CHAPTER

6

Testing and Debugging

OUTLINE

Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fand er sich in seinem Bett zu einem ungeheuren Ungeziefer verwandelt.
One morning, as Gregor Samsa was waking up from anxious dreams, he discovered that in bed he had been changed into a monstrous verminous bug.

–Franz Kafka,* The Metamorphosis, *as translated by Ian Johnston

CHAPTER POINTS

- Strong testing of code is as important as the ability to write codes.
- Finding problems inside a code, i.e., debugging, is an exercise in intuition as well as inference from how the code fails its tests.
- Assertions can help isolate problems inside of large codes, especially when functions call other functions.
- It is possible to catch errors and handle them in Python using `try-except` blocks.

6.1 TESTING YOUR CODE

For any code it is incumbent on the programmer to make sure that code accomplishes the desired task. For instance, if a function is supposed to return the largest item in a list, the programmer should not deliver that function to a user unless it has been shown, for a variety of test cases, that it indeed gives the correct answer. This needs to be done anytime one writes code where any the following apply:

DOI: 10.1016/B978-0-12-812253-2.00007-8

1. The code will be used by somebody else;
2. The code will be used to be input for another piece of code (e.g., a function that calls another function);
3. The code will be used to make a decisions, or
4. The code will be turned in as a class assignment.

Writing tests for a code, and demonstrating that the code passes the tests, is essential to giving your code credibility. Consider the NumPy `dot` function. We have not questioned if it will give us the correct answer. To some degree we just believed that the super smart people at NumPy Inc. would not release faulty code. This implicit belief is correct, but not because of the corporate imprimatur of the developers. In fact NumPy is the product of a community of hundreds of developers, and millions of users. The developers have built tests into the source code that take the known value of the dot product, and compare it to the value returned by `np.dot`. Users of the code can view these tests inside the NumPy source code (i.e., the code that gets executed when NumPy is used). This test, and thousands like it, are run on a regular basis to make sure the code is behaving properly. Furthermore, the user community acts as a secondary test bed. Users of NumPy can report problems with the code to the developers, along with a minimal example demonstrating the flaw, and these will be resolved.

Taken together, we do not believe that `np.dot` works because some imperious entity tells us; it is because there is evidence of testing, and a user community to help identify errors that might have been missed. The ability to see which tests are used is a clear benefit of an open-source product such as NumPy. Were we not able to view the source code, we would have to rely on the quality of the software vendor, which can vary wildly from vendor to vendor.

In many of our cases, we will need to demonstrate that a piece of code we develop to make a numerical evaluation is giving us the correct answer. To do this we can often use exact solutions to simple problems, the known limits of a system, or look at the overall behavior of the solution compared with some theoretical behavior. It is important that we have more than a single test because a single test could be passed for the wrong reasons.

A simple example of a test being passed for the wrong reasons is a function that is supposed to compute the probability for an event. If the function returns a value of 0.5, no matter the inputs, it would pass a test checking whether the probability is in the interval [0, 1]. Furthermore, it would pass a test designed to check if the function correctly computes a value of 0.5 for a particular set of inputs. Clearly, there are many tests that this function would not pass, but selecting just a few tests may not cover enough of the function's intended use.

A more detailed example of testing, and why multiple tests are needed is presented below. Consider the following code to compute the multiplication factor (k-effective, k_{eff}) for a system of where nuclear fission is present, under the approximation of a one-group, bare reactor [7]. The formula for the multiplication factor is

$$k_{\text{eff}} = \frac{k_\infty}{1 + L^2 B_g^2},$$

where

$$k_\infty = \frac{\nu \Sigma_{\text{f}}}{\Sigma_{\text{a}}},$$

and

$$L^2 = \frac{D}{\Sigma_a} = \frac{1}{3\Sigma_{tr}\Sigma_a},$$

and if the reactor is a slab we have

$$B_g^2 = \left(\frac{\pi}{X}\right)^2.$$

The notation here is standard: Σ_a, Σ_{tr}, and Σ_f are the macroscopic absorption, transport, and fission cross-sections with units of inverse length, ν is the mean number of neutrons per fission, and D is the diffusion coefficient with units of length. The thickness of the slab is X.

