax error in almost every language. The correct version of this in Java or C would be

```
if ( 1 == a ) { ... }
```

or in Python

```
if 1 == a: ...
```

and will work in any language with the `==` equality operator.

Don't Compare Booleans to True or False

In a situation where you are writing an `if` block or `while` loop, and you have some Boolean variable or expression that determines whether the body should be executed, you should not explicitly test it against `true` or `false`.

In the previous section, we saw:

```
if ( 1 == a ) { ... }
```

Sometimes, we have a variable or the return value of a function in our conditional. For example:

```
int f();
if ( 1 == f() ) { ... }
```

What if `f()` returns a Boolean value?

```
boolean f() { ... }
...
if ( true == f() ) { ... }
```

What's wrong with the above? Well, it's redundant and consumes extra processing power. We could write it as

```
boolean f() { ... }
...
if ( f() ) { ... }
```

and it would be identical in meaning, be cleaner, and remove a superfluous Boolean evaluation.

You can also apply this when testing for something being `false`

```
boolean f() { ... }
if ( !f() ) { ... }
```

should be used instead of

```
boolean f() { ... }
if ( false == f() ) { ... }
```

as it is again more succinct, and a `not` is a simpler operation than a `==` is as far as the CPU is concerned. When your context has you programming in C, time efficiency might be the goal.

Don't Recompute Things

Let's say we have a function `g()`, and we want to make several comparisons to the output of it in Python:

```
def g(...): ...

if 1 == g(...): ...
elif 2 == g(...): ...
elif 3 == g(...): ...
else: ...
```

The problem here is two-fold. First, we're calling `g(...)` three times. Second, we are presumably assuming it will return the same thing each time if given the same input. For our context, we will assume that the repeated calls do return the same value each time. The issue we address is that each function call causes a new frame to be added to the stack, control is passed to the function, all of the function's logic is executed again, and then the result is returned. In some contexts, that will be a significant amount of repeated work. Even for "simple" functions such as `sqrt(x)` where this might not seem like a lot, it can add up. If we instead write:

```
def g(...): ...

g_val = g(...)
if 1 == g_val: ...
elif 2 == g_val: ...
elif 3 == g_val: ...
else: ...
```

then we only call `g(...)` *once*, and we're also guaranteed that our cascade of `if-elif-else` is using the *same value* in all of the conditional tests.

Order Matters in Conditionals

Most languages evaluate sub-expressions from left to right, ending as soon as the result is guaranteed. This is commonly known as short-circuiting. Consider

```
if ( a || b ) {...}
```

If `a` is true, then the entire expression resolves to true without considering `b`. If `a` is false, then we have to consider the value of `b`.

Similarly, in

```
if ( a && b ) {...}
```

our early-exit is when `a` is false, since again the value of `b` doesn't matter.

If you don't know which sub-expression is more likely to be true or false, then the order is less important, though if one is more expensive in the context of computational complexity then that could still be important. Let's consider

```
if x**7<y and y>=100: ...
```

Exponentiation is expensive (three multiplications when done efficiently, in this case), so this would be better written as

```
if y>=100 and x**7<y: ...
```

so that we can avoid the computation if `y < 100` even if only some of the time.

Keep Conditionals Simple

Conditionals are one place where if you know a bit about what optimizations the compiler will do, you can design code to rely more on the compiler's abilities. In source code, you should aim for clarity, but might also want to consider efficiency. Consider the following Java code segment:

```
if ( x < y + 1 ||
    ( x >= y && y < 1 ) ||
    z == 10 ) {...}
```

This isn't *too* bad to read, but it can take some time to parse the logic. Adding a few more parenthesis strategically could help. However, now consider the following:

```
bool small_x = x < y + 1;
bool small_y = x >= y && y < 1;
bool fixed_z = z == 10;
if ( small_x || small_y || fixed_z ) {...}
```

This makes it more clear when reading (and debugging) what you're trying to do, and in this case the compiler will essentially produce identical bytecode. You're also less likely to make mistakes such as altering the logic by forgetting parentheses that impact the order of operations, and thus the result:

```
if ( x < y + 1 ||
     x >= y && y < 1 ||
     z == 10 ) {...}
```

However, if your language supports short-circuit evaluation of Boolean expressions had the `||` logic been `&&` logic, then the overall runtime could potentially have changed. Also, some programmers might utilize *clever code* to design an expression such that a function call is not made if an earlier condition is not met.

