Chapter 13

DEBUGGING STRATEGIES

Writing a program is just one step in programming. Next comes getting your new program to work properly—in other words, to run without bugs. The most obvious bugs are errors which prevent the program from running at all. However, bugs also include problems which don't prevent a program from running, but lead to incorrect results. These sorts of bugs can be more serious because they are harder to identify in the first place. In this chapter, we describe some general strategies for avoiding, identifying, and fixing bugs. We also provide a table listing common Python error messages you may encounter, along with potential solutions.

Learning by debugging

Debugging is the process of locating and removing errors in your programs. Bugs come in two main types. The most obvious are those which cause your program to crash or not to run to completion. While frustrating, these errors can usually be located and fixed using the strategies we will outline shortly. The second type of bug is more insidious. These occur when your program runs to completion and appears to work, but is actually generating incorrect or incomplete results. This type of bug is difficult to detect and is more dangerous to your science than getting no result at all.

Addressing bugs that create incorrect output is a twofold process: first, designing your programs to avoid such bugs to begin with, and second, validating your results. The validation process can include hand-checking results for a subset of your data, comparing the results to those of another program, and building redundancies into your program to flag potential problems. You can also anticipate errors by writing your program to give you progress reports as it goes. For example, did the program process all the expected files? Did it analyze the expected number of data rows? You can even write a separate shell script to test your program under a variety of conditions and then check the results.[1]

Although debugging can be frustrating, it is also one of the best ways to improve your programming skills. This is especially true if you make notes on what you did to solve a problem—you will find yourself dealing with fewer and fewer errors as you learn to anticipate mistakes. In addition, chances are that most of the problems you will encounter have been found by others first, so searching for solutions online can lead you to new ways of approaching a particular programming challenge. This will expand your programming repertoire.

Debugging often forces you to dig a bit deeper to understand how tools you regularly use work under the hood. Stepping through the source code of modules or reading about esoteric orders of operation can be frustrating when your main goal is to fix a program so you can complete an analysis, but over time, debugging will build your depth of knowledge and make you a better programmer. It is very satisfying to write a large block of code and have it run perfectly the first time, and to use skills you learned when debugging one program to write another from scratch.

In this chapter we'll first discuss some general ways to identify and avoid bugs, and then present some specific error messages that you might encounter, along with probable causes and suggested solutions.

General strategies

Build upon working elements

When you begin to put together a program, begin by thinking through your general approach to the problem, then start building towards incremental successes. Get each element of your program working before moving too far ahead. If you are planning to read in a series of files, for example, you could get the program reading from a single file first, and then tuck that into a loop that does the same operation for several files. Or you could start by gathering and printing out the list of filenames you hope to read from. This strategy will let you identify errors before they are buried deep within many lines of code. Since you can't check the final result until all elements of the program have been written, this will often require writing test code that prints out the program status at various points. Once you are satisfied by the output of this test code, you can comment it out—functionally deleting it from your program, but leaving it available for future debugging.

If you are writing to a file, always set up a sandbox folder—that is, a place where you and your program can play safely—and work on a copy of your data rather than on the original. This will avoid erasing or modifying critical data while you develop, test, and debug. Remember, even an act so simple as opening the wrong file for writing can erase the data that file contains. Another safety measure

[1]Some professional programming tool kits actually come with an "infinite monkeys" feature, which randomly pushes buttons and enters text into your program to check whether some combination of user actions will make it crash.

is to print the output filename and file contents to the screen in preliminary tests, before writing the code that actually opens the output file and writes data to it. This will help head off problems before you clobber something by accident.