The code to compute k-effective is below:

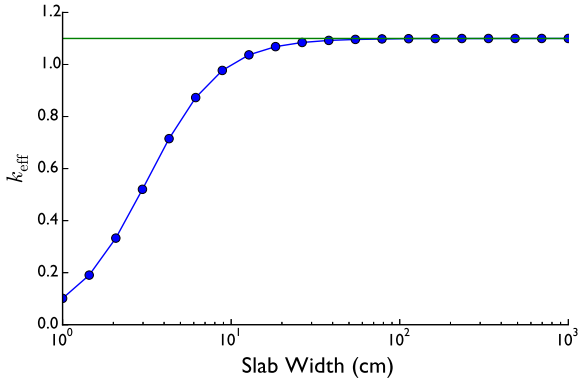
```
In [1]: import numpy as np
        def k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef):
            """ Computes the eigenvalue
                (k-effective) for a slab reactor

            Args:
                slab_length: the length of the slab
                nuSigma_f: value of nu * the macro. fission x-section
                Sigma_a: value of the macro. absorption x-section
                Diff_coef: the diffusion coefficient

            Returns:
                The value of k-effective
            """
            k_infinity = nuSigma_f / Sigma_a #k-infinity
            L = Diff_coef/Sigma_a #Diffusion length
            B = np.pi/slab_length #geometric buckling.
            k = k_infinity/(1+L**2 * B**2)
            return k
```

To make sure that our function is correct, there are several tests we could run to make sure that the function behaves the way that it should. A simple one would check that $k \to k_\infty$ as the slab size goes to infinity. The code below does this.

```
In [2]: import matplotlib.pyplot as plt
        #20 points from 10^0 to 10^3
        lengths = np.logspace(0,3,20)
        nuSigma_f = 1.1
        Sigma_a = 1.0
        Diff_coef = 1.0
        k_vector = k_effective(lengths, nuSigma_f,
                               Sigma_a, Diff_coef)
        plt.semilogx(lengths,k_vector,'o-')
        plt.semilogx(lengths,nuSigma_f/Sigma_a*np.ones(20))
        plt.xlabel('Slab Width (cm)')
        plt.ylabel('$k_\mathrm{eff}$')
        plt.show()
```



Viewing the resulting figure, we can see that the result goes to k_∞ as the slab size goes to infinity. Next, we test a particular case where we know the answer. If $\Sigma_a = D$ and $X = \pi$, the system will have k-effective equal to $k_\infty/2$. Here is that test:

```
In [3]: test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                             Sigma_a = 1, Diff_coef = 1)
        if np.fabs(test_k - 0.5) < 1.0e-8:
            print("The test passed, k =",test_k)
        else:
            print("Test failed, you should probably",
                  "fix the code, k =", test_k)
```

```
The test passed, k = 0.5
```

You might be tempted to think that our code is fine, but it turns out that there is a problem (or bug) in the code. Let's try the test again, this time with $D = 2\Sigma_a$. The answer should be one-third of k_∞.

```
In [4]: test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                             Sigma_a = 1, Diff_coef = 2)
        kinf = 1.0
        if np.fabs(test_k - 1.0/3.0) < 1.0e-8:
            print("The test passed, k =",test_k)
        else:
            print("Test failed, you should probably",
                  "fix the code, k =", test_k)
```

```
Test failed, you should probably fix the code, k = 0.2
```

It is clear that our code has managed to pass some tests, and has failed the third test we tried. Now begins our hunt to find the bug that is giving us the error.

6.2 DEBUGGING

Now that we have identified that our code has a bug, the process of finding that error is called debugging. Given that we ran three tests, and two of them passed we can use how the tests are different to identify where the bug might be.

The first test checked that as `slab_length` went to infinity, $k \to k_\infty$. Looking at the equations we can see that as the slab length goes to infinity, B_g goes to 0. Therefore, this test just checks that k_∞ is calculated properly, and indeed it is. The second test was designed so that B_g would be 1 and L^2 would be 1. In this instance things behaved like they should. When we switched things so that L^2 would be 2 and B_g would be 1, we had a failure. Therefore, the bug does not appear when $L^2 = 1$.

This is a covert bug (in that it is not apparent without a test). Take a moment to look at the function definition, and the equations it was trying to implement to see if you can identify the bug before moving on.

