It's bad that the program crashed, but it's also not a good idea to let users see tracebacks. Nontechnical users will be confused by them, and in a malicious setting, attackers will learn more than you want them to know from a traceback. For example, they'll know the name of your program file, and they'll see a part of your code that isn't working properly. A skilled attacker can sometimes use this information to determine which kind of attacks to use against your code.

The else Block

We can make this program more error resistant by wrapping the line that might produce errors in a `try-except` block. The error occurs on the line that performs the division, so that's where we'll put the `try-except` block. This example also includes an `else` block. Any code that depends on the `try` block executing successfully goes in the `else` block:

```
--snip--
while True:
    --snip--
    if second_number == 'q':
        break
❶   try:
        answer = int(first_number) / int(second_number)
❷   except ZeroDivisionError:
        print("You can't divide by 0!")
❸   else:
        print(answer)
```

We ask Python to try to complete the division operation in a `try` block ❶, which includes only the code that might cause an error. Any code that depends on the `try` block succeeding is added to the `else` block. In this case if the division operation is successful, we use the `else` block to print the result ❸.

The `except` block tells Python how to respond when a `ZeroDivisionError` arises ❷. If the `try` block doesn't succeed because of a division by zero error, we print a friendly message telling the user how to avoid this kind of error. The program continues to run, and the user never sees a traceback:

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

First number: 5
Second number: 0
You can't divide by 0!
```

```
First number: 5
Second number: 2
2.5

First number: q
```

The `try-except-else` block works like this: Python attempts to run the code in the `try` block. The only code that should go in a `try` block is code that might cause an exception to be raised. Sometimes you'll have additional code that should run only if the `try` block was successful; this code goes in the `else` block. The `except` block tells Python what to do in case a certain exception arises when it tries to run the code in the `try` block.

By anticipating likely sources of errors, you can write robust programs that continue to run even when they encounter invalid data and missing resources. Your code will be resistant to innocent user mistakes and malicious attacks.

Handling the FileNotFoundError Exception

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all. You can handle all of these situations in a straightforward way with a `try-except` block.

Let's try to read a file that doesn't exist. The following program tries to read in the contents of *Alice in Wonderland*, but I haven't saved the file *alice.txt* in the same directory as *alice.py*:

alice.py

```
filename = 'alice.txt'

with open(filename, encoding='utf-8') as f:
    contents = f.read()
```

There are two changes here. One is the use of the variable `f` to represent the file object, which is a common convention. The second is the use of the `encoding` argument. This argument is needed when your system's default encoding doesn't match the encoding of the file that's being read.

Python can't read from a missing file, so it raises an exception:

```
Traceback (most recent call last):
  File "alice.py", line 3, in <module>
    with open(filename, encoding='utf-8') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

The last line of the traceback reports a `FileNotFoundError`: this is the exception Python creates when it can't find the file it's trying to open.

In this example, the `open()` function produces the error, so to handle it, the `try` block will begin with the line that contains `open()`:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
```

In this example, the code in the `try` block produces a `FileNotFoundError`, so Python looks for an `except` block that matches that error. Python then runs the code in that block, and the result is a friendly error message instead of a traceback:

```
Sorry, the file alice.txt does not exist.
```

The program has nothing more to do if the file doesn't exist, so the error-handling code doesn't add much to this program. Let's build on this example and see how exception handling can help when you're working with more than one file.

Analyzing Text

You can analyze text files containing entire books. Many classic works of literature are available as simple text files because they are in the public domain. The texts used in this section come from Project Gutenberg (*http://gutenberg.org/*). Project Gutenberg maintains a collection of literary works that are available in the public domain, and it's a great resource if you're interested in working with literary texts in your programming projects.

Let's pull in the text of *Alice in Wonderland* and try to count the number of words in the text. We'll use the string method `split()`, which can build a list of words from a string. Here's what `split()` does with a string containing just the title `"Alice in Wonderland"`:

```
>>> title = "Alice in Wonderland"
>>> title.split()
['Alice', 'in', 'Wonderland']
```

The `split()` method separates a string into parts wherever it finds a space and stores all the parts of the string in a list. The result is a list of words from the string, although some punctuation may also appear with some of the words. To count the number of words in *Alice in Wonderland*, we'll use `split()` on the entire text. Then we'll count the items in the list to get a rough idea of the number of words in the text:

```
filename = 'alice.txt'

try:
```

```
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
else:
    # Count the approximate number of words in the file.
❶   words = contents.split()
❷   num_words = len(words)
❸   print(f"The file {filename} has about {num_words} words.")
```

I moved the file *alice.txt* to the correct directory, so the `try` block will work this time. At ❶ we take the string `contents`, which now contains the entire text of *Alice in Wonderland* as one long string, and use the `split()` method to produce a list of all the words in the book. When we use `len()` on this list to examine its length, we get a good approximation of the number of words in the original string ❷. At ❸ we print a statement that reports how many words were found in the file. This code is placed in the `else` block because it will work only if the code in the `try` block was executed successfully. The output tells us how many words are in *alice.txt*:

```
The file alice.txt has about 29465 words.
```

The count is a little high because extra information is provided by the publisher in the text file used here, but it's a good approximation of the length of *Alice in Wonderland*.

Working with Multiple Files

Let's add more books to analyze. But before we do, let's move the bulk of this program to a function called `count_words()`. By doing so, it will be easier to run the analysis for multiple books:

word_count.py

```
def count_words(filename):
❶   """Count the approximate number of words in a file."""
    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"Sorry, the file {filename} does not exist.")
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has about {num_words} words.")

filename = 'alice.txt'
count_words(filename)
```

Most of this code is unchanged. We simply indented it and moved it into the body of `count_words()`. It's a good habit to keep comments up to date when you're modifying a program, so we changed the comment to a docstring and reworded it slightly ❶.

Now we can write a simple loop to count the words in any text we want to analyze. We do this by storing the names of the files we want to analyze in a list, and then we call `count_words()` for each file in the list. We'll try to count the words for *Alice in Wonderland*, *Siddhartha*, *Moby Dick*, and *Little Women*, which are all available in the public domain. I've intentionally left *siddhartha.txt* out of the directory containing *word_count.py*, so we can see how well our program handles a missing file:

```
def count_words(filename):
    --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

The missing *siddhartha.txt* file has no effect on the rest of the program's execution:

```
The file alice.txt has about 29465 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

Using the `try-except` block in this example provides two significant advantages. We prevent our users from seeing a traceback, and we let the program continue analyzing the texts it's able to find. If we don't catch the `FileNotFoundError` that *siddhartha.txt* raised, the user would see a full traceback, and the program would stop running after trying to analyze *Siddhartha*. It would never analyze *Moby Dick* or *Little Women*.

Failing Silently

In the previous example, we informed our users that one of the files was unavailable. But you don't need to report every exception you catch. Sometimes you'll want the program to fail silently when an exception occurs and continue on as if nothing happened. To make a program fail silently, you write a `try` block as usual, but you explicitly tell Python to do nothing in the `except` block. Python has a `pass` statement that tells it to do nothing in a block:

```
def count_words(filename):
    """Count the approximate number of words in a file."""
    try:
        --snip--
    except FileNotFoundError:
❶       pass
    else:
        --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

The only difference between this listing and the previous one is the `pass` statement at ❶. Now when a `FileNotFoundError` is raised, the code in the `except` block runs, but nothing happens. No traceback is produced, and there's no output in response to the error that was raised. Users see the word counts for each file that exists, but they don't see any indication that a file wasn't found:

```
The file alice.txt has about 29465 words.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

The `pass` statement also acts as a placeholder. It's a reminder that you're choosing to do nothing at a specific point in your program's execution and that you might want to do something there later. For example, in this program we might decide to write any missing filenames to a file called *missing_files.txt*. Our users wouldn't see this file, but we'd be able to read the file and deal with any missing texts.

Deciding Which Errors to Report

How do you know when to report an error to your users and when to fail silently? If users know which texts are supposed to be analyzed, they might appreciate a message informing them why some texts were not analyzed. If users expect to see some results but don't know which books are supposed to be analyzed, they might not need to know that some texts were unavailable. Giving users information they aren't looking for can decrease the usability of your program. Python's error-handling structures give you fine-grained control over how much to share with users when things go wrong; it's up to you to decide how much information to share.

Well-written, properly tested code is not very prone to internal errors, such as syntax or logical errors. But every time your program depends on something external, such as user input, the existence of a file, or the availability of a network connection, there is a possibility of an exception being raised. A little experience will help you know where to include exception handling blocks in your program and how much to report to users about errors that arise.

TRY IT YOURSELF

10-6. Addition: One common problem when prompting for numerical input occurs when people provide text instead of numbers. When you try to convert the input to an `int`, you'll get a `ValueError`. Write a program that prompts for two numbers. Add them together and print the result. Catch the `ValueError` if either input value is not a number, and print a friendly error message. Test your program by entering two numbers and then by entering some text instead of a number.

(continued)

10-7. Addition Calculator: Wrap your code from Exercise 10-6 in a `while` loop so the user can continue entering numbers even if they make a mistake and enter text instead of a number.

10-8. Cats and Dogs: Make two files, *cats.txt* and *dogs.txt*. Store at least three names of cats in the first file and three names of dogs in the second file. Write a program that tries to read these files and print the contents of the file to the screen. Wrap your code in a `try-except` block to catch the `FileNotFound` error, and print a friendly message if a file is missing. Move one of the files to a different location on your system, and make sure the code in the `except` block executes properly.

10-9. Silent Cats and Dogs: Modify your `except` block in Exercise 10-8 to fail silently if either file is missing.

10-10. Common Words: Visit Project Gutenberg (*https://gutenberg.org/*) and find a few texts you'd like to analyze. Download the text files for these works, or copy the raw text from your browser into a text file on your computer.

You can use the `count()` method to find out how many times a word or phrase appears in a string. For example, the following code counts the number of times `'row'` appears in a string:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Notice that converting the string to lowercase using `lower()` catches all appearances of the word you're looking for, regardless of how it's formatted.

Write a program that reads the files you found at Project Gutenberg and determines how many times the word `'the'` appears in each text. This will be an approximation because it will also count words such as `'then'` and `'there'`. Try counting `'the '`, with a space in the string, and see how much lower your count is.

Storing Data

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for a visualization. Whatever the focus of your program is, you'll store the information users provide in data structures such as lists and dictionaries. When users close a program, you'll almost always want to save the information they entered. A simple way to do this involves storing your data using the `json` module.

The `json` module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs. You can also use `json` to share data between different Python programs. Even better, the JSON data format is not specific to Python, so you can share data you store in the JSON format with people who work in many other programming languages. It's a useful and portable format, and it's easy to learn.

NOTE *The JSON (JavaScript Object Notation) format was originally developed for JavaScript. However, it has since become a common format used by many languages, including Python.*

Using json.dump() and json.load()

Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use `json.dump()` to store the set of numbers, and the second program will use `json.load()`.

The `json.dump()` function takes two arguments: a piece of data to store and a file object it can use to store the data. Here's how you can use `json.dump()` to store a list of numbers:

number_writer.py