Think about your assumptions

Several types of frustrating errors are due to mistaken assumptions of the most basic sort. Here are some things to be cautious of:

1. **Make sure you are editing the version of the program that you are actually using.** If at any point you have copied the program file to a data directory, saved a different version, or made a copy to send to a colleague,[2] you may end up editing a different version of the program than the one that actually executes when you type its name at the command line. One of the most frustrating debugging experiences is to identify and correct a problem, yet continue to get the same error when you run the program—simply because the copy you edited is not the copy you're executing. You can use the `which` command to get the absolute path of the program being called from the command line (for example, `which myprogram.py`). Make sure the name reported is the same file you are editing. As we have explained elsewhere, it is best to not have more than one copy of the same program on your computer. You can use a program from anywhere as long as the folder containing it is part of your `PATH`.
2. **Make sure you save your changes to the program file before re-executing.** The shell will execute the version of the program saved on your disk, not what you happen to have in your text editor. If you fix your program but don't save the changes, it will still fail.
3. **Check line endings and compensate for them if needed.** If your program reads text files, make sure it is really reading one line at a time. If you have incorrect end-of-line characters it may assume that all the contents of the file are in a single line. One way around this problem is to open text files with universal newline support by adding a `U` flag to the open command:

   ```
   InFile = open(InFileName, 'rU')
   ```

   When files are opened this way, all line endings are converted to newline (`\n`) characters.

[2] Incidentally, sending programs by email is more difficult than it sounds. Many virus checkers will strip off attachments that contain a shebang line, and some will even remove script text pasted into the body of a message. If you try to compress the program into a ZIP file, your attachment will probably still not go through, due to virus scanning activity. You might need to copy the script without the shebang line, or else post it on an FTP site or in a shared network folder on your computer.

4. **Check the contents of your data file.** A bug-free program can crash or generate incorrect results if there is a problem with the input data files. These problems can come in many forms. Are there really only AGCTs in your sequence files, or might there be other symbols, like `-`, `?`, or an asterisk at the end? Are the numbers that you read in always positive, or are there negative symbols in front of some? In many cases you will find yourself working with files whose formats have not been explicitly defined; in such cases, the input file may be correct in the eyes of another program, but not what you anticipated when you wrote your own program. For instance, you may be expecting to see `-1` for a missing value, but instead discover data that use `NaN` for this purpose instead. This can cause an error if you try to convert all values to integers.

You should anticipate formatting issues by checking that the data are as you expect before you parse or use them. If you get errors that you suspect are due to problems with the input file, add code to print the line number of the data files as the program processes the input. Then take a look at the data lines in the neighborhood of the last line that was processed before the crash. This is another example of the value of writing test code that provides feedback as the program runs.

Specific debugging techniques

Isolate the problem

When your program fails and prints out an error message, the solution can be obvious or it can be obscure. Often the problem is not on the line that is reported as generating the error. If you don't see a problem on the line that fails, start looking for errors in the preceding lines. If there is a missing parenthesis or unclosed quotation mark, for example, the program may continue beyond the line containing the error before encountering something that does not compute.

Commenting out sections of your program can help isolate the problem. Comments in effect turn these sections off, so that you can examine how the program runs without them. Once the problem has been found and fixed, it is a simple matter to remove the comment marks. You can turn a line of code into a comment in many languages just by adding a # to the beginning of the line. This works well for one or several lines of code, but gets cumbersome if you want to turn off a large block of the program; in the latter case, multiline comments are easier to use. In Python, these are demarcated by triple quotes: three quote marks in a row both before and after the code to be skipped.[3]

Comments can also be used to label the end of loops. With nested statements (several levels of indentation), it can be difficult to identify which if statement or `for` loop is designated by a particular indentation level. Putting a short comment,

[3]As you may recall from Chapter 8, triple quotes define a multiline string that includes line endings. Putting triple quotes around lines of code creates a string out of them. However, because this string isn't assigned to a variable, the end result is that the contents of the string are invisible to the rest of the program.

such as `#end if negative` or `#end for each file`, at the end of that indented block, using the proper indentation level, can help you avoid inadvertent indentation (or outdentation) of commands into the wrong block.

Write verbose software

As you work through your program, give yourself plenty of diagnostic `print` statements. Report the name of the file that has been opened, the value of the first line or the number of lines processed, and a summary view of lists that you parse or build from input files. Each time you insert such a statement, preface it with an `if Debug:` statement. This allows you to define the variable `Debug` at the top of your program as either `True` or `False`, so as to quickly turn on and off the status reports during alternating bouts of testing and actual use. You can even make this extra output an option that can be toggled on and off with an argument when the program is run at the command line.

```
import sys
Debug = True

# (insert program statements here)

# wherever you want to give feedback, insert these lines
if Debug:
    print MyList
    # or you can use
    sys.stderr.write(MyList)
```

You don't have to create a fancy `print` statement to output your results during debugging. Usually something simple and quick will suffice. Python will print the entire contents of a list or variable if you just use the name, even if the variable is not a string.

