LAB 1:

Exercise 1-1 Experiment with IDLE's interactive shell

This exercise guides you through experimenting with Python's interactive shell.

1. Start IDLE. That should display a window for the interactive shell.
2. Enter the following print() functions and arithmetic expressions into the interactive console:

```
>>> print("The meaning of life")
The meaning of life
>>> print()

>>> 30 + 12
42
>>> 30-12
18
>>> 3 * 4
12
>>> 12/3
4.0
>>> 1 / 3
0.3333333333333333
>>> 3 * 4 + 30
42
>>> 3 * (4 + 30)
102
```

Note that (1) the print() function prints a blank line if you don't code anything between the parentheses, (2) the * is used for multiplication in an arithmetic expression, (3) the spaces before and after arithmetic operators aren't required, and (4) parentheses are used just as they are in algebraic expressions. If you make errors, the shell should display appropriate error messages.

3. Try entering these statements to see how they work:

var1 = 30
var2 = 25
var1 + 10
var1
var1 + var2
var_2

The last one is deliberately incorrect, so it should display an error message.

4. Continue experimenting if you like. However, this should make more sense after you read the next chapter and learn how to start writing Python code.

Exercise 1-2 Use IDLE to run programs

This exercise guides you through the process of using IDLE to compile and run three programs.

Run the console version of the Future Value program

1. Start IDLE and open the future_value.py file that should be in this folder:

   murach/python/book_apps/ch01

2. Press F5 to compile and run the program. Then, enter values for monthly investment, yearly interest rate, and years when you're prompted for them. This should display the future value that's calculated from your entries.

3. When you're asked if you want to continue, enter "y" if you want to do another calculation or "n" to exit the program. Then, close the IDLE window for the source code.

Run the GUI version of the Future Value program

4. Open the future_value_gui.py file that should be in the same folder as the **future_value.py file.**

5. Select Run�Run Module to compile and run the program. Then, enter values for the first three text boxes, and click the Calculate button to view the future value that's calculated from your entries.

6. If you want to do another calculation, enter new values and click the Calculate button. When you're through, click the Exit button to exit the program. Then, close the IDLE window for the source code.

Run the Guess the Number program of chapter 4

7. Open the guess_the_number.py file that's in this folder:

   murach/python/book_apps/ch04

8. Compile and run the program. Then, guess the number. This demonstrates the type of program that you'll be able to write by the time you complete chapter 4.

9. Close the window for the source code.

Exercise 1-3 Use IDLE to test and debug a program

This exercise guides you through the process of using IDLE to test and debug a program that has one syntax error and one runtime error. This walks you through the procedure that's shown in figure 1-9.

1. Start IDLE and open the **future_value_errors.py** file that should be in this folder:

python/exercises/ch01

2. Press F5 to compile and run the program. This should display a dialog box for a syntax error, and it should highlight the variable named "years" in the statement that gets the number of years.

3. Fix this syntax error by adding a right parenthesis at the end of the statement in the preceding line.

4. Press F5 again, and press Enter to save the updated file when the dialog box is displayed. This time, the program should compile cleanly and run in the interactive shell.

5. Enter the values for monthly investment, yearly interest rate, and years. This should cause a NameError exception that says "year" is not defined in this statement:

```
months = year * 12
```

6. Fix this runtime error by changing "year" to "years".

7. Compile and run this program again. This time, the program should be able to calculate and display a future value for the three values that you enter.

8. To exit this program, enter "n" at the last prompt. Otherwise, the program will continue running.

LAB 2:

Exercise 2-1 Modify the Miles Per Gallon program

In this exercise, you'll test and modify the code for the Miles Per Gallon program in figure 2-15. When you're finished, the program will get another user entry and do two more calculations, so the console will look something like this:

```
Enter miles driven:          150
Enter gallons of gas used:   15
Enter cost per gallon:       3

Miles Per Gallon:            10.0
Total Gas Cost:              45.0
Cost Per Mile:               0.3
```

If you have any problems when you test your changes, please refer to figure 1-9 of the last chapter, which shows how to fix syntax and runtime errors.