Consider

```
myNodePointer.hasNext() && myNodePointer.goNext()
```

While this can be appealing for brevity, it makes your code both harder to maintain for the next person (potentially *you* in six months) and less portable to another language that does not have the same short-circuiting behavior.

There is also the option of nesting conditionals, which might have a small cost if the compiler optimizes for certain pipelining, but which might be better for ease of reading. For example, while many would likely prefer

```
if ( x > 1 && y < 3 ) {...}
```

to

```
if ( x > 1 ) {
   if ( y < 3 ) {...}
}
```

because these expressions are short, when you have longer compound Boolean expressions they can be much harder to follow.

You also might find that you need to add more cases later, so even in this simple case, you might find your code later looking more like:

```
if ( x > 1 && y < 3 ) {...}
else if ( 0 == x ) {...}
else if ( x > 1 && 20 == y ) {...}
```

Logically, we could group the first and third conditionals on the `x > 1` expression, and the flow would again be more clear to read. This is a place where some of the logic in CMSC250 can come in handy.

Don't Do Function Calls or Expensive Computations in Loop Conditionals (Don't Recompute Things Redux)

Let's say we're trying to factor a number `n`. We might have a loop with the following logic:

```
for ( int i = 2 ; i <= sqrt(n) ; i++ ) {...}
```

Each time we iterate through the loop, we are recalculating `sqrt(n)`, which is expensive. It's much better to do something like:

```
double sqrt_n = sqrt(n);
for ( int i = 2 ; i <= sqrt_n ; i++ ) {...}
```

so that we only have to compute the square root once. This is similar in nature to the earlier section on not recomputing things.

Not All Computations are Equal

If you are going to be comparing values, say x and $f(x)$, it's important to consider whether it's faster to compute $y = f(x)$ or $x = f^{-1}(y)$. For example,

```
if sqrt( (px-cx)**2 + (py-cy)**2 ) < radius : ...
```

will take more time than the mathematically equivalent

```
if (px-cx)**2 + (py-cy)**2 < radius**2 : ...
```

Don't Change For-Loop Variables in the Body

If you see a loop like

```
for ( int i = 0 ; i < n ; i++ ) {...}
```

you expect this loop to run `n` times, and `i` will have values `0` through `n-1`, each in ascending order.

Now consider:

```
for ( int i = 0 ; i < n ; i++ ) {
   if ( f(i) == 3 ) {
      // skip the next one!
      i++;
   }
}
```

At this point, we have no idea how many times the body of the loop will run! The better approach would be to convert it to a `while` loop in this type of situation.

Don't Optimize Prematurely

This might seem like odd advice, since we've presented a few cases where you can make your code more efficient. While those are good programming habits to develop, you shouldn't necessarily comb through your code looking for individual instances to fix. You should also be careful about unrolling loops or other optimization tricks (you really don't need a Duff's Device, unless you're in very specific situations) unless you *know* they will make a difference.

Basically, keep your code as readable as possible, since you're going to need to be able to debug or extend it eventually. In *Testing Your Code* we'll look at how to figure out where your code is actually spending the most time, which is where you want to focus your efforts *if* your code needs to be made more efficient.

Larger Software Structure

We now turn to principles that apply to larger segments of code and the larger scope of writing sustainable projects. Here, we assume the previous lessons have been learned, and our focus is on maintainability of code. While these concepts will help when bugs or inefficient code are discovered, and we will mention bugs and inefficient code, the ***prevention*** of bugs and inefficient code in general is not the central concern in this chapter.