If part of a program is generating errors or you suspect that it is functioning incorrectly, it can be very informative to print out the values of lists or dictionaries that are being used at that point in the code. For example, you might be trying to use an index for an element that isn't there. (Remember that indexing starts at `0` for the first element, and that the last element has the index of the length of the list minus 1). Or you might be trying to append to a list which hasn't been initialized (`MyList = []`). Many of these errors will become obvious if you add a `print MyList` statement before carrying out more operations.

You can also use the Python interpreter as a debugging device. Paste portions of your program into the terminal, and then explore the values of variables by typing their names. This approach works best for testing parsing procedures, regular expressions, and other string-based operations; it doesn't work as well for operations involving reading and writing files, but you can simulate the first line that is read from a file by just defining that value manually. For example, instead of `for MyLine in MyFile:` at the prompt, you can just say `MyLine=""` and then paste the value of an example line between the quotes.

TABLE 13.1 Common errors running Python programs and some potential solutions

Example of reported error	Probable causes and solutions
`-bash: myscript.py: command not found`	If this error begins with `-bash` or with the name of your shell, then the program you are trying to run (`myscript.py` in this example) is either not in a folder listed in your `PATH`, or else its permissions are not set to executable. See Chapter 6 for setting the `PATH` variable, and try `chmod u+x myscript.py` at the command line to designate the file as executable.
`/Users/lucy/scripts/myprogram.py: line 3: import: command not found`	If the `command not found` error is reported from within your Python program, rather than from `bash` (you can tell based on the text that precedes it), then there may be a problem with your shebang line. Specifically, it may not be sending the contents of your script to the Python program or other suitable interpreter. Double-check the `#!` line in the file, or copy one out of a working program. This type of error can also be generated if you misspell a built-in Python function within your program. Carefully check lines that call such functions for typos.
`bad interpreter not a directory`	The shebang line has a slash after `/usr/bin/env/`, as if `env` were a directory. Remove the slash so the interpreter understands that `env` is a program.
`usr/bin/env: bad interpreter: No such file or directory`	The part of the statement in your shebang line after env wasn't found. Copy the contents of the shebang line after the `#!` characters and paste into a terminal window to see if this launches Python.
`permission denied`	Permissions not executable. Fix with `chmod u+x myscript.py`
`name 'x' is not defined`	There are several possibilities here: • You misspelled a variable name in the program. • You are trying to modify a variable that was not originally defined. For example, you may be adding to a list or string that was not first defined as empty. Fix by initializing the variable, e.g., `MyList=[]` or `MyString=""`. • You are trying to use a function that needs to be imported from a module first. • You are using a function without the required module name in dot notation, for example `randint(5)` instead of `random.randint(5)`. Either add the module name to the function name, or else rephrase your import using `from` (as in `from random import *`) to access all the module's functions directly.

Error messages and their meanings

Common Python errors

When you run your Python script, in addition to the status messages that it prints out, you will also see error messages generated by the Python interpreter if your program crashes. In some cases, these will give a clear indication of what needs to be fixed, but in many situations they can be difficult to understand. Table 13.1 presents a list of the most common Python error messages, along with their possible causes, and suggestions on how to locate and fix the problem.