1. Start IDLE and open the mpg.py file that should be in this folder:
   python/exercises/ch02

2. Press F5 to compile and run the program. Then, enter valid values for miles driven and gallons used. This should display the miles per gallon in the interactive shell.

3. Test the program with invalid entries like spaces or letters. This should cause the program to crash and display error messages for the exceptions that occur.

4. Modify this program so the result is rounded to just one decimal place. Then, test this change.

5. Modify this program so the argument of the round() function is the arithmetic expression in the previous statement. Then, test this change.

6. Modify this program so it gets the cost of a gallon of gas as an entry from the user. Then, calculate the total gas cost and the cost per mile, and display the results on the console as shown above.

Exercise 2-2 Modify the Test Scores program

In this exercise, you'll modify the Test Scores program in figure 2-16. When you're finished, the program will display the three scores entered by the user in a single line, as shown in this console:

```
Enter 3 test scores
======================
Enter test score:        75
Enter test score:       85
Enter test score:       95
======================
Your Scores:            75 85 95
Total Score:            255
Average Score:          85
```

If you have any problems when you test your changes, please refer to figure 1-9 of the last chapter, which shows how to fix syntax and runtime errors.

1. Start IDLE and open the test_scores.py file that should be in this folder:

   python/exercises/ch02

2. Press F5 to compile and run the program. Then, enter valid values for the three scores. This should display the results in the interactive shell.

3. Modify this program so it saves the three scores that are entered in variables named score1, score2, and score3. Then, add these scores to the total_score variable, instead of adding the entries to the total_score variable without ever saving them.

4. Display the scores that have been entered before the other results, as shown above.

Exercise 2-3 Create a simple program

Copying and modifying an existing program is often a good way to start a new program. So in this exercise, you'll copy and modify the Miles Per Gallon program so it gets the length and width of a rectangle from the user, calculates the area and the perimeter of the rectangle, and displays the results in the console like this:

```
The Area and Perimeter program
Please enter the length: 25
Please enter the width: 10

Area = 250
Perimeter = 70

Bye!
```

1. Start IDLE and open the mpg_model.py file that is in this folder:

    python/exercises/ch02

    Then, before you do anything else use the File->Save As command to save the file as **rectangle.py**.

2. Modify the code for this program so it works for the new program. Remember that the area of a rectangle is just length times width, and the perimeter is 2 times length plus 2 times width.

LAB 3

Exercise 3-1 Enhance the Miles Per Gallon program

In this exercise, you'll enhance the Miles Per Gallon program in figure 3-8 so the console display looks something like this:

```
The Miles Per Gallon program
Enter miles driven:          150
Enter gallons of gas used:   15.2
Enter cost per gallon:       4.25

Miles Per Gallon:            9.87
Total Gas Cost:              64.6
Cost Per Mile:               0.4

Get entries for another trip (y/n)? y
Enter miles driven:          225
Enter gallons of gas used:   16
Enter cost per gallon:       4.25
Miles Per Gallon:            14.06
Total Gas Cost:              68.0
Cost Per Mile:               0.3


Get entries for another trip (y/n)?
```

1. Start IDLE and open the mpg.py file that's in this folder:

   murach/python/exercises/ch03

2. Test the program with valid and invalid values.

3. Enhance the program so it lets the user repeat the entries and get the miles per gallon for more than one trip. To do that, use a while loop.

4. Modify this program so it gets the cost of a gallon of gas as another entry from the user, and validate this entry before using it in your calculations. If all three entries are valid, calculate the total gas cost and the cost per mile, and display the results on the console.

