1

GETTING STARTED WITH DEBUGGING

The first step in the debugging process is writing buggy code. We'll start there, but then we'll quickly move on to what comes next: resolving bugs. We'll discuss the wide variety of errors Python coders commonly encounter, including errors that Python won't catch. We'll talk about how to find the locations of bugs, how to read traceback messages, and how to write code that handles bugs. Finally, we'll discuss the mindset that all good debuggers need to have: beginning with the end in mind. Let's get started!

Resolving a Bug

We'll start with code that's not buggy. The following code defines a list and prints out an element of the list:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci[5])
```

Our list, `fibonacci`, contains the first eight numbers in the *Fibonacci sequence*, a sequence where each number is found by taking the sum of the previous two numbers. We then printed out the element with index 5.

You should see an output of `8`. So far, there are no bugs—just simple Python code that runs smoothly.

Next, let's introduce a bug into our code. Suppose that our intention is to find a Fibonacci number larger than 8, so we attempt to access an element of `fibonacci` with a higher index than the one we accessed before:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci[10])
```

Listing 1-1: Buggy code

We tried to print the element of our `fibonacci` list with index 10. But there's a problem: the `fibonacci` list only has eight elements; there is no element with index 10. If you save these two lines as a Python script called *ch1.py*, and then run the script, you should get the following output:

```
Traceback (most recent call last):
  File "ch1.py", line 2, in <module>
    print(fibonacci[10])
          ~~~~~~~~~^^^^
IndexError: list index out of range
```

Listing 1-2: Traceback output

This message is called a *traceback*, and we will discuss every part of it later in the chapter. For now, let's focus on the most important part of every traceback: the last line. You can see that it says `IndexError: list index out of range`. This is telling us what we already knew: that the list index we tried to access (10) was out of the range of the indices that exist in the `fibonacci` list. We tried to access something that didn't exist, and this caused Python to output an error message instead of the larger Fibonacci number we were hoping to see.

Since this error is related to list indices, it's called an *IndexError*. In the programming world, the common parlance is that this code *threw* an IndexError and that Python *caught* the error. In Python, there are more than a dozen main types of errors, and as soon as Python catches any one of them, it will stop running your code, displaying a traceback message instead.

If you spend enough time writing Python code, you will encounter this and many other types of errors. You shouldn't feel like you're a bad coder if your code often throws errors. Even the most talented coders write buggy code every day. As you get familiar with each type of error, you can learn strategies for dealing with each of them. We will go over many such strategies in this book.

When we write Python code, like the code in Listing 1-1, we almost never intend for the output to be an error like the IndexError we saw in Listing 1-2. Anything in code that causes the code's actual output to be different from the code's intended output is called a *bug*. The process of finding and resolv-

ing bugs is called *debugging*. For our first debugging task, let's resolve the error in Listing 1-1.

There's a straightforward way to resolve this particular `IndexError`. Remember that our intention Listing 1-1 was to get a Fibonacci number larger than 8. We got an error because we tried to access a list element that didn't exist. We can resolve the error and accomplish our original intention by making sure that our code accesses a list element that corresponds to a large Fibonacci number and **also** a list index that actually exists in the `fibonacci` list:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci[7])
```

Listing 1-3: Accessing a list element that actually exists

Here, we accessed the list element with index 7, the highest index in our list. When you run this code, you won't see any traceback message about any error. Instead, you will see the output 21, the last element in our list. Since the actual output of our code matches what we want the output to be, the code is no longer buggy—we have succeeded at debugging.

Commonly Encountered Errors

In the previous section, we encountered a particular type of error called an `IndexError`. It's important to understand IndexErrors, but there are many other types of error you may encounter when writing Python code. Let's go over some of the most common ones.

Syntax Errors

Take a look at the following lines of code:

```
my_string_1 = "With foes ahead, behind us dread,\nBeneath the sky shall be our bed,"
my_string_2 = 'Until at last our toil be passed,\nOur journey done, our errand sped.'
```

This code is simple: all it does is define two strings. You should notice that the first string is enclosed by double quotes (`"`), while the second string is enclosed by single quotes (`'`). This is perfectly fine, since Python allows both types of quotes to enclose strings.

However, we'll have a problem if we try to mix these two types of quotation marks. Let's look at the following line of code that tries to use a single quote on the left side of a string, and a double quote on the right side of the string.

```
my_string_3 = 'We must away! We must away!\nWe ride before the break of day!"
```

If you save this code to a file called *ch1.py*, then ask Python to run it, you'll get the following traceback message:

```
  File "<stdin>", line 1
```

```
    my_string_3 = 'We must away! We must away!\nWe ride before the break of day!"
                  ^
SyntaxError: unterminated string literal (detected at line 1)
```

Remember that the last line of every traceback message is the most important line. This line tells us that we have an `unterminated string literal`. From Python's point of view, the string called `my_string_3` is "unterminated," meaning it never ends. Since the string starts with a single quotation mark, Python is expecting another single quotation mark to indicate the end of the string. There is no other single quotation mark, so Python thinks that the string has never ended, which makes the code impossible to run.

This type of error is called a *SyntaxError* because it's a violation of the rules of Python itself. Syntax errors are special because Python can catch them before it runs any of your code. Python will throw the error before a single line of code is run.

Type Errors

Another common error you might encounter is called a *TypeError*. Let's look at some code that will throw this type of error:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci["7"])
```

