RUNNING PYTHON SCRIPTS OUTSIDE OF IDLE

So far, you've been running your Python scripts using the interactive shell and file editor in IDLE. However, you won't want to go through the inconvenience of opening IDLE and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, OS X, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using IDLE.)

Project: Password Locker

You probably have accounts on many different websites. It's a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts. It's best to use password manager software on your computer that uses one master password to unlock the password manager. Then you can copy any account password to the clipboard and paste it into the website's Password field.

The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

THE CHAPTER PROJECTS

This is the first "chapter project" of the book. From here on, each chapter will have projects that demonstrate the concepts covered in the chapter. The projects are written in a style that takes you from a blank file editor window to a full, working program. Just like with the interactive shell examples, don't only read the project sections—follow along on your computer!

Step 1: Program Design and Data Structures

You want to be able to run this program with a command line argument that is the account's name—for instance, *email* or *blog*. That account's password will be copied to the clipboard so that the user can paste it into a Password field. This way, the user can have long, complicated passwords without having to memorize them.

Open a new file editor window and save the program as *pw.py*. You need to start the program with a `#!` (*shebang*) line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each account's name with its password, you can store these as

strings in a dictionary. The dictionary will be the data structure that organizes your account and password data. Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable `sys.argv`. (See Appendix B for more information on how to use command line arguments in your programs.) The first item in the `sys.argv` list should always be a string containing the program's filename (`'pw.py'`), and the second item should be the first command line argument. For this program, this argument is the name of the account whose password you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the `sys.argv` list has fewer than two values in it). Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name
```

Step 3: Copy the Right Password

Now that the account name is stored as a string in the variable `account`, you need to see whether it exists in the `PASSWORDS` dictionary as a key. If so, you want to copy the key's value to the clipboard using `pyperclip.copy()`. (Since you're using the `pyperclip` module, you need to import it.) Note that you don't actually *need* the `account` variable; you could just use `sys.argv[1]` everywhere `account` is used in this program. But a variable named `account` is much more readable than something cryptic like `sys.argv[1]`.

Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.
```

```
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

This new code looks in the `PASSWORDS` dictionary for the account name. If the account name is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no account with that name.

That's the complete script. Using the instructions in Appendix B for launching command line programs easily, you now have a fast way to copy your account passwords to the clipboard. You will have to modify the `PASSWORDS` dictionary value in the source whenever you want to update the program with a new password.

Of course, you probably don't want to keep all your passwords in one place where anyone could easily copy them. But you can modify this program and use it to quickly copy regular text to the clipboard. Say you are sending out several emails that have many of the same stock paragraphs in common. You could put each paragraph as a value in the `PASSWORDS` dictionary (you'd probably want to rename the dictionary at this point), and then you would have a way to quickly select and copy one of many standard pieces of text to the clipboard.

On Windows, you can create a batch file to run this program with the WIN-R Run window. (For more about batch files, see Appendix B.) Type the following into the file editor and save the file as *pw.bat* in the *C:\Windows* folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

With this batch file created, running the password-safe program on Windows is just a matter of pressing WIN-R and typing `pw <account name>`.

Project: Adding Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front. But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, if I copied the following text (for the Wikipedia article "List of Lists of Lists") to the clipboard:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

1. Paste text from the clipboard
2. Do something to it
3. Copy the new text to the clipboard

That second step is a little tricky, but steps 1 and 3 are pretty straightforward: They just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let's just write the part of the program that covers steps 1 and 3. Enter the following, saving the program as *bulletPointAdder.py*:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
```

```
# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

The `TODO` comment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text and Add the Star

The call to `pyperclip.paste()` returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in `text` would look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

The `\n` newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard. There are many "lines" in this one string value. You want to add a star to the start of each of these lines.

You could write code that searches for each `\n` newline character in the string and then adds the star just after that. But it would be easier to use the `split()` method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in `lines` and then loop through the items in `lines`. For each line, we add a star and a space to the start of the line. Now each string in `lines` begins with a star.

Step 3: Join the Modified Lines

The `lines` list now contains modified lines that start with stars. But `pyperclip.copy()` is expecting a single string value, not a list of string values. To make this single string value, pass `lines` into the `join()` method to get a single string joined from the list's strings. Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes for "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don't need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.

Summary

Text is a common form of data, and Python comes with many helpful string methods to process the text stored in string values. You will make use of indexing, slicing, and string methods in almost every Python program you write.

The programs you are writing now don't seem too sophisticated—they don't have graphical user interfaces with images and colorful text. So far, you're displaying text with `print()` and letting the user enter text with `input()`. However, the user can quickly enter large amounts of text through the clipboard. This ability provides a useful avenue for writing programs that manipulate massive amounts of text. These text-based programs might not have flashy windows or graphics, but they can get a lot of useful work done quickly.

Another way to manipulate large amounts of text is reading and writing files directly off the hard drive. You'll learn how to do this with Python in the next chapter.

And since Python scripts are themselves just text files with the *.py* file extension, your Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a *.py* file (as opposed to saving variables with the `shelve` module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the `shelve` module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. `File` objects, for example, cannot be encoded as text.

Project: Generating Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

- Creates 35 different quizzes.
- Creates 50 multiple-choice questions for each quiz, in random order.
- Provides the correct answer and three random wrong answers for each question, in random order.
- Writes the quizzes to 35 text files.
- Writes the answer keys to 35 text files.

This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call `open()`, `write()`, and `close()` for the quiz and answer key text files.
- Use `random.shuffle()` to randomize the order of the questions and multiple-choice options.

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

❶ import random

# The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
❸ for quizNum in range(35):
    # TODO: Create the quiz and answer key files.

    # TODO: Write out the header for the quiz.

    # TODO: Shuffle the order of the states.

    # TODO: Loop through all 50 states, making a question for each.
```