Avoid Code Duplication

Every function, method, or block of code is a potential location for bugs to appear. Tracking down these bugs can be difficult and time-consuming, and one key to minimizing bugs beyond careful testing of modular code (see Chapter 4) and mitigating their impact is minimizing code duplication. That is, once you write a segment of code to accomplish a specific task (whether complex or not) in one place, you should utilize coding practices that will allow you to use that code segment wherever in your program you need that task accomplished. Often, you will do this by defining a *helper* function or method, which isolates the code which accomplishes the task, and calling it in all of the places where it is needed. This is sometimes referred to as modular or structured programming. In fact, one of the key ideas behind object-oriented programming is *encapsulation*, where we support code reuse by creating classes that bundle data with the code to operate upon that data.

As an example, consider we had the need to find the square root of a number, and no such functionality was provided by the programming language itself. We might be tempted to write a block of code to accomplish this task, and then copy & paste that block of code each time we needed to find a square root, changing the variable each time. This approach would open us up to several potential issues. First, we might have several places where the variable name needs to change, and miss one or more. This would likely be a challenging bug to find. Second, if we later discover that the code we had written has a bug (perhaps a subtle corner case) or inefficiency within it, we would need to track down ***every*** place to which we had copied the original code, and carefully make the changes there as well.

To help you with the practice of code reuse, many modern programming languages provide structural support. As examples, languages like Java and C++ have *templates* or *generics*, and languages like Python have *duck-typing*. These support passing any object to a function as long as it supports the operations and methods used by the function.

Sometimes, the task isn't *completely* the same in all of the different situations, but you might be able to design a code segment which addresses multiple, highly-related, tasks which would contain much of the same underlying logic. This can be done by using flags or additional parameters to allow you to craft a single block of code that can accomplish the slightly different tasks.

Consider how we could think of the tasks of rotating one image whose layout is a square, another image whose layout is a tall rectangle, and another image whose layout is a wide rectangle as three different tasks. However, there would be much overlap in logic and code, so it would be strategically wise to create a single code segment that could handle all three slightly different scenarios.

Let's look at a brief example in practice:

```
class Particle {
   double x;
   double y;
   double vx;
   double vy;

   public Particle(double xIn, double yIn,
                   doublevxIn, double vyIN) {
      x = xIn; y = yIn;
      vx = vxIn; vy = vyIn;
   }

   double distance(double originX,
                   double originY) {
      double dx = x - originX;
      double dy = y - originY;
      return java.lang.Math.sqrt( dx*dx + dy*dy );
   }

   double speed() {
      return java.lang.Math.sqrt( vx*vx + vy*vy );
   }
}
```

While this is not a *worst* case example of code duplication, it is illustrative of the general issue while being brief.

With some thought about the similarities and differences between the two tasks, we could instead implement things as:

```
class Particle {
   double x;
   double y;
   double vx;
   double vy;

   public Particle(double xIn, double yIn,
                   double vxIn, double vyIN) {
```

```
      x = xIn; y = yIn;
      vx = vxIn; vy = vyIn;
   }

   private double magnitude(double a,
                            double b) {
      return java.lang.Math.sqrt( a*a + b*b );
   }

   double distance(double originX,
                   double originY) {
      return magnitude( x-originX, y-originY );
   }

   double speed() {
      return magnitude( vx, vy );
   }
}
```

Imagine a scenario where we had a bug or inefficiency in our approach to finding the magnitude by taking the square root of the squares. Upon discovering that issue, with the second approach above we would only have to fix it in one section of code.

Refactoring

When implementing an idea, many have a general tendency to start working along one line of thought, and sticking with that initial approach regardless of how difficult things become. There is also a tendency to try to "save" existing code by adding in more code to fix things or to address unanticipated scenarios. This can lead to what is sometimes called "spaghetti code" (an expression for lots of logical threads and patches all jumbled together in a tangled mess). It might also express itself through the use of variables whose names are not meaningful (because they were part of a hastily-done fix and not carefully thought out, such as `a` or `flag3`). Issues such as these can lead to code that is difficult to debug, optimize, and sustain over time (either by the original programmer or by those who follow). Even in the scope of a student assignment, these are things to avoid as they can lead to such difficulties even in a short time span.