Exercise 3-2 Enhance the Test Scores program

In this exercise, you'll enhance the Test Scores program in figure 3-16 so the console display looks something like this:

```
The Test Scores program

Enter test scores
Enter 'end' to end input
======================
Enter test score: 75
Enter test score: 85
Enter test score: 95
Enter test score: end
======================
Total Score: 255
Average Score: 85

Enter another set of test scores (y/n)? y

Enter test scores
Enter 'end' to end input
======================
Enter test score: 95
Enter test score: -85
Test score must be from 0 through 100. Try again.
Enter test score: 85
Enter test score: 60
Enter test score: end
======================
Total Score: 240
Average Score: 80

Enter another set of test scores (y/n)?
```

1. Start IDLE and open the test_scores.py file that's in this folder:
murach/python/exercises/ch03

2. Test the program with valid and invalid values.

3. Enhance the program so it lets the user enter two or more sets of scores. Use a while loop to do that. That nests one while loop within another.

4. Enhance this program so the user enters “end” to end a set of score entries, but keep the validation of the test scores. To do this, you need to change the if statement within the inner while loop. In fact, you may want to nest one if statement within the else clause of another one to get the results that you want.

5. Copy the while statement, paste it below the existing while loop, and then comment out the original while statement. Now, modify the while statement so it uses an assignment expression to get the lowercase value the user enters and tests that it’s not equal to “end”. Modify the code within this loop so it works with the assignment expression.

Exercise 3-3 Enhance the Future Value program

In this exercise, you'll enhance the Future Value program so the console display looks something like this:

```
Welcome to the Future Value Calculator

Enter monthly investment: 0
Entry must be greater than 0. Please try again.

Enter monthly investment: 100

Enter yearly interest rate: 16


Entry must be greater than 0 and less than or equal to 15.

Please try again.
Enter yearly interest rate: 12

Enter number of years: 100


Entry must be greater than 0 and less than or equal to 50.

Please try again.

Enter number of years: 10


Year = 1                    Future Value = 1280.93

Year = 2                    Future Value = 2724.32

Year = 3                    Future Value = 4350.76

Year = 4                    Future Value = 6183.48

Year = 5                    Future Value = 8248.64

Year = 6                    Future Value = 10575.7

Year = 7                    Future Value = 13197.9

Year = 8                    Future Value = 16152.66

Year = 9                    Future Value = 19482.15

Year = 10                    Future Value = 23233.91


Continue (y/n)? n
```

1. In IDLE, open the future_value.py file that’s in this folder:

murach/python/exercises/ch03

2. Test the program, but remember that it doesn’t do any validation so enter valid numbers.

3. Add data validation for the monthly investment entry. Use a while loop to check the validity of the entry and keep looping until the entry is valid. To be valid, the investment amount must be greater than zero. If it isn’t, an error message like the first one shown above should be displayed.

4. Use the same technique to add data validation for the interest rate and years. The interest rate must be greater than zero and less than or equal to 15. And the years must be greater than zero and less than or equal to 50. For each invalid entry, display an appropriate error message. When you’re finished, you’ll have three while loops and a for loop nested within an outer while loop.

5. Modify the statements in the for loop that calculate the future value so one line is displayed for each year that shows the year number and future value, as shown above. To do that, you need to work with the integer that’s returned by the range() function.

LAB 4

Exercise 4-1 Enhance the Future Value program

In this exercise, you'll enhance the Future Value program in figure 3-17 so it validates the three user entries with messages that are something like these:

```
Welcome to the Future Value Calculator
Enter monthly investment:                    0
Entry must be greater than 0 and less than or equal to 1000
Enter monthly investment:                    100
Enter yearly interest rate:                  16

Entry must be greater than 0 and less than or equal to 15
Enter yearly interest rate:                  12
Enter number of years:                       100

Entry must be greater than 0 and less than or equal to 50
Enter number of years:                       10
Future Value =                               23233.91
Continue (y/n)? n
```

1. In IDLE, open the future_value.py file that's in this folder:

python/exercises/ch04

2. Test the program, but remember that it doesn't do any validation, so enter valid numbers.

Add two validation functions to the program

3. In preparation for adding two functions named get_float() and get_int() to the program, create a hierarchy chart or outline that includes those functions. The functions will be used to get valid numbers and integers from the user as described in the next steps.