Since this program will be randomly ordering the questions and answers, you'll need to import the `random` module ❶ to make use of its functions. The `capitals` variable ❷ contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with `TODO` comments for now) will go inside a `for` loop that loops 35 times ❸. (This number can be changed to generate any number of quiz files.)

Step 2: Create the Quiz File and Shuffle the Question Order

Now it's time to start filling in those TODOs.

The code in the loop will be repeated 35 times—once for each quiz—so you have to worry about only one quiz at a time within the loop. First you'll create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period. Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Add the following lines of code to *randomQuizGenerator.py*:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
❶   quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
❷   answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Write out the header for the quiz.
❸   quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
    quizFile.write('\n\n')

    # Shuffle the order of the states.
    states = list(capitals.keys())
❹   random.shuffle(states)

    # TODO: Loop through all 50 states, making a question for each.
```

The filenames for the quizzes will be *capitalsquiz<N>.txt*, where *<N>* is a unique number for the quiz that comes from `quizNum`, the `for` loop's counter. The answer key for *capitalsquiz<N>.txt* will be stored in a text file named *capitalsquiz_answers<N>.txt*. Each time through the loop, the `%s` placeholder in `'capitalsquiz%s.txt'` and `'capitalsquiz_answers%s.txt'` will be replaced by `(quizNum + 1)`, so the first quiz and answer key created will be *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*. These files will be created with calls to the `open()` function at ❶ and ❷, with `'w'` as the second argument to open them in write mode.

The `write()` statements at ❸ create a quiz header for the student to fill out. Finally, a randomized list of US states is created with the help of the `random.shuffle()` function ❹, which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate the answer options for each question, which will be multiple choice from A to D. You'll need to create another `for` loop—this one to generate the content for each of the 50 questions on the quiz. Then there will be a third `for` loop nested inside to generate the multiple-choice options for each question. Make your code look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

    # Loop through all 50 states, making a question for each.
    for questionNum in range(50):

        # Get right and wrong answers.
❶       correctAnswer = capitals[states[questionNum]]
❷       wrongAnswers = list(capitals.values())
❸       del wrongAnswers[wrongAnswers.index(correctAnswer)]
❹       wrongAnswers = random.sample(wrongAnswers, 3)
❺       answerOptions = wrongAnswers + [correctAnswer]
❻       random.shuffle(answerOptions)

        # TODO: Write the question and answer options to the quiz file.

        # TODO: Write the answer key to a file.
```

The correct answer is easy to get—it's stored as a value in the `capitals` dictionary ❶. This loop will loop through the states in the shuffled `states` list, from `states[0]` to `states[49]`, find each state in `capitals`, and store that state's corresponding capital in `correctAnswer`.

The list of possible wrong answers is trickier. You can get it by duplicating *all* the values in the `capitals` dictionary ❷, deleting the correct answer ❸, and selecting three random values from this list ❹. The `random.sample()` function makes it easy to do this selection. Its first argument is the list you want to select from; the second argument is the number of values you want to select. The full list of answer options is the combination of these three wrong answers with the correct answers ❺. Finally, the answers need to be randomized ❻ so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--
```

```
    # Loop through all 50 states, making a question for each.
    for questionNum in range(50):
        --snip--

        # Write the question and the answer options to the quiz file.
        quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
            states[questionNum]))
❶       for i in range(4):
❷           quizFile.write('    %s. %s\n' % ('ABCD'[i], answerOptions[i]))
        quizFile.write('\n')