The code below has the bug fixed. It is a very small change, but often the hardest bugs to find are the ones that require just a small correction.

```
In [5]: import numpy as np
        def k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef):
            """ Computes the eigenvalue
                (k-effective) for a slab reactor

            Args:
                slab_length: the length of the slab
                nuSigma_f: value of nu * the macro. fission x-section
                Sigma_a: value of the macro. absorption x-section
                Diff_coef: the diffusion coefficient

            Returns:
                The value of k-effective
            """
             k_infinity = nuSigma_f / Sigma_a #k-infinity
             L2 = Diff_coef/Sigma_a #Diffusion length squared
             B = np.pi/slab_length #geometric buckling.
             k = k_infinity/(1+L2 * B**2)
             return k
```

Then, re-running the test from before, we get the correct answer.

```
In [6]:  test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                              Sigma_a = 1, Diff_coef = 2)
         kinf = 1.0
         if np.fabs(test_k - 1.0/3.0) < 1.0e-8:
             print("The test passed, k =",test_k)
         else:
            print("Test failed, you should probably",
                  "fix the code, k =", test_k)

The test passed, k = 0.3333333333333333
```

The bug in the code was that L^2 and L were mixed up. That is why the error did not show up when $L^2 = 1$ because in that case the values are the same.

In this example we see how tests can help us debug. By running several tests we can hone in on the error in the code: the first test demonstrated that k_∞ was correctly calculated, the second test showed that when $L = 1$, the code was correct. Note also that only one test is not sufficient, as we saw. This is why you want multiple tests for your code, you want to do more

than just make sure the code works in one limit or one case, you need to test several possible cases and see that the code works in all of them.

Debugging is a mindset. When looking at your code you need to be thinking critically about how the code works, and what the value of a variable is after each line, etc. Then you want to compare that with what the code is supposed to do. This is one way that comments can be important: they indicate the intension of the programmer and what the code *should* be doing. When comparing code to what it should be doing, a keen attention to detail is necessary because the error could be as small as a single character.

Unfortunately, there is no simple recipe for debugging code. Experience is an important ingredient because the errors that appear in code often repeat, and having made a mistake once will make it possible for the programmer to look for that mistake again in a similar piece of code. The value of experience becomes apparent when a novice programmer presents code with an error to an instructor or another expert. Sometimes without even looking at the code, the expert can identify the problem based on the described behavior. Such an occurrence can cause the novice programmer to despair that he or she "will never be that good", or some other self-defeating watchword. Typically, the expert can diagnose the problem so quickly because of that expert's past mistakes.

Despite not having a simple recipe, when debugging it can be useful to ask the following questions:

- How is the code failing?
- What is the code doing correctly?
- What pieces of the code are most likely to have an error?

Moreover, your best tool for debugging is the `print` function. When in doubt have the code print out what happens after every line and check that with your intuition/expectation. For example, if there is a mistake in a formula you could print out the result and compare it with a hand calculation.

BOX 6.1 LESSON LEARNED

The best debugging tool is the `print` function. When you want to figure out what is going on in your code, pepper it with print statements, like a chef preparing steak au poivre, to see what each line is doing.

There are other tools for debugging. If you use IDLE, Anaconda, or another integrated development environment (such as Visual Studio or XCode), there is a built-in debugger. One of the features a debugger allows you to do is set breakpoints. A breakpoint is a point in the code where execution stops, and you can enter commands interactively to see what the value of different variables are, for example. Also, once the code stops at a breakpoint you can step through the code line by line and see how the code is actually executing, e.g., is a particular `if` statement evaluating to true or false, or how many times does a loop execute. For more information about the debugger check your development environment's documentation.

6.3 ASSERTIONS

In life it pays to be assertive. The same is true in programming. With the `assert` statement, you can check assumptions that are embedded in your code using an `assert` statement. The `assert` statement takes an expression as input. If the expression evaluates to true, then the `assert` does nothing. However, if the expression evaluates to false, an error is thrown and the code stops executing. The benefit of the `assert` statement is that you can make the code stop dead in its tracks, if some assumption made is violated. Then, Python will tell you where the code stopped.

BOX 6.2 PYTHON PRINCIPLE

The `assert` statement is called using the syntax

```
assert expression
```

where `expression` is a python expression that evaluates to true or false. If `expression` evaluates to true, nothing happens. When `expression` evaluates to false, an error occurs and code execution stops. Python will indicate where in the code the `assert` statement failed. This type of error is called an `AssertionError`.

