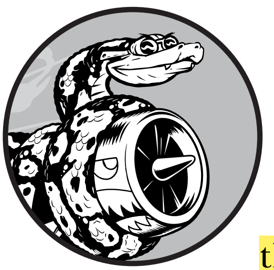
10

FILES AND EXCEPTIONS



Now that you've mastered the basic skills you need to write organized programs that are easy to use, it's time to think about making your programs even more relevant and usable. In this chapter you'll learn to work with files so your programs can quickly analyze lots of data. You'll learn to handle errors so your programs don't crash when they encounter unexpected situations. You'll learn about *exceptions*, which are special objects Python creates to manage errors that arise while a program is running. You'll also learn about the `json` module, which allows you to save user data so it isn't lost when your program stops running.

Learning to work with files and save data will make your programs easier for people to use. Users will be able to choose what data to enter and when to enter it. People can run your program, do some work, and then close the program and pick up where they left off later. Learning to handle exceptions will help you deal with situations in which files don't exist and deal with other problems that can cause your programs to crash. This will make your programs more robust when they encounter bad data, whether it comes from innocent mistakes or from malicious attempts to break your programs. With the skills you'll learn in this chapter, you'll make your programs more applicable, usable, and stable.

Reading from a File

An incredible amount of data is available in text files. Text files can contain weather data, traffic data, socioeconomic data, literary works, and more. Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file. For example, you can write a program that reads in the contents of a text file and rewrites the file with formatting that allows a browser to display it.

When you want to work with the information in a text file, the first step is to read the file into memory. You can read the entire contents of a file, or you can work through the file one line at a time.

Reading an Entire File

To begin, we need a file with a few lines of text in it. Let's start with a file that contains *pi* to 30 decimal places, with 10 decimal places per line:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

To try the following examples yourself, you can enter these lines in an editor and save the file as *pi_digits.txt*, or you can download the file from the book's resources through *https://nostarch.com/pythoncrashcourse2e/*. Save the file in the same directory where you'll store this chapter's programs.

Here's a program that opens this file, reads it, and prints the contents of the file to the screen:

file_reader.py

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
print(contents)
```

The first line of this program has a lot going on. Let's start by looking at the `open()` function. To do any work with a file, even just printing its contents, you first need to *open* the file to access it. The `open()` function needs one argument: the name of the file you want to open. Python looks for this file in the directory where the program that's currently being executed is stored. In this example, *file_reader.py* is currently running, so Python looks for *pi_digits.txt* in the directory where *file_reader.py* is stored. The `open()` function returns an object representing the file. Here, `open('pi_digits.txt')` returns an object representing *pi_digits.txt*. Python assigns this object to `file_object`, which we'll work with later in the program.

The keyword `with` closes the file once access to it is no longer needed. Notice how we call `open()` in this program but not `close()`. You could open and close the file by calling `open()` and `close()`, but if a bug in your program prevents the `close()` method from being executed, the file may never close. This may seem trivial, but improperly closed files can cause data to be lost or corrupted. And if you call `close()` too early in your program, you'll find yourself trying to work with a *closed* file (a file you can't access), which leads to more errors. It's not always easy to know exactly when you should close a file, but with the structure shown here, Python will figure that out for you. All you have to do is open the file and work with it as desired, trusting that Python will close it automatically when the `with` block finishes execution.

Once we have a file object representing *pi_digits.txt*, we use the `read()` method in the second line of our program to read the entire contents of the file and store it as one long string in `contents`. When we print the value of `contents`, we get the entire text file back:

```
3.1415926535
  8979323846
  2643383279

```

The only difference between this output and the original file is the extra blank line at the end of the output. The blank line appears because `read()` returns an empty string when it reaches the end of the file; this empty string shows up as a blank line. If you want to remove the extra blank line, you can use `rstrip()` in the call to `print()`:

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents.rstrip())
```

Recall that Python's `rstrip()` method removes, or strips, any whitespace characters from the right side of a string. Now the output matches the contents of the original file exactly:

```
3.1415926535
  8979323846
  2643383279
```

File Paths

When you pass a simple filename like *pi_digits.txt* to the `open()` function, Python looks in the directory where the file that's currently being executed (that is, your *.py* program file) is stored.

Sometimes, depending on how you organize your work, the file you want to open won't be in the same directory as your program file. For example, you might store your program files in a folder called *python_work*; inside *python_work*, you might have another folder called *text_files* to distinguish your program files from the text files they're manipulating. Even though *text_files* is in *python_work*, just passing `open()` the name of a file in *text_files* won't work, because Python will only look in *python_work* and stop there; it won't go on and look in *text_files*. To get Python to open files from a directory other than the one where your program file is stored, you need to provide a *file path*, which tells Python to look in a specific location on your system.