TABLE 13.1 (continued)

Example of reported error	Probable causes and solutions
`Indentation error:`	Take a guess! If the text has been pasted from a Web source, use Show Invisibles in your text editor to make sure the indentations are all just tabs or all just spaces. Also remember that invisible characters are sometimes be introduced when copying and pasting.
`Attribute error`	Misspelling of a built-in function. For example, you would get this error if you have a string `MyString`, and use `MyString.lowercase()` instead of `MyString.lower()`. To fix, double-check the function or variable name that comes after the dot in your dot notation.
`type error 'xx' object is not callable`	Trying to retrieve values from a list using parentheses instead of square brackets, causing the list be interpreted as a function name.
`traceback...zero division error`	Division by zero. This often happens when a function returns an unexpected zero, or when there is an unexpected zero in the input data. Check user and variable input to make sure that strings exist and are not blank. Test to make sure that a number is nonzero before using it as a denominator.
`Non-ASCII character '\xe2' in file`	One possibility is that the file contains one or more "curly quotes." Search for these and replace any found with "straight quotes." Another possibility is that some other symbol such as a bullet or degree sign has been used without specifying Unicode UTF-8. Add `# coding: utf-8` below the `#!` line.
`invalid syntax`	This could be many things, either in the line indicated or in preceding lines: • Missing colon after `if`, `else`, or `for` statement • Missing close parenthesis or close bracket • Using = instead of == in a logical test. • Mixing spaces and tabs when indenting

Shell errors

A common shell script error is `Illegal byte sequence`, which is sometimes reported by shell commands upon reading a file. This error originates from the presence of Unicode characters (atypical punctuation such as • ° ≠ or curly quotes) in a file being read. Some command-line programs, including `cut` and `tr`, can't process Unicode characters. If you get such an error, open the file in a text editor and remove the problematic characters using search and replace.

Other common errors stem from the use of symbols—for example, \ > * < ; / and also spaces—which have powerful meanings in the shell (and not just as emoticons). If used improperly, these symbols can result in the loss of files, so it is a good idea to test your shell commands and scripts thoroughly within a sandbox folder before using them on actual data. If function arguments include strings with punctuation, be sure there are quotes around the string and that symbols are escaped with \ where necessary. Otherwise, the shell will interpret the punctuation according to its own convention. For instance, `grep ">" test.fasta` finds all lines in the file `test.fasta` that contain >, while `grep > test.fasta` redirects the output of `grep` to `test.fasta`, overwriting all of that file's contents.

Making your program more efficient

Optimization

In some cases a program does what is expected, but is so slow or requires so much memory that it is essentially unusable. Given the speed of modern computers, the efficiency of a program may not be a significant issue for typical data processing tasks. However, as the size of your datasets grow and your analyses become more complex, improving efficiency may become essential—or at the least, a way to make your analysis experience more serene.

In most cases, there are several ways to write any given program; therefore, as you plan a project, it is good to think about which approach would be the most efficient. Most of a typical program's time will be spent doing the same operations over and over in a loop. Identify which loops will be reiterated the most, and focus on optimizing the code inside them before you move on to other, lower priorities. Make sure that variables aren't being created if they aren't being used. If a function is being called with the same arguments many times, see if you can call it once and then store the result in a variable that can be reused.

You can get a sense of the relative efficiency of different approaches by printing or saving the time when you start an operation (one function in the `time` module is `time.localtime()`) and then printing the time again, or else the elapsed time, as soon as the program completes. (If you are going to print elapsed time, you can use `time.time()`, which returns the number of seconds since a fixed date.) These tools make it possible to time operations precisely, and thus optimize speed even on a small dataset that takes no more than a few seconds to process:

```
import time
StartTime = time.time()
# perform your commands here
print "Elapsed: %.5f" % (time.time() - StartTime)
```

However, if you are trying to highly optimize a very small piece of code on a very small dataset (presumably as a test before moving to a larger dataset), the elapsed time may indeed be too short to measure precisely. To get around this, you can nest your program within a loop to repeat it a thousand times or so. This will reveal the effects of your optimizations more clearly.

Look for calculations that are done repeatedly inside a loop, even though the value does not change between loop iterations. Move these statements so they are assigned to a variable *before* entering the loop, and only recalculate when there is a chance the value will change. A similar rule applies to opening and closing files: you should not close and re-open a file each time you write a value to it. Open the file, then loop through to read or write, then close the file after exiting the loop.