As an example, we use our `k_effective` function from before. Physically, all of the cross-sections in the model we implemented should be non-negative. However, the code will work even if the inputs are negative:

```
In [7]:  test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                              Sigma_a = -2, Diff_coef = 1)
         print("With negative Sigma_a, k =", test_k)

With negative Sigma_a, k = -1.0
```

The code worked, but the answer is non-sensical. We would like to tell the user that `k_effective` was called with an improper value. We could just change the help text to indicate that each input needs to be greater than zero. However, if we do this it will not prevent the user from running the code with negative cross-sections. Moreover, if the user does not respect the instruction for the function inputs, the function will behave strangely or give an error

```
In [8]:  test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                              Sigma_a = -1, Diff_coef = 1)
         print("With negative Sigma_a, k =", test_k)


---------------------------------------------------------------------
    ZeroDivisionError Traceback (most recent call last)

        <ipython-input-23-a6b005d701c7> in <module>()
```

```
    ----> 1 test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                                 Sigma_a = -1, Diff_coef = 1)
          2 print("With negative Sigma_a, k =", test_k)


        <ipython-input-21-2a945842eba5> in
        k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef)
         15     L2 = Diff_coef/Sigma_a #Diffusion length
         16     B = np.pi/slab_length #geometric buckling.
    ---> 17     k = k_infinity/(1+L2 * B**2)
         18     return k


        ZeroDivisionError: float division by zero
```

In such a case even though the function caller made a mistake, by passing in a negative cross-section, because the code failed in the function `k_effective`, the programmer of that function is likely to be blamed.

This is where the assert statement comes in. It can assure that the arguments to the function are what they should be. Therefore, the function can throw an error when it is called improperly. Below, we do this for `k_effective`; the docstring is not shown for brevity.

```
In [9]: import numpy as np
        def k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef):
            assert (slab_length > 0)
            assert (nuSigma_f > 0)
            assert (Sigma_a > 0)
            assert (Diff_coef > 0)
            k_infinity = nuSigma_f / Sigma_a #k-infinity
            L2 = Diff_coef/Sigma_a #Diffusion length
            B = np.pi/slab_length #geometric buckling.
            k = k_infinity/(1+L2 * B**2)
            return k
```

If we call this function with invalid parameters, it will give an error and indicate where the error occurred.

```
In [10]: test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                               Sigma_a = -1, Diff_coef = 1)
         print("With negative Sigma_a, k =", test_k)


---------------------------------------------------------------------
    AssertionError Traceback (most recent call last)

        <ipython-input-10-a6b005d701c7> in <module>()
    ----> 1 test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                                 Sigma_a = -1, Diff_coef = 1)
          2 print("With negative Sigma_a, k =", test_k)
```

```
      <ipython-input-9-b097a84a2036> in
      k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef)
       14     assert (slab_length > 0)
       15     assert (nuSigma_f > 0)
  ---> 16     assert (Sigma_a > 0)
       17     assert (Diff_coef > 0)
       18     k_infinity = nuSigma_f / Sigma_a #k-infinity


      AssertionError:
```

Notice that Python tells us which assertion failed so we know that the function call had a bad value of `Sigma_a`. The program still fails, but it tells us exactly why. Additionally, it indicates that the error is not in the function itself, but with the input parameters.

We can also use assertions to test that the code behaves the way we expect. In the case of this function, we know that k is in $[0, k_\infty]$. Therefore, we can embed that check in two uses of `assert`.

```
In [11]: import numpy as np
         def k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef):
             assert (slab_length > 0)
             assert (nuSigma_f > 0)
             assert (Sigma_a > 0)
             assert (Diff_coef > 0)
             k_infinity = nuSigma_f / Sigma_a #k-infinity
             L2 = Diff_coef/Sigma_a #Diffusion length
             B = np.pi/slab_length #geometric buckling.
             k = k_infinity/(1+L2 * B**2)
             assert k >= 0
             assert k <= k_infinity
             return k
```

These types of assert statements can help you debug later on down the road. This is especially true when you have functions being called by other functions. In these situations `assert` statements can assure that functions were called properly and help isolate where any problems lie.

6.4 ERROR HANDLING

There are times when you want to handle an error so that either the program can continue or print a useful error message before exiting. The complete topic of error handling is outside the scope of this work, and in the purview of writing production software, which we are not tackling. Nevertheless, error handling can make debugging and finding errors in code easier. Additionally, we can use error handling to help us execute tests of our code.

Error handling is often called exception handling. When a program is running, if it encounters an error it can raise an exception. The exception will give some indication of what the error is. In Python, you can place some code in a special block of code called a `try` block.