Avoid or breaking this habit might take work, but it is definitely worth it overall.
The value in this can be in the short-term (you spend less time and effort tracking down issues and tracing through your own spaghetti), or in the medium or longer terms (returning to your own code days or weeks later, or coming to someone else's code to alter functionality or to see how a task had previously been accomplished). If the author or maintainer of code notices these sorts of things are happening, it is considered good practice to put in the effort to *refactor* that code segment. Often, this refactoring will make further development more efficient and less error-prone (more than paying off the refactoring time), while also making future maintenance easier.

Refactoring can be applied in several ways. In the simplest, it could take the form of renaming variables to have contextual meaning. This is common enough that some IDEs have a quick way to rename all instances of a variable in a given scope. Other small but useful forms of refactoring supported by some IDEs include renaming methods, updating method signatures, and even extracting a code segment into a new function or method. Example of refactoring that cannot be automated are the restructuring of a long chain of conditional statements, or the consolidation of redundant tests. Some longer and more specific refactoring approaches follow below.

Collecting Common Implementations

Closely related to avoiding code duplication, a frequent target of refactoring is extracting a common implementation from multiple classes or modules into a new one that the original classes can use as a service. The example code in the previous section provides a good example of an opportunity to do this. Imagine that the Particle class as well as several others

Reminder:

```
class Particle {
   double x;
   double y;
   double vx;
   double vy;

   public Particle(double xIn, double yIn,
                   double vxIn, double vyIN) {
      x = xIn; y = yIn;
      vx = vxIn; vy = vyIn;
   }

   private double magnitude(double a,
                            double b) {
      return java.lang.Math.sqrt( a*a + b*b );
   }

   double distance(double originX,
                   double originY) {
      return magnitude( x-originX, y-originY );
   }

   double speed() {
      return magnitude( vx, vy );
   }
}
```

Imagine that the Particle class as well as several others classes in a project have pairs of fields that are really only meaningful as closely-connected pairs, thinking of them as 2D points or 2D vectors.

We could make this cleaner across multiple classes by extracting a new `Vector2D` utility class to encapsulate the logically related pair of values, as well as provide common/standard operations that can be performed upon them.

```
class Vector2D {
   double x;
   double y;

   Vector2D(double a, double b) {
      x = a; y = b;
   }

   Vector2D difference(Vector2D other) {
      return new Vector2D(x-other.x,y-other.y);
   }

   double length() {
      return java.lang.Math.sqrt( x*x + y*y );
   }
   double magnitude() { return length(); }
}

class Particle {
   Vector2D position;
   Vector2D velocity;

   public Particle(double xIn, double yIn,
                   double vxIn, double vyIN) {
      position = new Vector2D(xIn, yIn);
      velocity = new Vector2D(vxIn, vyIn);
   }

   double distance(Vector2D origin) {
      return position.difference(origin).length();
   }

   double speed() { return velocity.length(); }
}
```

Now we have a simple class that we can use wherever we might need to store and interact with a 2D vector. We provide both `magnitude()` and `length()`, even though they are identical, because the former is a commonly used term, so some users of this class might expect it to exist.

We could, perhaps, even extend this to creating a general-purpose Vector class on up to three (or more) dimensions, if we determine that our overall project needs it. If done well, with multiple constructors representing 2D, 3D (etc.) spaces, our `Particle` implementation above would not need to change at all other than in the name of the class being used, which as has been mentioned is a refactoring ability that is often made easy by an IDE.

Reducing Code Complexity

An issue that can arise is that initial software design and the maintenance of that software over time can present different challenges in practice. When planning out your code, and implementing that plan, you should have a solid idea of what is needed and how things should work. However, it is likely going to be difficult to envision all possible future situations. In practice, there are also situations where you discover mid-way through your implementation that some assumption or choice made initially is just not going to work well. These scenarios might be more likely on a class project where you are not undertaking a full software engineering cycle, but they can happen in industry as well, particularly in long-lived software.

The first step is to identify when your code is suboptimally organized.
A good clue can be when adding each new piece of logic becomes increasingly difficult. You might find yourself having to add special cases, trying to work around your current design. You also might notice a proliferation of small classes that are similar, but not enough to combine.