Listing 1-4: Attempting to specify a list index with a string

Look at Listing 1-3 and compare it to Listing 1-4. You'll see that the difference is very small: in Listing 1-4, we attempted to print `fibonacci["7"]]` (where `"7"` is a string), whereas in Listing 1-3, we attempted to print `fibonacci[7]]` (where `7` is an integer). Unfortunately, Python doesn't allow list indexes to be specified as strings, so it will throw an error. Here is the traceback message of the error it throws:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not str
```

You can see that this tells us exactly what the problem is: when we specify a list index, it must have the right data type, and string (`str`) data types are not allowed to specify list indices. In general, `TypeErrors` arise when ...

To resolve the bug, you will have to remove the quotation marks from the `"7"` so that it's interpreted as an integer rather than a string.

Name Errors

Consider the following code that appends even numbers to a list:

```
even_numbers = []
for n in range(10):
    if n % 2==0:
```

```
        even_numbers.append(n)
print(even_numbers)
```

You can see that the first line creates an empty list called `even_numbers`. The fourth line appends even numbers to this empty list. Finally, the last line prints our `even_numbers` list, which now contains several even numbers that we appended to it.

So far, so good. But look at what happens when we remove the first line and keep the rest of the code the same:

```
for n in range(10):
    if n % 2==0:
        even_numbers.append(n)
print(even_numbers)
```

Listing 1-5: Referencing a variable before it's defined

Here, we never bother to create an empty list called `even_numbers`. So, when the code gets to (what's now) the third line, it tries to append a number to some object called `even_numbers`. But there is no object called `even_numbers`. Our code is trying to append a number to an object that doesn't exist, and this throws an error.

If you save this code to a file called *ch1.py*, then attempt to run it with Python, you'll see the following traceback message:

```
# traceback message
File "<file_name>", line 5, in <module>
    even_numbers.append(n)
NameError: name 'even_numbers' is not defined
```

Python is calling this a *NameError*, because we referenced a variable whose name it doesn't recognize. You'll get a similar error if you ever try to reference a variable that hasn't yet been created. The resolution is simple: make sure to define every variable before referencing it.

ModuleNotFoundErrors and FileNotFoundErrors

Python scripts usually rely on outside resources to run successfully. For example, many Python scripts read external files or data. Most Python scripts also import packages to increase their capabilities.

There's nothing wrong with a script relying on outside resources. But there's a potential problem: sometimes your script relies on a resource it can't find. Consider the following example:

```
import species
```

This line tries to import a package called `species`, which is meant to help with analysis of exoplanet data. If you have not yet installed the `species` package on your computer, then you will see the following traceback message:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'species'
```

This is a *ModuleNotFound* error, meaning exactly what it says: Python tried to import the `species` package (also called the `species` module), but was unable to find it. You will have to install the `species` module if you want to be able to run this code.

A related type of error is the *FileNotFoundError*. You will see this type of error if you try to read a file that doesn't exist, or exists in a location that Python can't find:

```
import pandas as pd
df = pd.read_csv('exoplanets.csv')
print(df)
```

Here, we imported the `pandas` package, and tried to use it to import a file called *exoplanets.csv*. If that file doesn't exist, or if it exists in a place that Python doesn't know to search, then Python will output a *FileNotFoundError*. You can resolve this error by making sure the file exists, and then making it clear to Python exactly where it is. For example, the following change adds the explicit file path of the exoplanets file to our code:

```
import pandas as pd
df = pd.read_csv('/home/Users/OmarKhayyám/Desktop/exoplanets.csv')
print(df)
```

If you run this code on your computer, you should adjust it to match the file name of the file you're looking for.

Key Errors

So far, many of our Python scripts have used lists. But there are other useful data types in Python. Let's look at a way to use a *dictionary* in Python:

```
dict = {1: "Hello", 2: "World"}
print("{} {}".format(dict[1],dict[2]))
```

Here, we defined a dictionary called `dict`. This Python dictionary has two `keys` (the numbers 1 and 2), and two `values` (the words "Hello" and "World"). The keys and the values are associated with each other, so when we write `dict[1]`, Python knows we mean "Hello" (the dictionary value with key 1), and when we write `dict[2]`, Python knows we mean "World" (the dictionary value with key 2). That's why this code prints `Hello World` if you run it in Python. But if we change it to use an incorrect key, we could have problems:

```
dict = {1: "Hello", 2: "World"}
print("{} {}".format(dict[0],dict[1]))
```

Here is the traceback message:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

We have a *KeyError*, because we tried to access the key 0 when no such key exists in our dictionary (only 1 and 2 exist as keys). It's essentially the same problem we faced when we had an IndexError: we're trying to access a part of a data structure that doesn't exist. We can resolve it the same way we resolved the IndexError: by making sure that we only try to access keys and values that actually exist in our dictionary.

Other Types of Errors

The errors listed above are some of the most common errors that Python coders encounter. But there are more types of errors. You can see more details about other, less common errors in Appendix A. You can also refer to the official documentation at *https://docs.python.org/3/library/exceptions.html*.

Warnings

So far, we've seen examples of code that cause Python to throw an error. But errors are not the only thing that can be thrown and caught. Sometimes, Python will output *warnings*: alerts about potential problems in your code that aren't serious enough to halt the code's execution. Warnings and errors both belong to a class of objects that Python calls *exceptions*.