TABLE 6.1 Common Exceptions in Python

Exception	Meaning
`AssertionError`	The argument passed to an `assert` was `False`
`FloatingPointError`	An error happened in a floating point calculation
`KeyError`	A dictionary key that was not valid was used
`KeyboardInterrupt`	The user pressed `ctrl-c` to exit
`NameError`	An undefined variable was used to exit
`OverflowError`	A number too large was created/used
`RecursionError`	Function called itself too many times
`TypeError`	The wrong type was used in an expression
`ZeroDivisionError`	Attempted to divide by zero

After the `try` block, the types of exception to handle are listed using `except` blocks. Only the exceptions that are explicitly handled are caught.

One type of exception is the `ZeroDivisionError` that is raised when a number is divided by zero. First, we will look at an uncaught exception:

```
In [12]: z = 10.5/0


---------------------------------------------------------------------------

        ZeroDivisionError Traceback (most recent call last)

        <ipython-input-12-011b064d3b54> in <module>()
    ----> 1 z = 10.5/0


        ZeroDivisionError: float division by zero
```

To catch this exception and proceed in the code, we use a `try` block and `except` block as

```
In [13]: try:
             z = 10.5/0
         except ZeroDivisionError:
             print("You cannot divide by 0")

You cannot divide by 0
```

Notice that the `except` block takes the name of the exception as an argument (in this case `ZeroDivisionError`). A list of common exception types, and what they mean, are given in Table 6.1.

One thing that happens when you catch a raised exception, is that the program will continue on. This can be a useful feature, but often times you want to catch an exception, print a useful error message, and then have the program end. This can be done by adding a `raise` statement to the end of the `except` block. The `raise` statement tells Python to still fail due to the exception, despite the fact that we caught it. This changes the previous example by one line, but changes the output and forces the program to quit:

```
In [14]: try:
             z = 10.5/0
         except ZeroDivisionError:
             print("You cannot divide by 0, exiting")
             raise

You cannot divide by 0, exiting

---------------------------------------------------------------------------

        ZeroDivisionError Traceback (most recent call last)

        <ipython-input-14-525c0fef7adc> in <module>()
          1 try:
    ----> 2     z = 10.5/0
          3 except ZeroDivisionError:
          4     print("You cannot divide by 0, exiting")
          5     raise

        ZeroDivisionError: float division by zero
```

BOX 6.3 PYTHON PRINCIPLE

To handle errors, use a `try` block combined with `except` blocks. These have the form

```
try:
     [SomeCode]
except ExceptionA:
     [ExcepCodeA]
except ExceptionB:
     [ExcepCodeB]
...
except:
     [CatchAll]
```

The code in the `try` block, `[SomeCode]`, will be executed. If there is an exception raised while running the code in the `try` block that matches one of the parameters in the `except` blocks below (in this example, `ExpectionA`, and `ExceptionB`), the code in that `except` block will be executed and the code will continue. See Table 6.1 for a list of common exceptions. After executing the `except` block, the code will continue as normal.

The last `except` block may not have an exception type. This block will catch all other exceptions and execute its code, in this case `[CatchAll]`. This should be used with caution as unexpected exceptions may arise that causes errors later in the program.

The `raise` statement may be used to raise an exception, and if called inside an `except` block will raise the current exception type. This can be used to make Python quit the program due to an error.

For a more in depth example of error handling we return to the function `k_effective`. Previously, we added assert statements to make sure that the user gave the function all positive inputs. It would be more useful to the user, and for debugging, to have the code output what the values of the inputs were, if one of the `assert` calls fails. We do this in the code below (again without the docstring for brevity):

```
In [15]: import numpy as np
         def k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef):
             try:
                 assert (slab_length >0)
                 assert (nuSigma_f > 0)
                 assert (Sigma_a >0)
                 assert (Diff_coef > 0)
             except AssertionError:
                 print("Input Parameters are not all positive.")
                 print("slab_length =",slab_length)
                 print("nuSigma_f =",nuSigma_f)
                 print("Sigma_a =",Sigma_a)
                 print("Diff_coef =",Diff_coef)
                 raise
             except:
                 print("An unexpected error occurred when",
                       "checking the function parameters")
                 raise

             k_infinity = nuSigma_f / Sigma_a #k-infinity
             L2 = Diff_coef/Sigma_a #Diffusion length
             B = np.pi/slab_length #geometric buckling.
             k = k_infinity/(1+L2 * B**2)
             assert k >= 0
             assert k <= k_infinity
             return k
```