        # Write the answer key to a file.
❸       answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[
           answerOptions.index(correctAnswer)]))
    quizFile.close()
    answerKeyFile.close()
```

A `for` loop that goes through integers `0` to `3` will write the answer options in the `answerOptions` list ❶. The expression `'ABCD'[i]` at ❷ treats the string `'ABCD'` as an array and will evaluate to `'A'`,`'B'`, `'C'`, and then `'D'` on each respective iteration through the loop.

In the final line ❸, the expression `answerOptions.index(correctAnswer)` will find the integer index of the correct answer in the randomly ordered answer options, and `'ABCD'[answerOptions.index(correctAnswer)]` will evaluate to the correct answer's letter to be written to the answer key file.

After you run the program, this is how your *capitalsquiz1.txt* file will look, though of course your questions and answer options may be different from those shown here, depending on the outcome of your `random.shuffle()` calls:

```
Name:

Date:

Period:

                    State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
    A. Hartford
    B. Santa Fe
    C. Harrisburg
    D. Charleston

2. What is the capital of Colorado?
    A. Raleigh
    B. Harrisburg
    C. Denver
    D. Lincoln

--snip--
```

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

```
1. D
2. C
3. A
4. C
--snip--
```

Project: Multiclipboard

Say you have the boring task of filling out many forms in a web page or software with several text fields. The clipboard saves you from typing the same text over and over again. But only one thing can be on the clipboard at a time. If you have several different pieces of text that you need to copy and paste, you have to keep highlighting and copying the same few things over and over again.

You can write a Python program to keep track of multiple pieces of text. This "multiclipboard" will be named *mcb.pyw* (since "mcb" is shorter to type than "multiclipboard"). The *.pyw* extension means that Python won't show a Terminal window when it runs this program. (See Appendix B for more details.)

The program will save each piece of clipboard text under a keyword. For example, when you run `py mcb.pyw save spam`, the current contents of the clipboard will be saved with the keyword *spam*. This text can later be loaded to the clipboard again by running `py mcb.pyw spam`. And if the user forgets what keywords they have, they can run `py mcb.pyw list` to copy a list of all keywords to the clipboard.

Here's what the program does:

- The command line argument for the keyword is checked.
- If the argument is `save`, then the clipboard contents are saved to the keyword.
- If the argument is `list`, then all the keywords are copied to the clipboard.
- Otherwise, the text for the keyword is copied to the keyboard.

This means the code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Read and write to the clipboard.
- Save and load to a shelf file.

If you use Windows, you can easily run this script from the Run… window by creating a batch file named *mcb.bat* with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Step 1: Comments and Shelf Setup

Let's start by making a skeleton script with some comments and basic setup. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
#        py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
#        py.exe mcb.pyw list - Loads all keywords to clipboard.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

# TODO: Save clipboard content.

# TODO: List keywords and load content.

mcbShelf.close()
```

It's common practice to put general usage information in comments at the top of the file ❶. If you ever forget how to run your script, you can always look at these comments for a reminder. Then you import your modules ❷. Copying and pasting will require the `pyperclip` module, and reading the command line arguments will require the `sys` module. The `shelve` module will also come in handy: Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file. Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into your program. The shelf file will be named with the prefix *mcb* ❸.

Step 2: Save Clipboard Content with a Keyword

The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
❷         mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
❸     # TODO: List keywords and load content.

mcbShelf.close()
```

If the first command line argument (which will always be at index `1` of the `sys.argv` list) is `'save'` ❶, the second command line argument is the keyword for the current content of the clipboard. The keyword will be used as the key for `mcbShelf`, and the value will be the text currently on the clipboard ❷.

If there is only one command line argument, you will assume it is either `'list'` or a keyword to load content onto the clipboard. You will implement that code later. For now, just put a `TODO` comment there ❸.

Step 3: List Keywords and Load a Keyword's Content

Finally, let's implement the two remaining cases: The user wants to load clipboard text in from a keyword, or they want a list of all available keywords. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
        mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # List keywords and load content.
❶   if sys.argv[1].lower() == 'list':
❷       pyperclip.copy(str(list(mcbShelf.keys())))
    elif sys.argv[1] in mcbShelf:
❸       pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

If there is only one command line argument, first let's check whether it's `'list'` ❶. If so, a string representation of the list of shelf keys will be copied to the clipboard ❷. The user can paste this list into an open text editor to read it.

Otherwise, you can assume the command line argument is a keyword. If this keyword exists in the `mcbShelf` shelf as a key, you can load the value onto the clipboard ❸.

And that's it! Launching this program has different steps depending on what operating system your computer uses. See Appendix B for details for your operating system.

Recall the password locker program you created in Chapter 6 that stored the passwords in a dictionary. Updating the passwords required changing the source code of the program. This isn't ideal because average users don't feel comfortable changing source code to update their software. Also, every time you modify the source code to a program, you run the risk of accidentally introducing new bugs. By storing the data for a program in a different place than the code, you can make your programs easier for others to use and more resistant to bugs.