4. Add a function named get_float() to the program. This function should accept one argument, which is a prompt like "Enter monthly investment: ". Then, this function should use the input function to get an entry from the user using the prompt that's passed to it, and the entry should be converted to a float value. Next, this function should check this entry to make sure it's greater than 0. If it is, the entry is valid and the number should be returned to the calling statement. If it isn't, an appropriate error message should be displayed, and the user should enter another value. To make this work, the function should use a while statement that gets an entry until it is valid.

5. Modify the main() function so it uses this function to get the monthly investment entry. That tests the function.

6. Enhance the get_float() function so it gets three arguments: a prompt, a low validity value, and a high validity value. This function should work as before, except the entry must be greater than the low value and less than or equal to the high value. If the entry is invalid, this function should display error messages like those above using the low and high arguments. Otherwise, it should return the value to the calling statement.

7. Modify the main() function so it uses the get_float() function for the first two entries. The low and high arguments should be 0 and 1000 for the first entry, and 0 and 15 for the second entry.

8. Add a get_int() function that works like the get_float() function, except that it gets an integer entry instead of a float entry. The low and high arguments should be 0 and 50. Then, call this function from the main() function so it gets the years entry.

Create a validation module

9. Use the File�Save As command to save a copy of the Future Value program as validation.py. This will be the file for a module that stores the get_float() and get_int() functions. So, delete the calculate_future_value() function, but keep the main() function and the if statement after the main() function.

10. In the main() function, delete everything inside the while loop except the last statement that asks whether the user wants to continue. Within this loop, code two call statements that test the get_float() and get_int() functions.

11. Run the program to test the two functions. When you're through, you can close the file.

Use the validation module in the Future Value program

12. Go back to the future_value.py file. Then, comment out the get_number and get_integer() functions.

13. Add an import statement that imports the validation module.
14. Modify the code in the main() function so it uses the functions in the validation module.

LAB 5

Exercise 5-1 Test and debug a Test Scores program

In this exercise, you will test and debug a variation of the Test Scores program of chapter 3.

1. In IDLE, open the test_scores.py file that's in this folder:

python/exercises/ch05

Then, review the code.

2. Create a test plan that thoroughly tests the program with valid data. This can be a handwritten table or a spreadsheet that includes the test data for three or four test runs as well as the expected results.

3. Use your test plan as a guide to testing the program. Then, note any inaccurate results that you discover during testing.

4. Debug any logic errors. 5. Test the program with the same data to be sure it works correctly.

Exercise 5-2 Trace and test the functions of the Future Value program

In this exercise, you will trace the operation of the calculate_future_value() function of a Future Value program. You'll also use the IDLE shell to test the functions in this program.

Use print() functions to trace the execution of a function

1. In IDLE, open the future_value.py file that's in this folder:

python/exercises/ch05

Then, test the program and note that the future value results are obviously inaccurate. In fact, the calculate_future_value() function has two logic errors.

2. To debug this problem, scroll down to the calculate_future_value() function, and add print() functions that show you how the values of the variables change each time through the for loop.

3. Run the program and review the results.

4. Fix the errors and comment out the print() functions that you added.

Use the IDLE shell to test the functions of this program

5. Run the Future Value program to make sure its functions are loaded into the shell. Then, cancel the execution of the program.

6. Test its three functions as shown in figure 5-6. This shows you how easy it is to test a function without running the entire program.

Exercise 5-3 Step through a Future Value program

This exercise guides you through the use of the IDLE debugger as you step through the Future Value program.

Step through the calculate_future_value() function

1. In IDLE, open the future_value.py file that's in this folder:

python/exercises/ch05

If you did exercise 5-2, this will be a corrected version of this program. Otherwise, it will still contain two logic errors that lead to inaccurate results.