The next step, once you've identified that a code complexity problem likely exists, is to take a step back, and reassess your design. How does reality differ from your initial expectations? If you were to rewrite everything from scratch, what would you do differently? This might come to you quickly, or you might have to spend a few days sorting things out.

- Don't shy away from drawing diagrams or writing out some quick pseudocode to help you visualize and conceptualize what is going on.
- Don't shy away from identifying blocks of code to rewrite once you have a new understanding of the big picture.

Once you identify where you should go with your code, it then becomes time to figure out the best way to get there. You don't want to rewrite a large section of your code if you don't have to, but at times it will be the most efficient and least error-prone approach for a section. Hopefully, much of your logic can be moved around or only slightly modified, but again it is wise to not make that a goal which gets in the way of consolidating what might have become spaghetti and numerous special cases.

The most important things to consider tend to be:

1. Will it take more time to modify the code that I have written into the new structure, or to rewrite it using the new understanding that I have built?
2. What is the cost of a refactoring now as compared to leaving things as they are, in the context of future development and maintenance costs from the current state of the code?

There are no easy answers that we can provide here, and of course any estimates and decisions you make now are subject to the same caveats as your initial ones as the project continues. In reality, it is quite possible that over the course of a lengthy project that you might find yourself needing to apply different types of refactoring to your code multiple times. However, with experience you hopefully find that you need two or fewer refactorings in the lifetime of a single project.

Testing Your Code

Once you have written your code, how do you know that it does what you want it to do in all possible situations? You have undoubtedly been introduced to the concept of testing, but you might not know how to undertake it effectively, thoroughly, and in a sustainable manner.

Unit Testing

One common approach is Unit Testing. The idea behind unit testing is to thoroughly test the individual *units* of code that make up a full project. This starts from elements such as functions or methods, and builds up from there. Given well-defined input descriptions, you can verify that your code produces the correct output on any input that could be sent. This might also include testing that so-called "invalid" input (that might be syntactically, but not logically, valid) does not produce undesired results (bad output or a crash). Note that an application of unit testing with which you might be most familiar, an autograder, is *not* really doing unit testing; it's using the technology and approaches of unit testing to accomplish something else.

So, how do you apply unit testing to your code? Let's consider a Python module with a single function, which we'll call `divide.py`:

```
def divide(numerator, denominator):
   return numerator/denominator
```

We are going to use the `unittest` package to unit test the correctness of this code. The typical way to do this is to add the unit testing as the behavior if you try to run the module as a stand-alone script (eg, `python3 divide.py`). We would then write `divide.py` as

```
def divide(numerator, denominator):
   return numerator/denominator

if __name__ == '__main__':
   import unittest

   class DivideTest(unittest.TestCase):
      def test_denom1(self):
         self.assertEqual(divide(3,1), 3)

      def test_denom2(self):
```

```
        self.assertEqual(divide(4,2), 2)
        self.assertEqual(divide(5,2), 2.5)

   unittest.main()
```

Both tests (`test_denom1` and `test_denom2`) are what are called "happy path" tests. That is to say, they are testing to confirm that the code works properly on input scenarios that would be considered valid (no exceptions or other error conditions involved). We could, in fact, have combined these into a single test, and whether you do this or not is largely a matter of style. By having multiple small tests where each contains limited scenarios, reports about those tests will make it very clear which individual scenarios have failed. By combining multiple scenarios into larger tests, if the test fails, you might not know which scenario was the cause, and/or might not have had the later scenarios tested at all.

It is also considered good style to have meaningful test names, just as it is to have meaningful variable and function/method names. If you look at the details, *test_denom1* is testing a scenario where the denominator passed in has the value 1. It can also be useful to incorporate comments if a test function contains multiple individual tests, or if a meaningful name would be too long.