```
import json

numbers = [2, 3, 5, 7, 11, 13]

❶ filename = 'numbers.json'
❷ with open(filename, 'w') as f:
❸     json.dump(numbers, f)
```

We first import the `json` module and then create a list of numbers to work with. At ❶ we choose a filename in which to store the list of numbers. It's customary to use the file extension *.json* to indicate that the data in the file is stored in the JSON format. Then we open the file in write mode, which allows `json` to write the data to the file ❷. At ❸ we use the `json.dump()` function to store the list `numbers` in the file *numbers.json*.

This program has no output, but let's open the file *numbers.json* and look at it. The data is stored in a format that looks just like Python:

```
[2, 3, 5, 7, 11, 13]
```

Now we'll write a program that uses `json.load()` to read the list back into memory:

number_reader.py

```
import json

❶ filename = 'numbers.json'
❷ with open(filename) as f:
❸     numbers = json.load(f)

print(numbers)
```

At ❶ we make sure to read from the same file we wrote to. This time when we open the file, we open it in read mode because Python only needs to read from the file ❷. At ❸ we use the `json.load()` function to load the information stored in *numbers.json*, and we assign it to the variable `numbers`. Finally we print the recovered list of numbers and see that it's the same list created in *number_writer.py*:

```
[2, 3, 5, 7, 11, 13]
```

This is a simple way to share data between two programs.

Saving and Reading User-Generated Data

Saving data with `json` is useful when you're working with user-generated data, because if you don't store your user's information somehow, you'll lose it when the program stops running. Let's look at an example where we prompt the user for their name the first time they run a program and then remember their name when they run the program again.

Let's start by storing the user's name:

remember_me.py

```
import json

❶ username = input("What is your name? ")

filename = 'username.json'
with open(filename, 'w') as f:
❷     json.dump(username, f)
❸     print(f"We'll remember you when you come back, {username}!")
```

At ❶ we prompt for a username to store. Next, we use `json.dump()`, passing it a username and a file object, to store the username in a file ❷. Then we print a message informing the user that we've stored their information ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Now let's write a new program that greets a user whose name has already been stored:

greet_user.py

```
import json

filename = 'username.json'

with open(filename) as f:
❶     username = json.load(f)
❷     print(f"Welcome back, {username}!")
```

At ❶ we use `json.load()` to read the information stored in *username.json* and assign it to the variable `username`. Now that we've recovered the username, we can welcome them back ❷:

```
Welcome back, Eric!
```

We need to combine these two programs into one file. When someone runs *remember_me.py*, we want to retrieve their username from memory if possible; therefore, we'll start with a `try` block that attempts to recover the username. If the file *username.json* doesn't exist, we'll have the `except` block prompt for a username and store it in *username.json* for next time:

remember_me.py