Let's look at an example of code that throws a warning:

```
fibonacci_seven = 21.0

if fibonacci_seven==21.0:
    print('The fibonacci_seven object is equal 21.0')

if fibonacci_seven is 21.0:
    print('The fibonacci_seven object is the same object as 21.0')
```

Listing 1-6: Using "is", a confusing operator

You can see here that we have two `if` statements: the first `if` statement checks whether `fibonacci_seven` is equal to the number 21.0; the second `if` statement checks whether `fibonacci_seven` is the same object as the number 21.0.

You may think that asserting that something *equals* 21.0 (as our first `if` statement does) is the same as saying that something is 21.0 (as our second `if` statement does). But in fact, Python treats these statements differently. When it checks for equality (when we specify `==`), it checks whether two numbers have the same size. But when it checks for identify (when we specify `is`), it checks whether they refer to the same object. From Python's point of view, "the same object" means that two objects literally reside at the same location in the computer's memory.

It's very possible for two objects to have the same size, but not be the same object in the computer's memory. This is exactly what happens in Listing 1-6: the `fibonacci_seven` object is the same size as `21.0`, but it's stored in the computer's memory in a different place, so when we ask if one of them `is` the other one, Python believes this is `False`. It's technically not against the rules of Python to use `is` rather than `==` for comparing numbers, but the maintainers of Python know that it's rare for someone to actually want to use `is` in this situation, so Python outputs the following warning:

```
/ch1.py:5: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if fibonacci[7] is 21.0:
The Fibonacci number with index 7 is equal to 21
```

You can learn more about warnings in Appendix B, and also in the official Python documentation at https://docs.python.org/3/library/warnings.html.

When your code throws an exception (whether it's an error or a warning), it's very possible that your code is buggy. But many bugs don't lead to exceptions being thrown or caught, so debugging is much more than just resolving exceptions. In the next section, we'll go over some exceptions that Python won't catch.

Errors that Python Won't Catch

So far, we've encountered situations in which we write code, try to run our code, and the code throws an error or warning instead of successfully running. We saw many types of errors, including syntax errors, value errors, index errors, and more.

There are some situations in which something is wrong with our Python code, but Python doesn't output a traceback message or warning. It's important to understand which problems Python won't catch; they can be be more subtle, and you will need to catch them yourself.

Infinite Loops

Let's start with a quick reminder of how `while` loops work:

```
n=1
while n<10:
    print(n)
    n = n+1
```

Remember that `while` statements are supposed to evaluate conditions. Here, `while n<10:` specifies that the indented code block will be executed as long as the variable `n` is less than 10. That condition will be true at first, since we set `n=1`. But it might become false later, since the indented code block contains a command to increase the value of `n`. As soon as `n` is greater than or equal to 10, the `while` loop will stop executing. This is how while loops are supposed to work: instead of running forever, they're supposed to stop executing after some finite number of iterations.

But some loops never reach a stopping condition. The following code snippet shows an example of this. Be careful! You may not want to run the following code snippet at home. If you run it, Python will attempt to continue running it literally forever. You will only be able to stop it by typing **CTRL-C** or **CTRL-Z** in the window where it's running, or by turning off your computer. Proceed with caution:

```
n=1
while n>0:
    print(" disorder-which, repeated, becomes order:")
```

This code contains an ordinary `print()` statement, which does nothing more than output text to the Python console. However, the `print()` statement is preceded by `while n>0:`. Unlike `while n<10:`, this condition will always be true, since we started with `n=1` and we didn't include any code to change the value of `n`. So this `while` loop will never stop executing, unless something interrupts it (like if the machine where it's running is manually shut down, or if you use the **`CTRL + C`** command we mentioned above to interrupt Python.

When a loop continues running without any chance of reaching an exit condition, we call this an *infinite loop*. It's a dangerous type of error because it's not one that Python catches for you. Because it's one that you'll notice only when the code is running—during its *runtime*—we call it a *runtime error*.

Semantic Errors

Let's look at some seemingly straightforward code:

```
def get_the_seventh_fibonacci_number():
    fibonacci = [1,1,2,3,5,8,13,21]
    return(fibonacci[2])
```

Listing 1-7: Exception-free code achieving the wrong goal (a semantic error)

Here, we defined a function called `get_the_seventh_fibonacci_number()`. You can call this function by running `print(get_the_seventh_fibonacci_number())` in Python. You'll find that it runs smoothly: it outputs `2` quickly and without outputting any errors or traceback messages. The problem is that the function's output, `2`, is the third Fibonacci number, whereas the function's name indicates that it should be outputting the seventh Fibonacci number instead.

Here, we have syntax that's correct, and code that runs quickly without exceptions or problems. The bug here is at a higher level than the syntax or even the code. The bug here is the contradiction between the intended purpose of the function and what it actually does. Bugs like this, where correct syntax implements the wrong functionality, are called *semantic errors*.

Semantic errors could have many causes. One common cause of semantic errors is misunderstanding. Sometimes a project leader has one idea of what a Python program should do, but fails to communicate it properly to the developers responsible for writing the actual code. The developers may

write beautiful code, but if they have misunderstood the leader's vision, it will be good code doing the wrong thing—a semantic error.