Let's make this a little more advanced by not restricting ourselves to "happy path" tests though, by adding a test where we attempt to divide by 0:

```
def divide(numerator, denominator):
   return numerator/denominator

if __name__ == '__main__':
   import unittest

   class DivideTest(unittest.TestCase):
      def test_denom1(self):
         self.assertEqual(divide(3,1), 3)

      def test_denom2(self):
         self.assertEqual(divide(4,2), 2)
         self.assertEqual(divide(5,2), 2.5)

      def test_denom0(self):
         self.assertEqual(divide(1,0), 0)

   unittest.main()
```

This illustrates a logically invalid attempt at division which is syntactically valid, and produces the following:

```
======================================================================
ERROR: test_denom0 (__main__.DivideTest.test_denom0)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/mike/projects/coding-best-practices/chapters/dividetest.py", line 18, in test_deno
```

```
    self.assertEqual(divide(1,0), 0)
                     ^^^^^^^^^^^
  File "/home/mike/projects/coding-best-practices/chapters/dividetest.py", line 4, in divide
    return numerator/denominator
           ~~~~~~~~~^~~~~~~~~~~
ZeroDivisionError: division by zero

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (errors=1)
```

This demonstrates what will happen if a program attempts to divide by 0 with no protection in its code. We can "fix" this in one of two ways:

1. Add `self.assertRaises(ZeroDivisionError)` in the test to reflect what we might actually have meant to test: that the exception we expect code using this function to catch is the one that is thrown, or
2. Change `divide()` to test for a 0 denominator, and handle the error appropriately itself in some way.

Either of these could be fine in this case (we would likely raise that error in `divide()`). In most cases, you would want to fix the code to handle errors more gracefully, but it is important to not create more potential problems down the road by creating invalid output. For example, having the divide function return a 0 in this case might be seen as more graceful, but it would be incorrect mathematically. This could lead to harder-to-find bugs that would have been found sooner if the function threw an error on the improper call.

Unexpected exceptions are not the only error conditions, of course. Many times, we want to ensure an expected output, but a bug in the code produces something different. To show you what this looks like, consider this incorrect, but illustrative, modification to `test_denom2`:

```
    def test_denom2(self):
        self.assertEqual(divide(4,2), 1)
        self.assertEqual(divide(5,2), 2.5)
```

Running this produces:

```
======================================================================
FAIL: test_denom2 (__main__.DivideTest.test_denom2)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/mmarsh/classes/educational_materials/coding-best-practices/chapters/./dividetest.py", li
    self.assertEqual(divide(4,2), 1)
    ~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^
AssertionError: 2.0 != 1
```

This introduces the core concept of a unit test. Next, we explore the application of the core concept in practice.

How to Know What Tests to Write?

While it might be tempting to "spot check" on a small number of anticipated scenarios, generally you want to test all of the logical paths your code could take (we'll discuss this in greater detail soon), and validate that your assumptions hold. While good software engineering has the development of unit tests along with (or even ahead of) the development of code, in practice (especially in class projects) often unit tests are developed over time "on demand" as bugs are discovered as an approach to debugging. The premise is that once you know there is a bug, you can set out to write a test which triggers that bug to help find and fix the cause.

For example, perhaps the author of the divide function initially did not think of the scenario of it being called with a denominator of 0. Once a program crashes due to that scenario, they might then add an appropriate test as they track down the root cause. Under this general approach, the situation in which you might usually find yourself is that some valid set of inputs is producing an incorrect output, potentially reported to you by a user of your code. Clearly, you want to capture this behavior, so you write a unit test that takes inputs that cause incorrect output, and assert the *correct* output.

At this point, you have a unit test that will *fail* given the bug. Now your job is to fix the code, and verify it with this new unit test, which should then succeed (as should all previous unit tests). Now you have not just a test for an old bug, but a *regression* test that will fail if the bug is reintroduced by subsequent changes.

However, this will not lead to the creation of unit tests that test all of the logical paths your code could take. A different approach, that can better facilitate this level of testing, is to create unit tests based on a module's requirement specifications, and then use those tests as a means of verifying that the behavior of the module matches the spec. In general, the bulk of your unit testing should follow this approach, of validating that your code implements the specification. Reactive unit testing, where you create the tests on-the-fly due to error reports, should be the exception, not the rule. Good unit testing catches bugs before they enter production (or submitted) code, so reactive unit tests reveal a failure in testing before your software is released.

Mocking

Sometimes your unit tests require another class, whether to provide inputs, generate outputs, or as parameters to a method. Since this other class is not what you're trying to test, might take a while to run, and might have its own bugs, you generally don't want to use it directly. That's why most unit testing frameworks have the ability to *mock* a class (as in a mock-up of a class, not ridiculing it).