Because *text_files* is inside *python_work*, you could use a relative file path to open a file from *text_files*. A *relative file path* tells Python to look for a given location relative to the directory where the currently running program file is stored. For example, you'd write:

```
with open('text_files/filename.txt') as file_object:
```

This line tells Python to look for the desired *.txt* file in the folder *text_files* and assumes that *text_files* is located inside *python_work* (which it is).

NOTE *Windows systems use a backslash (\) instead of a forward slash (/) when displaying file paths, but you can still use forward slashes in your code.*

You can also tell Python exactly where the file is on your computer regardless of where the program that's being executed is stored. This is called an *absolute file path*. You use an absolute path if a relative path doesn't work. For instance, if you've put *text_files* in some folder other than *python_work*—say, a folder called *other_files*—then just passing `open()` the path `'text_files/filename.txt'` won't work because Python will only look for that location inside *python_work*. You'll need to write out a full path to clarify where you want Python to look.

Absolute paths are usually longer than relative paths, so it's helpful to assign them to a variable and then pass that variable to `open()`:

```
file_path = '/home/ehmatthes/other_files/text_files/filename.txt'
with open(file_path) as file_object:
```

Using absolute paths, you can read files from any location on your system. For now it's easiest to store files in the same directory as your program files or in a folder such as *text_files* within the directory that stores your program files.

NOTE *If you try to use backslashes in a file path, you'll get an error because the backslash is used to escape characters in strings. For example, in the path* `"C:\path\to\file.txt"`*, the sequence* `\t` *is interpreted as a tab. If you need to use backslashes, you can escape each one in the path, like this:* `"C:\\path\\to\\file.txt"`.

Reading Line by Line

When you're reading a file, you'll often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way. For example, you might want to read through a file of weather data and work with any line that includes the word *sunny* in the description of that day's weather. In a news report, you might look for any line with the tag `<headline>` and rewrite that line with a specific kind of formatting.

You can use a `for` loop on the file object to examine each line from a file one at a time:

file_reader.py

```
❶ filename = 'pi_digits.txt'

❷ with open(filename) as file_object:
❸     for line in file_object:
          print(line)
```

At ❶ we assign the name of the file we're reading from to the variable `filename`. This is a common convention when working with files. Because the variable `filename` doesn't represent the actual file—it's just a string telling Python where to find the file—you can easily swap out `'pi_digits.txt'` for the name of another file you want to work with. After we call `open()`, an object representing the file and its contents is assigned to the variable `file_object` ❷. We again use the `with` syntax to let Python open and close the file properly. To examine the file's contents, we work through each line in the file by looping over the file object ❸.

When we print each line, we find even more blank lines:

```
3.1415926535

  8979323846

  2643383279

```

These blank lines appear because an invisible newline character is at the end of each line in the text file. The `print` function adds its own newline each time we call it, so we end up with two newline characters at the end of each line: one from the file and one from `print()`. Using `rstrip()` on each line in the `print()` call eliminates these extra blank lines:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Now the output matches the contents of the file once again:

```
3.1415926535
  8979323846
  2643383279
```

Making a List of Lines from a File

When you use `with`, the file object returned by `open()` is only available inside the `with` block that contains it. If you want to retain access to a file's contents outside the `with` block, you can store the file's lines in a list inside the block and then work with that list. You can process parts of the file immediately and postpone some processing for later in the program.

The following example stores the lines of *pi_digits.txt* in a list inside the `with` block and then prints the lines outside the `with` block:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
❶     lines = file_object.readlines()

❷ for line in lines:
    print(line.rstrip())
```

At ❶ the `readlines()` method takes each line from the file and stores it in a list. This list is then assigned to `lines`, which we can continue to work with after the `with` block ends. At ❷ we use a simple `for` loop to print each line from `lines`. Because each item in `lines` corresponds to each line in the file, the output matches the contents of the file exactly.

Working with a File's Contents

After you've read a file into memory, you can do whatever you want with that data, so let's briefly explore the digits of *pi*. First, we'll attempt to build a single string containing all the digits in the file with no whitespace in it:

pi_string.py

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

❶ pi_string = ''
❷ for line in lines:
    pi_string += line.rstrip()

❸ print(pi_string)
print(len(pi_string))
```