2. In the calculate_future_value() function, set a breakpoint on the first executable statement. Then, go the IDLE shell and start the debugger. That should display the Debug Control window. Now, check the Source checkbox.

3. Try to arrange the editor, shell, and Debug Control windows so you can see all three. That should make it easier for you to switch from one window to another and to see what's happening.

4. Go to the editor and run the program. That should take you to the Debug Control window. Then, click Go to run the program to the breakpoint, and enter the required values when prompted in the interactive shell.

5. At the breakpoint, click the Step button to step through the function and note how the values of the local variables change. When you see how this works, click the Quit button to end the debugging session.

Step through the entire program

6. In the editor window, remove the breakpoint in the calculate_future_value() function, and set a breakpoint on the first statement of the main module. Then, run the program to the breakpoint as before.

7. Step through the program. This should take you from the main() function to the functions that it calls. This may also take you to Python functions that operate behind the scenes. Since these functions probably won't make much sense to you, you can step out of them right away. In short, experiment with the Step, Over, and Out buttons as you step through the code.

8. When you're through experimenting, click on the Quit button to end the session.

LAB 6

Exercise 6-1 Use a list for the Test Scores program

In this exercise, you'll modify a Test Scores program that gets the test scores that a user enters and then calculates and displays the average test score. You'll enhance this program by storing the test scores in a list and then getting and displaying other statistics for the test scores, like this:

```
The Test Scores program
Enter 'x' to exit
Enter test score:          75
Enter test score:          85
Enter test score:          95
Enter test score:          x

Total:                     255
Number of Scores:          3
Average Score:             85
Low Score:                 75
High Score:                95
Median Score               85
```

1. In IDLE, open the test_scores.py file that's in this folder:

    python/exercises/ch06

2. Review the code, and test the program.

3. Modify the get_scores() function so the test scores are stored in a list named scores. This list should be returned by the function when all scores have been entered. The function should still make sure that the entries are valid, but the score_total and count variables aren't needed and shouldn't be updated.

4. Modify the process_scores() function so the scores list is its only argument. Then, this function should use a for statement to total the scores in the list. It should use the len() function to get the number of scores in the list. And it should get the average by dividing the total scores by the length.
5. Modify the main() function so the list that's returned by the get_scores() function is stored in a variable. Then, modify the call to the process_scores() function so it passes just the scores list to it.

6. Test this program to make sure everything is working right.

7. Enhance this program by getting and displaying all of the other statistics shown above. For an odd number of scores, the median score is the score that has the same number of scores below it as above it. For an even number of scores, calculate the median by taking the average of the two middle numbers.

Exercise 6-2 Enhance the Movie List 2D program

In this exercise, you'll modify the Movie List 2D program in figure 6-8 so it provides a third column for each movie. Otherwise, this program should work the same way it did before:

```
COMMAND MENU
list - List all movies
add - Add a movie
del - Delete a movie
find - Find movies by year
exit - Exit program

Command: add
Name: Gone with the Wind
Year: 1939
Price: 14.95
Gone with the Wind was added.

Command: list
1. Monty Python and the Holy Grail (1975) @ 9.95
2. On the Waterfront (1954) @ 5.59
3. Cat on a Hot Tin Roof (1958) @ 7.95
4. Gone with the Wind (1939) @ 14.95

Command: find
Year: 1954
On the Waterfront was released in 1954

Command:
```

1. In IDLE, open the movie_list_2d.py file that's in this folder:

    python/exercises/ch06

2. Enhance the program so it provides for the price column that's shown above.

3. Enhance the program so it provides a find by year function that lists all of the movies that were released in the year that the user requests, as shown above.