In Java, three popular mocking packages are EasyMock, Mockito, and JMockit. All three of these allow you to define an instance of what is essentially a subclass of the class you're mocking, where calling a particular method returns a pre-determined value, or sequence of values. Python's `unittest` module has a `mock` submodule. There are other unit testing frameworks for Python, which also have their own mocking modules.

We are not going to go into detail about mocking, but we encourage you to look up the documentation for various frameworks to get an idea of how they work. It does not matter which one you start with — wherever you end up coding professionally, they will likely have a particular testing framework, including mocking, with which you will have to familiarize yourself. Familiarity with any of these will help you pick up another reasonably quickly.

Code Coverage

Previously in this chapter, we mentioned that for full and proper testing your unit tests should cover all of your code's logical paths. This might be done by writing out many explicit tests, or even by writing small programs within your unit tests that generate scenarios for the various paths.

As an example of the latter, perhaps when testing a sorting function or method, you write a small program within a test that generates 1,000 lists of different sizes, populating some with random data and others with patterned data, then sorting each and having code that tests whether the supposedly-sorted lists are in fact sorted. This is not a proof of correctness, but is almost certainly going to be better than hardcoding some arrays and the confirmation tests.

Manually keeping track of all the paths which your tests have covered can be difficult, since with complex enough modules it can be easy to overlook some possible paths. This is where automated tools for tracking *code coverage* come in very handy.

Code coverage tracking tools can use the *abstract syntax tree* (AST) the compiler (or interpreter) builds of your code to report on thoroughness of the testing. For our purposes, we ask you to not worry about what the AST is or does, but rather just to know that it breaks your code down into individual blocks. As your unit tests run, the code coverage tool keeps track of which blocks in the AST have been executed. Once all unit tests have run, the code coverage tool gives you a report informing you of the fraction of blocks and lines that have been *covered* (run by) your unit tests. This report is often broken down by class, method, and function.

What makes this approach even more straight-forward in practice is that your IDE probably has code coverage built into it (or available as a plugin), and will show you the coverage details in your source code.

Typically, a line will be highlighted in:

- green if it was tested "completely" (for some set of inputs, not exhaustively),
- red if was not tested *at all*, or
- orange or yellow if it was *partially* tested.

What does "partially" tested mean? Typically, this means you have an `if` or `while` statement where the conditional is tested for one value (true or false), but not the other.

While there is no single standard of what "enough" coverage is, most professional software development shops will have their own internal code coverage requirements. Some example of what these might be include:

- All non-trivial methods must be 100% covered.
- The codebase must be at least 80% covered.
- The codebase must be 100% covered.

The last of these might be generally seen as overkill, but in regulated industries it might be required. Targeting 100% coverage could result in odd-looking unit tests where some setter is called, followed immediately by the corresponding getter. These are not particularly *helpful* tests, but if 100% code coverage is mandated, they're not uncommon. Of course, a better approach if 100% coverage is not required might be to use the setters and getters more organically in other tests.

Something to which care should be given is making sure that if working towards a certain percentage goal, you are focusing on including good, realistic tests as you aim for that goal. You want to avoid falling into a false sense of security by hitting a target percentage by adding in generally unhelpful tests to pad things out.

Even when 100% coverage is not mandated, ensuring complete code coverage is a great way to make sure you're testing your code thoroughly. Even in academia, in upper-level courses where the logic can get very complicated, students often write methods with many branches, or branches on very complex conditionals. When testing their code, students might assemble inputs that follow some pattern (for example, just using monotonic sequences), and check that the outputs match their expectations. These tests can often miss branches or sub-expressions in a conditional, and using code coverage tools is a great way to identify when this has happened on your own, rather than waiting until secret tests or hand-run grading discovers the testing gaps by finding errors.

Regression and Integration Testing

While unit testing is intended to test individual code units as we develop, there are two other philosophies to keep in mind.