So far, we've seen exceptions that could be resolved easily, just by changing one number or adding or removing quotation marks. Dealing with semantic errors, by contrast, can be the most challenging part of debugging, partially because it can be hard to know whether the error even exists at all. Much of the rest of this book will address how to debug semantic errors.

Finding Bugs

Before you fix a bug, you need to know where the bug is. Many bugs are very easy to find. For example, consider the following code:

```
fibonacci = [1,1,2,3,5,8,13,21]

print("Let's get a Fibonacci number.")
print(fibonacci[5])

print("Let's find the next Fibonacci number after the end of our list.")
print(fibonacci[-1]+fibonacci[-2])

print("Let's add 1 to our largest Fibonacci number.")
print(fibonacci[-1]+'1')

print("Let's find the ratio of two of our Fibonacci numbers")
print(fibonacci[-1]/fibonacci[-2])
```

This code is meant to do some simple calculations related to our `fibonacci` list. But when we run it, it throws an error. Take a look at the output:

```
Let's get a Fibonacci number.
8
Let's find the next Fibonacci number after the end of our list.
34
Let's add 1 to our largest Fibonacci number.
Traceback (most recent call last):
  File "ch1.py", line 10, in <module>
    print(fibonacci[-1]+'1')
          ~~~~~~~~~~~~~~^~~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The first few lines show calculation outputs, just like we wanted. But the code didn't finish running: instead, we have a `TypeError` where we expected to see calculation outputs. There are a few ways to know exactly which part of the code caused this error:

File and line number in traceback Notice that the traceback message mentions `File "ch1.py", line 10`, meaning that our file called `ch1.py` contains the problematic code on line 10.

[Code snippet in traceback] In case the file and line number don't make the bug's location obvious enough, the traceback message contains a copy of the offending code: `print(fibonacci[-1]+'1')` along with punctuation below (`~~~~~~~~~~~~~^~~~`) in which the `^` character is supposed to pinpoint exactly the character on the line above that caused the error.

Printed outputs If you find traceback messages confusing, you can look at the outputs that were printed out before the beginning of the traceback message. You can see that the last output before the traceback message is `Let's add 1 to our largest Fibonacci number.` Thus we know that the bug is after the line of code that printed that message. Since every line of code after the first line is a `print()` statement, we know that the buggy line is the line immediately following the last printed line: the one that reads `print(fibonacci[-1]+'1')`.

Any of these three methods is sufficient to find the location of every bug we've discussed in the chapter so far. But not all professional code is as simple as the code we've introduced here. Let's go over a couple of bugs that are more difficult to locate, and how we can use these methods to find them.

Finding Bugs in Loops

Consider the following code that performs some calculations related to Fibonacci numbers:

```
import math
calcs = []
phi = (1+math.sqrt(5))/2
phi_conj = (1-math.sqrt(5))/2
n = 1
while n<100:
    phi_fib_n = int((phi**n - phi_conj**n)/(phi - phi_conj))
    calcs.append(1/(phi_fib_n-int(math.pi*10+3)))
    n+=1
```

This code is meant to perform some calculations related to Fibonacci numbers. You don't need to worry about the specifics of the calculations; we're only introducing it because it throws an error that we will learn how to locate. If you save the code to a file and try to run it in Python, you'll see this traceback message:

```
Traceback (most recent call last):
  File "ch1.py", line 15, in <module>
    calcs.append(1/(phi_fib_n-int(math.pi*10+3)))
                 ~^^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Remember that the last line of the traceback is the most important part. In this case, we have a *ZeroDivisionError*. Its name indicates that that something in the code has tried to divide by zero, which is against the rules of Python (as well as the rules of the universe as we usually understand them).

Resolving this error is simple in theory: we have to make sure that our code doesn't divide by zero. Let's find the bug that's attempting the illegal division by zero and resolve it.

The traceback message tells us which line caused the error: it was line 15, the one that tries to append to the `calcs` object. But the line 15 is part of a loop, so it's supposed to be executed nearly 100 times. The traceback message alone doesn't make it clear whether the error was thrown on the first iteration of the loop, or the 99th, or sometime in between, or whether it would be thrown on every iteration of the loop.

This is a case where the traceback message doesn't provide enough information to be completely sure about the cause of the error. But we can still rely on printed outputs to understand the location (the third method in our list above). Adding `print` statements to the code gives us a simple but effective way to determine exactly which iteration of the loop threw the error:

```
n = 1
calcs = []
phi = (1+math.sqrt(5))/2
phi_conj = (1-math.sqrt(5))/2
while n<100:
    print('Working on calculation number: ' + str(n))
    phi_fib_n = int((phi**n - phi_conj**n)/(phi - phi_conj))
    calcs.append(1/(phi_fib_n-int(math.pi*10+3)))
    n+=1
```

After adding a `print` statement, the updated code gives us following output:

```
Working on calculation number: 1
Working on calculation number: 2
Working on calculation number: 3
Working on calculation number: 4
Working on calculation number: 5
Working on calculation number: 6
Working on calculation number: 7
Working on calculation number: 8
Working on calculation number: 9
Traceback (most recent call last):
  File "/home/bradfordtuckfield/ch1temp.py", line 16, in <module>
    calcs.append(1/(phi_fib_n-int(math.pi*10+3)))
                 ~^^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

We can see that the traceback message appears just after the code has told us that it's working on its ninth calculation. Before completing the loop

and starting the tenth calculation, the code halted and the traceback message was output. This indicates that the ninth iteration of the loop was the one that threw our error and attempted division by zero.

We can prevent this error by ensuring that we don't ever divide by zero. Since we've determined that division by zero occurs in the ninth iteration of the loop, one simple solution is to make sure to exit the loop before iteration number nine:

```
n = 1
calcs = []
while n<9:
    print('Working on calculation number: ' + str(n))
    phi_fib_n = int((phi**n - phi_conj**n)/(phi - phi_conj))
    calcs.append(1/(phi_fib_n-int(math.pi*10+3)))
    n+=1
```

Notice that we found and resolved the error without ever closely examining the calculations in the code. In some cases, adding `print` statements or other easily readable outputs can enable you to resolve errors even when you don't fully understand what caused them. This can be especially useful if you ever have to debug complex or low-quality code written by a colleague. You may not need to deeply understand every strange or incorrect calculation they attempted. Instead, it may be enough to use the same methods we used here to simply find the bug and remove or avoid it.

Understanding Flow of Execution

Let's look at another example of a bug that's hard to find:

```
def function_eleven(list_c):
    list_d = [x*x for x in list_c]
    return(list_d)

def function_first(list1):
    list2 = function_a([x+1 for x in list1])
    return(list2)

def function_a(list7):
    list6 = function_eleven([x^2 for x in list7])
    return(list6)

def function_12(list_that):
    list_this = function_first([x*2 for x in list_that])
    return(list_this)

fibonacci_short = [1,1]
final_list = function_12(fibonacci_short)
```

```
print(final_list)
```

Listing 1-8: Code with a complex flow of execution

This code is confusing for a few reasons. First of all, the functions seem like they were named based on inconsistent naming schemes. We have four functions: one is called `function_first()`, but it's not written or referenced first. We also have `function_a`, which could be first in an alphabetic list, but the other functions aren't named alphabetically. The other functions are numbered eleven and twelve, without any indication of what might have happened to functions ten, nine, eight, and so on.

The function names aren't the only thing that's confusing. If you look at the last few lines of Listing 1-8, you'll see that it calls `function_12()`, passing our familiar `fibonacci` list as its argument, and then prints the result. But `function_12()` references `function_first()`, which references `function_a()` in turn, which finally references `function_eleven()`. We have multiple functions with inconsistent names and unclear purposes referencing each other. Even though this code is less than 20 lines long, it's hard to know at first glance what it does, much less where its bug is.

When you run this code, the list called `fibonacci_short` will be passed as an argument `function_12`, and then subsequently transformed and passed to other arguments before finally being returned and printed. We can describe this by saying that the `fibonacci_short` list "flows" through the code from one operation to the next. To understand Listing 1-8 completely, we need to understand the order in which all of its operations are performed, also called its *flow of execution*. When code is hard to read and has a complex flow of execution, we often colloquially refer to it as *spaghetti code*.

If you run the code in Listing 1-8, you should find that it prints the following output:

```
[1, 1]
```

If you look at the code in Listing 1-8, you might be surprised about this output. The argument we used in the code is our `fibonacci_short` list, which consists of the first two numbers in the Fibonacci sequence (1 and 1). Then, in `function_12()`, we multiply every element of the list by 2. Every other function in Listing 1-8 looks like it should be increasing the size of list elements: adding 1, multiplying by 2, or squaring them. If every function looks like it should lead to larger outputs, then how is it that our final output is the same size as our initial input?

You might find this question hard to answer. There's no traceback message in the output, indicating that Python didn't catch any exceptions whatsoever - so the syntax is sound. Nevertheless, you suspect that there could be a semantic error in this code, since you didn't get the larger outputs you expected. But with spaghetti code like this, it can be hard to identify exactly what the semantic error is or where to find it in the code.

One simple way to understand the code better is to do the same thing we did with a `while` loop in the previous section: add one or more `print` statements that indicate exactly what is happening at every point in the flow:

```
def function_first(list1):
    print("Currently executing the beginning of function_first")
    print("The function input is: " + str(list1))
    print("")
    list2 = function_a([x+1 for x in list1])
    return(list2)

def function_12(list_that):
    print("Currently executing the beginning of function_12")
    print("The function input is: " + str(list_that))
    print("")
    list_this = function_first([x*2 for x in list_that])
    return(list_this)

def function_eleven(list_c):
    print("Currently executing the beginning of function_eleven")
    print("The function input is: " + str(list_c))
    print("")
    list_d = [x*x for x in list_c]
    return(list_d)

def function_a(list7):
    print("Currently executing the beginning of function_a")
    print("The function input is: " + str(list7))
    print("")
    list6 = function_eleven([x^2 for x in list7])
    return(list6)

fibonacci_short = [1,1]
final_list = function_12(fibonacci_short)

print("The final output is:")
print(final_list)
```

This is the same code as we saw in Listing 1-8, but with `print` statements added at the beginning of every function to tell us the results of all of the calculations that have been performed at each point in the flow. These `print` statements make the problem much easier to locate:

```
Currently executing the beginning of function_12
The function input is: [1, 1]

Currently executing the beginning of function_first
The function input is: [2, 2]

Currently executing the beginning of function_a
```

```
The function input is: [3, 3]

Currently executing the beginning of function_eleven
The function input is: [1, 1]