We start by opening the file and storing each line of digits in a list, just as we did in the previous example. At ❶ we create a variable, `pi_string`, to hold the digits of *pi*. We then create a loop that adds each line of digits to `pi_string` and removes the newline character from each line ❷. At ❸ we print this string and also show how long the string is:

```
3.1415926535  8979323846  2643383279
36
```

The variable `pi_string` contains the whitespace that was on the left side of the digits in each line, but we can get rid of that by using `strip()` instead of `rstrip()`:

```
--snip--
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

Now we have a string containing *pi* to 30 decimal places. The string is 32 characters long because it also includes the leading 3 and a decimal point:

```
3.141592653589793238462643383279
32
```

NOTE *When Python reads from a text file, it interprets all text in the file as a string. If you read in a number and want to work with that value in a numerical context, you'll have to convert it to an integer using the* `int()` *function or convert it to a float using the* `float()` *function.*

Large Files: One Million Digits

So far we've focused on analyzing a text file that contains only three lines, but the code in these examples would work just as well on much larger files. If we start with a text file that contains *pi* to 1,000,000 decimal places instead of just 30, we can create a single string containing all these digits. We don't need to change our program at all except to pass it a different file. We'll also print just the first 50 decimal places, so we don't have to watch a million digits scroll by in the terminal:

pi_string.py

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()
```

```
pi_string = ''
for line in lines:
    pi_string += line.strip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

The output shows that we do indeed have a string containing *pi* to 1,000,000 decimal places:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python has no inherent limit to how much data you can work with; you can work with as much data as your system's memory can handle.

NOTE *To run this program (and many of the examples that follow), you'll need to download the resources available at* https://nostarch.com/pythoncrashcourse2e/.

Is Your Birthday Contained in Pi?

I've always been curious to know if my birthday appears anywhere in the digits of *pi*. Let's use the program we just wrote to find out if someone's birthday appears anywhere in the first million digits of *pi*. We can do this by expressing each birthday as a string of digits and seeing if that string appears anywhere in `pi_string`:

```
--snip--
for line in lines:
    pi_string += line.strip()

❶ birthday = input("Enter your birthday, in the form mmddyy: ")
❷ if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi.")
```

At ❶ we prompt for the user's birthday, and then at ❷ we check if that string is in `pi_string`. Let's try it:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

My birthday does appear in the digits of *pi*! Once you've read from a file, you can analyze its contents in just about any way you can imagine.

TRY IT YOURSELF

10-1. Learning Python: Open a blank file in your text editor and write a few lines summarizing what you've learned about Python so far. Start each line with the phrase *In Python you can. . . .* Save the file as *learning_python.txt* in the same directory as your exercises from this chapter. Write a program that reads the file and prints what you wrote three times. Print the contents once by reading in the entire file, once by looping over the file object, and once by storing the lines in a list and then working with them outside the `with` block.

10-2. Learning C: You can use the `replace()` method to replace any word in a string with a different word. Here's a quick example showing how to replace `'dog'` with `'cat'` in a sentence:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Read in each line from the file you just created, *learning_python.txt*, and replace the word *Python* with the name of another language, such as *C*. Print each modified line to the screen.

Writing to a File

One of the simplest ways to save data is to write it to a file. When you write text to a file, the output will still be available after you close the terminal containing your program's output. You can examine output after a program finishes running, and you can share the output files with others as well. You can also write programs that read the text back into memory and work with it again later.

Writing to an Empty File

To write text to a file, you need to call `open()` with a second argument telling Python that you want to write to the file. To see how this works, let's write a simple message and store it in a file instead of printing it to the screen:

write_message.py

```
filename = 'programming.txt'

❶ with open(filename, 'w') as file_object:
❷     file_object.write("I love programming.")
```

The call to `open()` in this example has two arguments ❶. The first argument is still the name of the file we want to open. The second argument, `'w'`, tells Python that we want to open the file in *write mode*. You can open a file in *read mode* ('r'), *write mode* ('w'), *append mode* ('a'), or a mode that allows you to read and write to the file ('r+'). If you omit the mode argument, Python opens the file in read-only mode by default.

The open() function automatically creates the file you're writing to if it doesn't already exist. However, be careful opening a file in write mode ('w') because if the file does exist, Python will erase the contents of the file before returning the file object.

At ❷ we use the write() method on the file object to write a string to the file. This program has no terminal output, but if you open the file *programming.txt*, you'll see one line:

programming.txt