One core idea behind *regression testing* is that any time part of a module is modified, it is beneficial to retest everything in that module, to check that no change had a side effect on another part of the module. Basically, test that any new features or fixes to existing features do not cause an error in another existing feature.

Sometimes this is simply a matter of re-running all existing unit tests. Sometimes, it also includes updating existing tests to incorporate appropriate new features into old tests.

Second, in addition to making sure each unit works as expected on its own, we often want to test how different units interact with one another. This is referred to as *integration testing*, and more closely matches what autograders might do.

The idea behind integration testing is to test a complex collection of code as a whole. For example, in addition to testing a single method of a class, we also construct an instance of the class and perform a number of operations on it, verifying the resulting state of the object as we go.

There are two categories of integration testing:

- *Internal* integration testing, which is essentially what is described above. Here, the class (or classes) are tested in isolation, but the various methods are tested interacting with each other.
- *External* integration testing, which integrates the code being tested with other programs or systems. Testing code that uses a variety of other classes without mocking, or testing code as it interacts with a database or web server are good examples of this.

It is also important to consider that the programming language and/or general approach of using encapsulation or not can have an impact on the use of both regression and integration testing.

Imagine how the use of a global variable or a singleton pattern might impact the ways in which changes in one place could cause a bug in another.

Test-Driven Development

The final topic we will cover is *test-driven development*. This is a way of developing code that inverts the previously-mentioned process of

1. Write the code.
2. Test the code.

Instead, in test-driven development, the process is:

1. Write tests that specify given inputs and expected outputs, taken from the project specifications.
2. Implement the project module by module, until all of the tests pass.

What does this look like? Let's consider:

```
def sortMyList(input_list):
   pass
```

Initially, we have a method that "does" nothing, but we know what we expect it to do: It should return a list with the elements of `input_list` in sorted order. We make use of a null body so it can compile, even though it provides no functionality as of yet, acting as a placeholder for the eventual implementation. We could still write unit tests for when the implementation is ready.

Some example tests follow:

```
if __name__ == '__main__':
   import unittest

   class SortingTest(unittest.TestCase):
      def test_empty(self):
         self.assertEqual(sortMyList([]),
                          [])

      def test_already_ordered(self):
         self.assertEqual(sortMyList([1,2,3]),
                          [1,2,3])

      def test_reversed_order(self):
         self.assertEqual(sortMyList([3,2,1]),
                          [1,2,3])

      def test_mixed_order(self):
         self.assertEqual(sortMyList([2,1,3]]),
                          [1,2,3])

   unittest.main()
```

All of these tests will fail initially, but as we develop our `sortMyList` function, possibly iteratively, the goal is that they will all pass in the end.

For example, our first iteration of the `sortMyList` function might be:

```
def sortMyList(input_list):
   return []
```

which would make `test_empty()` pass, though the others would still fail.
While a next iteration might be returning `input_list`, which would make `test_empty()` and `test_in_order()` pass, that is not a realistic iteration for how this function would be developed, and reveals a potential behavior this development approach could enable.

Another noteworthy point is that while the last two tests will likely both pass with the next true iteration of this very simple example, they do not represent a thorough testing of this function. Consider the following test function being added to `SortingTest`:

```
   def test_random_order(self):
      n = 1000
      values = []
      for i in range(1, n):
         # Populate the list with n random values.
         values.append(
            random.randint(-8192, 8192))

      # Call your sorting function.
      result = sortMyList(values)

      # Verify that the list is the same size
      # as it was and now sorted.
      sortedLen = len(result)
      self.assertEqual(n,sortedLen+1)
      for j in range(sortedLen-1):
         self.assertTrue(result[j] <= result[j+1])
```

This is a more thorough test of various properties a list of values might have. A better test might also confirm that each element in the original list still exists in the new one. An even more thorough test would be to add an outer loop to test on many randomly-generated lists.

In real-world systems, your tests will reflect the API your team is responsible for delivering. It is even possible that tests would be provided as part of the operational acceptance test criteria provided by a client or written into a contract.

Even when utilizing test-driven development, you will likely *also* add more tests as you continue to develop for reasons already mentioned, such as to address bugs, aim for code coverage, or address added code complexity concerns.