The final output is:
[1, 1]
```

You can see that these `print` statements make it much easier to follow the flow of execution. We see that the input to `function_12()` is `[1,1]`. We know that `function_12()` multiplies inputs by 2, so it's not surprising to see that the input to `function_first`, the next function reached in the flow of execution, is `[2,2]`. The input to `function_a`, in turn, is `[3,3]`. Just as we expected, the values are consistently increasing. But next, we see that the input to `function_eleven` is `[1,1]`. We see that between the beginning of `function_a` and the beginning of `function_eleven`, something has gone wrong: `function_a` has decreased, rather than increased, the input values it received.

Let's look more closely at `function_a` to understand what happened. This function performs the calculation `x^2` with every number that's passed to it as input. In many programming languages, `x^2` means "x squared," the product of x with itself. But remember, in Python, the `^` operator doesn't perform squaring; it performs a rarer operation called *bitwise or*, which can create outputs that are smaller than its inputs. If the code in Listing 1-8 had been written by someone who was more accustomed to other programming languages, it's very possible that they wrote `x^2` when they intended to perform squaring (which is denoted by `x**2` in Python).

What we have here is a potential semantic error, one whose location was only discovered by adding `print` statements to our spaghetti code. Being able to debug spaghetti code will be a very valuable skill when you work with teams and need to debug low-quality, poorly documented code written by someone else.

Reading Traceback Messages

We've looked at several traceback messages in this chapter (including Listing 1-2). Let's take a moment to look closely at every element of a traceback message.

Remember the following buggy code, which was previously introduced as Listing 1-1:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci[10])
```

Listing 1-9: Buggy code, again

If you save this code in a file called *ch1.py*, and then run that file in Python, you'll get the following traceback message:

```
Traceback (most recent call last):
```

```
  File "ch1.py", line 2, in <module>
    print(fibonacci[10])
          ~~~~~~~~~~^^^^
IndexError: list index out of range
```

Listing 1-10: Traceback output, again

Let's go over every part of this traceback message. Remember that we usually start by reading the last line of a traceback message, then work upwards:

`IndexError: list index out of range` The last line of traceback message tells us the particular error of the buggy code in Listing 1-9.

`print(fibonacci[10])` The second-to-last part of the traceback shows the excerpt from the code that caused the error to be thrown, and it has helpful characters drawn beneath it (`~~~~~~~~~~^^^^`) that attempt to pinpoint the exact characters that caused the problem.

`File "ch1.py", line 2, in <module>` This line tells us the file whose code was running when the error was thrown, and the line in that file that threw the error.

`Traceback (most recent call last):` This line announces that we're reading a traceback message. The (`most recent call last`) parenthetical tells us that the code is displayed in reverse chronological order: the most recently executed line of code that caused the error is shown last, the second-most-recently executed line of code that caused the error is shown second-to-last, and so on.

We've encountered each of these four elements of traceback messages previously in this chapter. The only part that we haven't discussed in detail is the parenthetical on the first line that says (`most recent call last`). Let's look at some code that will throw an error that makes the meaning of this parenthetical more clear:

```
fibonacci = [1,1,2,3,5,8,13,21]

def function_a(some_list):
    toreturn = some_list[10]
    return(toreturn)

def function_b(some_list):
    toreturn = function_a(some_list)
    return(toreturn)

def function_c(some_list):
    toreturn = function_b(some_list)
    return(toreturn)

print(function_c(fibonacci))
```

Here we have three functions: `function_c()` calls `function_b()`, which calls `function_a()`. For its part, `function_a()` throws the same IndexError we saw at the beginning of the chapter, caused by trying to access a list element that doesn't exist. Let's look at the traceback message that Python outputs when we attempt to run this code:

```
Traceback (most recent call last):
  File "/home/bradfordtuckfield/ch1.py", line 15, in <module>
    print(function_c(fibonacci))
          ^^^^^^^^^^^^^^^^^^^^
  File "/home/bradfordtuckfield/ch1.py", line 12, in function_c
    toreturn = function_b(some_list)
               ^^^^^^^^^^^^^^^^^^^^
  File "/home/bradfordtuckfield/ch1.py", line 8, in function_b
    toreturn = function_a(some_list)
               ^^^^^^^^^^^^^^^^^^^^
  File "/home/bradfordtuckfield/ch1.py", line 4, in function_a
    toreturn = some_list[10]
               ~~~~~~~~~~^^^
IndexError: list index out of range
```

Listing 1-11: Traceback output

This traceback message is long, but you can see that it has all of the elements that were described in the list above. The reason it's longer is that it outputs four separate lines of code instead of just one when describing the location of the error.

You can see that just before the end of the traceback, it shows that

```
toreturn = some_list[10]
```

caused the IndexError, on line 4 of the script. But line 4 is only accessed because of the function call on line 8 (`toreturn = function_a(some_list)`). So, if you look further up in the traceback message, you can see that line 8 is also described in the part describing the location of the error. The function call on line 8, in turn, only happens because the function it lives in is called on line 12, which in turn is only accessed because of a function call on line 15. All of these lines are repeated in the traceback message. By reading the traceback from the end to the beginning, we're able to "trace back" the flow of how the error-throwing code was executed—hence the name of a traceback message.