`try` *and* `except` *to handle errors*

Several good tips for optimizing Python programs are discussed at `tinyurl.com/pcfb-speed`. Among them are some counterintuitive suggestions. For example, in some cases your program may be written to check if a value is present in a list or dictionary before accessing it. This takes time, regardless of whether the value is there. A quicker method is to try to access the value directly without checking, and if you get an error, assume that it wasn't there. Python has an interesting pair of commands, `try` and `except`, which can be used in combination for this strategy. A block of code indented under `try:` executes as usual up until an error occurs, in which case the program immediately exits the block and proceeds to the next unindented statement. Immediately after the `try` statement, a block of code indented under `except:` will be executed only if an error has been triggered by the preceding `try` statement. If there is no error, the `except` block is not executed, and in this situation, the program essentially "gets away with" not performing a time-consuming check. The `except` statement can further specify the type of error that is encountered. If you specify `except KeyError:` that block will be executed when trying to access a key that is not present in a dictionary and `except IOError:` is invoked when files cannot be opened because they cannot be found.

Figure 13.1 is an illustration of the same fragment of a sequence-reading program fashioned two ways: the first uses a traditional `if` statement to test if a key is in a dictionary, and the second uses `try` and `except`. In this example, the program loops through strings and tries to append them to the end of existing dictionary values. If no value is present, it defines a new key–value pair with the string in question. For a data file containing 77,000 sequences and 500,000 lines, the first

(A) Traditional

```
for Line in File:
  if Line[0]==">":
    Name=Line.strip()[1:]
  # lines with > are Names
  else:
    # check for a pre-existing key
    if Name in Dict.keys():
      Dict[Name] +=  Line.strip()
    # not a key so define
    else:
      Dict[Name] = Line.strip()
```

(B) Faster

```
for Line in File:
  if Line[0]==">":
    Name=Line.strip()[1:]
  else:
    try:
      # try to append with +=
      # assumes Name is a key
      Dict[Name] +=  Line.strip()
      # oops, not a key so define
    except KeyError:
      Dict[Name] = Line.strip()
```

FIGURE 13.1 The same fragment of a sequence-reading program fashioned two ways (A) Traditional `if` statement tests if a key is in a dictionary. (B) Attempts to append the dictionary without checking, using `try` and `except`. Version A takes 2000 times as long to complete.

approach finished in 33 minutes while the second approach took 1 second—2000 times faster!

When you're really stuck

Some errors may just prove too elusive to solve on your own in a reasonable amount of time. A fresh pair of eyes can help spot problems you have become blind to. It also helps when you have to explain to someone else how your program operates. As you step through the code, explaining it to this person line by line, you may suddenly realize the nature of the flaw. True, this other person may not have the same experience with your program that you do—but on the other hand, they may not be blinded by the assumptions you have been making, as explained in the first part of this chapter.

It is tempting to blame the system when things aren't working, and sometimes it really is "the computer's fault." Try your program on a different computer, or under a different operating system; this may reveal that a key library or piece of software is missing from your own computer.

Before you get too frustrated, reach out to the online communities of programmers, including the forums at `practicalcomputing.org`. Usually they will be eager to help. Be sure to provide all the context needed for understanding your problem, including data files, your system configuration, and the text of the programs involved. You need to give enough information that someone else could replicate the error on their own system, if they chose. By contrast, if you post only a few isolated lines with a question like "Why doesn't this work???" you are unlikely to get help.

SUMMARY

You have learned how to:

- Question your assumptions when tackling bugs
- Write a program in incremental steps, checking that each part works before proceeding to the next
- Give yourself feedback with `print` statements or `sys.stderr.write()` within your code
- Use `"""comments"""` to turn on and off portions of your program
- Interpret and solve Python error messages
- Learn from your mistakes

Moving forward

- As you encounter and troubleshoot errors, keep track of them in a file with a name like `pythonhints.txt` that you keep in your `~/scripts` directory. This is also a good place to store handy Python commands, code examples, or tricky syntax (for example, list comprehension) that you find useful.
- Try to recognize when an error is due just to syntax (e.g., an indentation error) and when it is a flaw in the design of the analysis (e.g., your script fails to process the last line in a file). Syntax errors will become less common as you write more scripts, but you always have to be vigilant against analysis errors.
- For more in-depth debugging of Python programs, you can explore some of the Python debuggers available, including iPython, IDLE, and the PyDev plug-in for Eclipse.