```
I love programming.
```

This file behaves like any other file on your computer. You can open it, write new text in it, copy from it, paste to it, and so forth.

NOTE *Python can only write strings to a text file. If you want to store numerical data in a text file, you'll have to convert the data to string format first using the* str() *function.*

Writing Multiple Lines

The write() function doesn't add any newlines to the text you write. So if you write more than one line without including newline characters, your file may not look the way you want it to:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating new games.")
```

If you open *programming.txt*, you'll see the two lines squished together:

```
I love programming.I love creating new games.
```

Including newlines in your calls to write() makes each string appear on its own line:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
```

The output now appears on separate lines:

```
I love programming.
I love creating new games.
```

You can also use spaces, tab characters, and blank lines to format your output, just as you've been doing with terminal-based output.

Appending to a File

If you want to add content to a file instead of writing over existing content, you can open the file in *append mode*. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object. Any lines you write to the file will be added at the end of the file. If the file doesn't exist yet, Python will create an empty file for you.

Let's modify *write_message.py* by adding some new reasons we love programming to the existing file *programming.txt*:

write_message.py

```
filename = 'programming.txt'

❶ with open(filename, 'a') as file_object:
❷     file_object.write("I also love finding meaning in large datasets.\n")
      file_object.write("I love creating apps that can run in a browser.\n")
```

At ❶ we use the `'a'` argument to open the file for appending rather than writing over the existing file. At ❷ we write two new lines, which are added to *programming.txt*:

programming.txt

```
I love programming.
I love creating new games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

We end up with the original contents of the file, followed by the new content we just added.

TRY IT YOURSELF

10-3. Guest: Write a program that prompts the user for their name. When they respond, write their name to a file called *guest.txt*.

10-4. Guest Book: Write a `while` loop that prompts users for their name. When they enter their name, print a greeting to the screen and add a line recording their visit in a file called *guest_book.txt*. Make sure each entry appears on a new line in the file.

10-5. Programming Poll: Write a `while` loop that asks people why they like programming. Each time someone enters a reason, add their reason to a file that stores all the responses.

Exceptions

Python uses special objects called *exceptions* to manage errors that arise during a program's execution. Whenever an error occurs that makes Python unsure what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a *traceback*, which includes a report of the exception that was raised.

Exceptions are handled with `try-except` blocks. A `try-except` block asks Python to do something, but it also tells Python what to do if an exception is raised. When you use `try-except` blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

Handling the ZeroDivisionError Exception

Let's look at a simple error that causes Python to raise an exception. You probably know that it's impossible to divide a number by zero, but let's ask Python to do it anyway:

division_calculator.py

```
print(5/0)
```

Of course Python can't do this, so we get a traceback:

```
Traceback (most recent call last):
  File "division_calculator.py", line 1, in <module>
    print(5/0)
❶ ZeroDivisionError: division by zero
```

The error reported at ❶ in the traceback, `ZeroDivisionError`, is an exception object. Python creates this kind of object in response to a situation where it can't do what we ask it to. When this happens, Python stops the program and tells us the kind of exception that was raised. We can use this information to modify our program. We'll tell Python what to do when this kind of exception occurs; that way, if it happens again, we're prepared.

Using try-except Blocks

When you think an error may occur, you can write a `try-except` block to handle the exception that might be raised. You tell Python to try running some code, and you tell it what to do if the code results in a particular kind of exception.

Here's what a `try-except` block for handling the `ZeroDivisionError` exception looks like:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised and runs the code in that block.

In this example, the code in the `try` block produces a `ZeroDivisionError`, so Python looks for an `except` block telling it how to respond. Python then runs the code in that block, and the user sees a friendly error message instead of a traceback:

```
You can't divide by zero!
```

If more code followed the `try`-`except` block, the program would continue running because we told Python how to handle the error. Let's look at an example where catching an error can allow a program to continue running.

Using Exceptions to Prevent Crashes

Handling errors correctly is especially important when the program has more work to do after the error occurs. This happens often in programs that prompt users for input. If the program responds to invalid input appropriately, it can prompt for more valid input instead of crashing.

Let's create a simple calculator that does only division:

division_calculator.py

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
❶     first_number = input("\nFirst number: ")
      if first_number == 'q':
          break
❷     second_number = input("Second number: ")
      if second_number == 'q':
          break
❸     answer = int(first_number) / int(second_number)
      print(answer)
```

This program prompts the user to input a `first_number` ❶ and, if the user does not enter *q* to quit, a `second_number` ❷. We then divide these two numbers to get an `answer` ❸. This program does nothing to handle errors, so asking it to divide by zero causes it to crash:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division_calculator.py", line 9, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```