In some cases, one single line of code can call another function, which can call another function in turn, resulting in a chain of a dozen or more function calls that all have to be listed in a single traceback message. The following code throws an error whose traceback message shows an example of this:

```
import numpy as np
from sklearn.datasets import make_circles

X, y = make_circles()
```

```
X_train = X[:70] # split train and test data into 70% and 30%
y_train = y[:70]
X_test = X[70:]
y_test = y[:70]

from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
print("Accuracy of KNN test set: {:.2f}".format(knn.score(X_test, y_test)))
```

You don't need to worry about the details of this code. It's intended to accomplish a simple machine learning task, but a small bug causes an extremely long traceback message to be output:

```
Traceback (most recent call last):
  File "...runpy.py", line 193, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "...runpy.py", line 86, in _run_code
    exec(code, run_globals)
  File "...pythonFiles\lib\python\debugpy\__main__.py", line 45, in <module>
    cli.main()
  File "...pythonFiles\lib\python\debugpy/..\debugpy\server\cli.py", line 444, in main
    run()
  File "...\pythonFiles\lib\python\debugpy/..\debugpy\server\cli.py", line 285, in run_file
    runpy.run_path(target_as_str, run_name=compat.force_str("__main__"))
  File "...runpy.py", line 263, in run_path
    return _run_module_code(code, init_globals, run_name,
  File "...runpy.py", line 96, in _run_module_code
    _run_code(code, mod_globals, init_globals,
  File "...runpy.py", line 86, in _run_code
    exec(code, run_globals)
  File "file_name", line 13, in <module>
    print("Accuracy of KNN test set: {:.2f}".format(knn.score(X_test, y_test)))
  File "...sklearn\base.py", line 499, in score
    return accuracy_score(y, self.predict(X), sample_weight=sample_weight)
  File "...sklearn\utils\validation.py", line 73, in inner_f
    return f(**kwargs)
  File "...sklearn\metrics\_classification.py", line 187, in accuracy_score
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
  File "...sklearn\metrics\_classification.py", line 81, in _check_targets
    check_consistent_length(y_true, y_pred)
  File "...sklearn\utils\validation.py", line 256, in check_consistent_length
    raise ValueError("Found input variables with inconsistent numbers of"
ValueError: Found input variables with inconsistent numbers of samples: [70, 30]
```

Listing 1-12: Very long traceback output

If you read the last line of this traceback message, you can see that the code has thrown a `ValueError`. If you look through the traceback message,

you can see that there's a long chain of functions that all call each other. Many of the functions that are described in the traceback are not part of our code in Listing 1-12. Instead, they're functions that were imported, and then called by our code. Usually, it's not necessary to look closely at the code of the packages you import: you should focus on the code that you yourself have written, and resolve only the bugs there.

Exception Handling

Resolving bugs isn't the only only task you need to do when debugging. Remember that an ounce of prevention is worth a pound of cure. In other words, it's better to prevent errors from being thrown in the first place than it is to merely resolve them. For example, imagine that you're working on a Python project with one or more colleagues. You start with some code that we introduced earlier in the chapter:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci[7])
```

Listing 1-13: Good code - I hope your colleague doesn't ruin it

This code outputs 21, the largest Fibonacci number in our `fibonacci` list, without throwing any errors. However, suppose that one of your colleagues edits the code and adds a bug:

```
fibonacci = [1,1,2,3,5,8,13,21]
fibonacci.pop()
print(fibonacci[7])
```

Listing 1-14: Changes and additions to code can bring back bugs you thought were dead

The second line, `fibonacci.pop()`, will remove the last element of the `fibonacci` list, so that there is no element of the list with index 7. When you run the code in Listing 1-14, you'll get an error that's almost identical to the error we got before:

```
Traceback (most recent call last):
  File "ch1.py", line 3, in <module>
    print(fibonacci[7])
          ~~~~~~~~~^^^
IndexError: list index out of range
```

This is all too common for programmers: you resolve a bug, and then some change to your code that may seem unrelated to the bug brings the bug back, or creates some completely new bug. This is especially likely when you work on a team: two colleagues aren't familiar enough with each other's code, and unknowingly create bugs for each other.

To make sure a bug that you've resolved can never come back, you could try to add some logic to your code so that no matter what change is made, you'll never get this same error again. You can do this by checking whether the index you're trying to access is in the range of accessible indexes:

```
fibonacci = [1,1,2,3,5,8,13,21]
index_of_interest=7
if abs(index_of_interest) in range(len(fibonacci)):
    print(fibonacci[index_of_interest])
```

Listing 1-15: Attempting bug-proofing

Here, we do an explicit check to determine whether a particular index number is in the range of indexes that we can access. Now, changes to the `fibonacci` list, or other changes to the code, shouldn't lead to any IndexErrors. But never underestimate how much damage your colleagues can do to your code. Suppose that someone on your team leaves the `if` statement intact, but adjusts some of the other code in a way that introduces a bug:

```
fibonacci = [1,1,2,3,5,8,13,21]
fibonacci2 = [1,1,2,3,5,8]
index_of_interest=7
if abs(index_of_interest) in range(len(fibonacci)):
    print(fibonacci2[index_of_interest])
```

When we run this code, we'll get an IndexError again, because our if statement is not sufficiently safe as a bug-proofing method.

When code is very complex, it can be difficult to be sure about whether a particular `if` statement is sufficient to protect against bugs. In the next section, we'll go over a more robust way to prevent errors from being thrown.

General Exception Handling with Try/Except

One last time, let's return to the very first example of buggy code that we looked at at the beginning of the chapter:

```
fibonacci = [1,1,2,3,5,8,13,21]
print(fibonacci[10])
```

You'll recall that we got an IndexError when we ran this code.

Now, let's use some built-in Python capabilities to prevent errors from being thrown:

```
fibonacci = [1,1,2,3,5,8,13,21]
try:
    print(fibonacci[10])
except IndexError:
    print('You tried to access an index that is out of range.')
```

Listing 1-16: Exception handling with try/except statements

Here, we've added what's called a *try statement*. This statement instructs Python to try everything in the indented block after `try:`. If the code in that block throws an error, Python will not output a traceback message or halt the code. Instead, it will proceed to the `except` statement just after the `try`

statement. If the code in the `try` block doesn't throw an error, then the `except` block is skipped, and the code's execution proceeds as normal.

When we run this code, we should get simple output:

```
You tried to access an index that is out of range.
```

In this case, Python tried to print the element of the `fibonacci` list with index 10. Since there was no such element, we would have gotten an `IndexError`, but our `try` and `except` statements ensured that Python didn't halt our code's execution or throw an error, printing a custom message instead.

Using `try` and `except` statements doesn't remove or resolve errors. But it ensures that if errors are thrown, they won't halt our code, and we'll be able to specify helpful responses in our `except` code blocks. In other words, `try` and `except` statements *handle* exceptions, and that's why we refer to them as *exception handling* methods. If you have mission-critical Python code that shouldn't ever crash, adding exception handling can be very valuable.

Exception Handling with Multiple Types of Errors

In Listing 1-16, you can see that we had a very specific `except` statement. We wrote `except IndexError:`, meaning that the code in the `except` block will only be run in case an IndexError is thrown. But `except` statements are flexible, and we can specify any type of error we'd like to handle. For example, if we want to protect against type errors, we can do the following:

```
fibonacci = [1,1,2,3,5,8,13,21]
try:
    fibonacci[1] + 'fibonacci[1]'
except TypeError:
    print('You got a type error.')
```

When we do this, we get simple output:

```
You got a type error.
```

We can specify multiple `except` statements for every `try` statement, and we can even specify groups of errors that should be handled identically:

```
fibonacci = [1,1,2,3,5,8,13,21]
try:
    fibonacci[1] + 'fibonacci[1]'
except (RuntimeError, TypeError, NameError):
    print('You got a RuntimeError or a TypeError or a NameError.')
except IndexError:
    print('you got an index error')
else:
    print('you got some other kind of error')
```

You can see that we have several `except` statements here. Python will attempt to run the code in the `try` block. If it encounters either a `RunTime Error` or a `TypeError` or a `NameError`, then it will execute the code in the first `except` block. If it encounters an `IndexError`, then it will execute the code in the second `except` block. If it encounters any other kind of error, it will execute the

code in the final `except` block. This allows us to specify different custom responses to any type of error or exception thrown by the code.

Begin with the End in Mind

Remember our definition of a bug: anything that causes your code's actual output to be different from its intended output is a bug. If you have a good idea of what the intended output of your code is, then it's often a straightforward process to make your code perform the intended function.

The problem is that it can actually be quite hard to clearly state the intended purpose of a piece of code. Sometimes, clients or project leaders will use language like this to describe what they want code to do:

- "Use AI to make more money for us"
- "Write a program that performs all the functions of a bank."
- "Collect data about our competitors"

These descriptions are reasonable as business goals, but are not detailed enough for serious programming work. Vague descriptions like these make semantic errors much more likely. For example, programmers who are told to work on code for a bank may use a formula for continuous compounding to calculate accrued interest, but the bank's owners may want to pay interest on a monthly, rather than continuous basis. Even if the code is flawless, if it implements the wrong formulas, it's buggy.

When debugging, it's important not only to have strong Python skills, but also to have the right mindset: a mindset concerned with clearly and thoroughly documenting the exact intended output for all code, and ensuring that code matches the output only after that documentation is complete.

The importance of beginning with the end in mind is something that the mathematician Charles Lutwidge Dodgson (pen name Lewis Carroll) understood when he wrote *Alice in Wonderland*, in which Alice had this conversation related to her direction of travel:

> Alice: Would you tell me, please, which way I ought to go from here?
> The Cheshire Cat: That depends a good deal on where you want to get to.
> Alice: I don't much care where.
> The Cheshire Cat: Then it doesn't much matter which way you go.

In the debugging process, something similar is true: we can resolve errors and adjust code all day, but if we don't have a clear, agreed-upon idea of exactly what the code is supposed to do, we'll never succeed. Like Alice, we might get somewhere, and we might get code that does something, but it's better to make slow progress towards the right thing than quick progress towards the wrong thing. Good debuggers spend a great deal of time thinking carefully about where they want to go, and exactly what their code is supposed to do before they write a single character of code. Good debuggers always begin with the end in mind.

Conclusion

In this chapter, we talked about the idea of a bug. We discussed examples of common errors, including errors that Python won't catch. We went over how to find bugs, how to read traceback messages, and how to handle exceptions. In the next chapter, we'll talk about a particular debugging tool called pytest, which is extremely common and also happens to be easy to use.