```
import json

# Load the username, if it has been stored previously.
#  Otherwise, prompt for the username and store it.
filename = 'username.json'
try:
❶     with open(filename) as f:
❷         username = json.load(f)
❸ except FileNotFoundError:
❹     username = input("What is your name? ")
❺     with open(filename, 'w') as f:
        json.dump(username, f)
        print(f"We'll remember you when you come back, {username}!")
else:
    print(f"Welcome back, {username}!")
```

There's no new code here; blocks of code from the last two examples are just combined into one file. At ❶ we try to open the file *username.json*. If this file exists, we read the username back into memory ❷ and print a message welcoming back the user in the `else` block. If this is the first time the user runs the program, *username.json* won't exist and a `FileNotFoundError` will occur ❸. Python will move on to the `except` block where we prompt the user to enter their username ❹. We then use `json.dump()` to store the username and print a greeting ❺.

Whichever block executes, the result is a username and an appropriate greeting. If this is the first time the program runs, this is the output:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Otherwise:

```
Welcome back, Eric!
```

This is the output you see if the program was already run at least once.

Refactoring

Often, you'll come to a point where your code will work, but you'll recognize that you could improve the code by breaking it up into a series of functions that have specific jobs. This process is called *refactoring*. Refactoring makes your code cleaner, easier to understand, and easier to extend.

We can refactor *remember_me.py* by moving the bulk of its logic into one or more functions. The focus of *remember_me.py* is on greeting the user, so let's move all of our existing code into a function called `greet_user()`:

remember_me.py

```
import json

def greet_user():
❶    """Greet the user by name."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        username = input("What is your name? ")
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")
    else:
        print(f"Welcome back, {username}!")

greet_user()
```

Because we're using a function now, we update the comments with a docstring that reflects how the program currently works ❶. This file is a little cleaner, but the function `greet_user()` is doing more than just greeting the user—it's also retrieving a stored username if one exists and prompting for a new username if one doesn't exist.

Let's refactor `greet_user()` so it's not doing so many different tasks. We'll start by moving the code for retrieving a stored username to a separate function:

```
import json

def get_stored_username():
❶    """Get stored username if available."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
❷        return None
    else:
        return username
```

```
def greet_user():
    """Greet the user by name."""
    username = get_stored_username()
❸   if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        filename = 'username.json'
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")

greet_user()
```

The new function `get_stored_username()` has a clear purpose, as stated in the docstring at ❶. This function retrieves a stored username and returns the username if it finds one. If the file *username.json* doesn't exist, the function returns `None` ❷. This is good practice: a function should either return the value you're expecting, or it should return `None`. This allows us to perform a simple test with the return value of the function. At ❸ we print a welcome back message to the user if the attempt to retrieve a username was successful, and if it doesn't, we prompt for a new username.

We should factor one more block of code out of `greet_user()`. If the username doesn't exist, we should move the code that prompts for a new username to a function dedicated to that purpose:

```
import json

def get_stored_username():
    """Get stored username if available."""
    --snip--

def get_new_username():
    """Prompt for a new username."""
    username = input("What is your name? ")
    filename = 'username.json'
    with open(filename, 'w') as f:
        json.dump(username, f)
    return username

def greet_user():
    """Greet the user by name."""
    username = get_stored_username()
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = get_new_username()
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Each function in this final version of *remember_me.py* has a single, clear purpose. We call `greet_user()`, and that function prints an appropriate message: it either welcomes back an existing user or greets a new user. It does this by calling `get_stored_username()`, which is responsible only for retrieving a stored username if one exists. Finally, `greet_user()` calls `get_new_username()` if necessary, which is responsible only for getting a new username and storing it. This compartmentalization of work is an essential part of writing clear code that will be easy to maintain and extend.

TRY IT YOURSELF

10-11. Favorite Number: Write a program that prompts for the user's favorite number. Use `json.dump()` to store this number in a file. Write a separate program that reads in this value and prints the message, "I know your favorite number! It's _____."

10-12. Favorite Number Remembered: Combine the two programs from Exercise 10-11 into one file. If the number is already stored, report the favorite number to the user. If not, prompt for the user's favorite number and store it in a file. Run the program twice to see that it works.

10-13. Verify User: The final listing for *remember_me.py* assumes either that the user has already entered their username or that the program is running for the first time. We should modify it in case the current user is not the person who last used the program.

Before printing a welcome back message in `greet_user()`, ask the user if this is the correct username. If it's not, call `get_new_username()` to get the correct username.

Summary

In this chapter, you learned how to work with files. You learned to read an entire file at once and read through a file's contents one line at a time. You learned to write to a file and append text onto the end of a file. You read about exceptions and how to handle the exceptions you're likely to see in your programs. Finally, you learned how to store Python data structures so you can save information your users provide, preventing them from having to start over each time they run a program.

In Chapter 11 you'll learn efficient ways to test your code. This will help you trust that the code you develop is correct, and it will help you identify bugs that are introduced as you continue to build on the programs you've written.