With this function, if it is passed a negative value, will raise an `AssertionError`. The code catches this error, prints out the input parameters to the user, and then exits by raising the exception. This functionality is demonstrated below.

```
In [16]: test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
                               Sigma_a = -1, Diff_coef = 1)

Input Parameters are not all positive.
slab_length = 3.141592653589793
nuSigma_f = 1
Sigma_a = -1
Diff_coef = 1

---------------------------------------------------------------

        AssertionError Traceback (most recent call last)

        <ipython-input-18-ee866f2992f9> in <module>()
          1 test_k = k_effective(slab_length = np.pi, nuSigma_f = 1,
    ----> 2                      Sigma_a = -1, Diff_coef = 1)


        <ipython-input-17-83b9328123be> in
        k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef)
          4         assert (slab_length >0)
          5         assert (nuSigma_f > 0)
```

```
    ----> 6         assert (Sigma_a >0)
          7         assert (Diff_coef > 0)
          8     except AssertionError:


        AssertionError:
```

Also, the function has a generic `except` statement that will catch any other errors in the `try` block. Because the `try` block involves comparison of numbers, if we pass a `string` as a parameter, there will be an error when checking if that parameter is greater than 0. This exception will be caught by the generic `except` statement, and the code prints out an error message, and then quits:

```
In [17]: test_k = k_effective(slab_length = "Pi", nuSigma_f = 1,
                              Sigma_a = -1, Diff_coef = 1)

An unexpected error occurred when checking the function parameters

------------------------------------------------------------

        TypeError Traceback (most recent call last)

        <ipython-input-22-6c226c0ff76e> in <module>()
          1 test_k = k_effective(slab_length = "Pi", nuSigma_f = 1,
    ----> 2                      Sigma_a = -1, Diff_coef = 1)


        <ipython-input-21-72247d51776d> in
        k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef)
          2 def k_effective(slab_length, nuSigma_f, Sigma_a, Diff_coef):
          3     try:
    ----> 4         assert (slab_length >0)
          5         assert (nuSigma_f > 0)
          6         assert (Sigma_a >0)


        TypeError: unorderable types: str() > int()
```

The exception is a `TypeError`, and was not anticipated, so the program tells the user something unexpected happened.

The `try-except` structure is a useful tool when writing functions that will be called by other functions, or when there is a chance for input parameters to have the wrong form. They go beyond `assert` statements to give the programmer the ability to tell the user what went wrong, and possibly fix the error, before either continuing or quitting.

The concepts of testing, debugging, assertions, and exceptions can be combined to make a capability to readily test and find bugs in code. In the programming exercises below, exceptions and assertions are combined to run a variety of tests on a piece of code, and to raise assertions when tests fail.

FURTHER READING

There are not many books the discuss the process of debugging, but a book by Butcher [8] does discuss how to build the mindset of a master debugger. For a more detailed, though perhaps too detailed for novices, description of exception handling in Python see the Python tutorial https://docs.python.org/3/tutorial/errors.html.

PROBLEMS

Short Exercises

6.1. Write an assert statement that guarantees a variable `n` is less than 100 and is positive.

6.2. Debug the following code:

```
for i in list_variable:
    list_variable = ['one',2,'III','quatro']
    print(i)
```

6.3. Write a function called `soft_equivalence` that takes as input 3 parameters: `a`, `b`, and `tol`. The function should return `True` if the absolute difference between `a` and `b` is less than `tol`, and return `False` otherwise. The parameter `tol` should be an optional parameter with a default value of 10^{-6}.

6.4. Create a list of tests you would perform to test a function `solve(A,b)` that returns the solution to the linear system $\mathbf{Ax} = \mathbf{b}$. You do not need to write any code, just describe the tests, either in equations and/or words.

Programming Projects

1. Test Function for k-Eigenvalue

In this problem you will create a test function that performs all of the tests we performed above on the k-eigenvalue function `k_effective`. Create a function named `test`, that takes no input parameters. This function needs to execute the three tests the we developed for the `k_effective` function: the infinite medium case and the two particular value cases. Each test should raise an assertion error if the test fails to be within some tolerance, appropriately defined by you, of the correct value. If an assertion error is raised, the error needs to be caught, and which test(s) failed should be printed to the screen. The function should return either `True`, if all of the tests passed, and `False`, if any test failed.

Demonstrate that you test function works on the `k_effective` as we defined it correctly above, and demonstrate that it will catch an error in the calculation, if a bug is inserted.