LAB 7

Exercise 7-1 Create a CSV file for trip data

In this exercise, you'll modify the Miles Per Gallon program so it stores the data for each calculation in a CSV file. Then, the program that you develop for exercise 7-2 should be able to display the data like this:

```
Distance          Gallons          MPG
225               17               13.24
1374              64               21.47
2514              79               31.82
```

1. In IDLE, open the mpg_write.py file that's in this folder:

    python/exercises/ch07

2. Review the code, and run the program so you remember how it works.

3. Enhance the program so it stores the data for each calculation, or trip, in a two-dimensional list. For each calculation, these values should be put in the list: miles driven, gallons of gas used, and the calculated MPG value.

4. Enhance the program so it saves the data in the list to a file named trips.csv when the user wants to exit from the program.

5. Test the program to make sure it works. To do that, you can open the CSV file with a spreadsheet program like Excel.

Exercise 7-2 Keep trip data in a CSV file

In this exercise, you'll modify the Miles Per Gallon program so it adds to the data in the file that you created in exercise 7-1. This program should display the data for each trip that's entered in a CSV file as shown here:

```
The Miles Per Gallon program
Distance            Gallons             MPG
225                 17                  13.24
1374                64                  21.47
2514                79                  31.82

Enter miles driven:        274
Enter gallons of gas:      18.5
Miles Per Gallon:          14.81

Distance            Gallons             MPG
225                 17                  13.24
1374                64                  21.47
2514                79                  31.82
274.0               18.5                14.81

More entries? (y or n):
```

1. In IDLE, open the mpg.py file that's in this folder:

    python/exercises/ch07

2. Add a write_trips() function that writes the data from a two-dimensional list named trips that's passed to it as an argument. This list contains the data for each trip that's entered, and it should be written to a CSV file named trips.csv. As the console above shows, the data for each trip consists of miles driven, gallons of gas used, and the calculated MPG value.
3. Add a read_trips() function that reads the data from the trips.csv file and returns the data for the trips in a two-dimensional list named trips.
4. Add a list_trips() function that displays the data in the trips list on the console, as shown above.
5. Enhance the main() function so it starts by getting the data from the CSV file and listing it as shown above.
6. Enhance the main() function so it adds the last trip that's entered to the trips list after it calculates the MPG. Then, display the data for the updated trips list.
7. Test all aspects of the program until you're sure that it works correctly.

Exercise 7-3 Keep trip data in a binary file

In this exercise, you’ll modify the programs that you created for exercises 7-1 and 7-2 so they create and use a binary file instead of a CSV file. Otherwise, everything should work the same.

Modify the CSV version of the write program

1. Open the mpg_write.py file that you created in exercise 7-l. Then, save it as mpg_write_binary.py in the same directory.
2. Modify this program so it saves the list as a binary file instead of a CSV file. The file should be named trips.bin.
3. Test the program to make sure it works. To do that, add statements that read the file at the end of the program and display the list that has been read.

Modify the CSV version of the trip program

4. Open the mpg.py file that you created in exercise 7-2. Then, save it as **mpg_binary.py**.
5. Modify this program so it works the same as it did with the CSV file.
6. Test this program to make sure it works.

LAB 8

Exercise 8-1 Add exception handling to the Future Value program

In this exercise, you'll modify the Future Value program so the user can't cause the program to crash by entering an invalid int or float value.

1. In IDLE, open the future_value.py file that's in this folder:

    python/exercises/ch08

2. Review the code and study the get_number() and get_integer() functions. Note that they receive three arguments: the prompt for a user entry, the low value that the entry must be greater than, and the high value that the entry must be less than or equal to. Then, review the calling statements in the main() function and note how these functions are used.
3. Test the program. Note that you can cause the program to crash by entering values that can't be converted to float and int values.
4. Add exception handling to the get_number() and get_integer() functions so the user has to enter valid float and int values. Then, test these changes to make sure the exception handling and the data validation work correctly.

Exercise 8-2 Enhance the Movie List 2.0 program

In this exercise, you'll modify the Movies List 2.0 program so it does more exception handling. You'll also use a raise statement to test for exceptions.

1. In IDLE, open the movies2.py file that's in this folder:

   python/exercises/ch08

2. Add data validation to the add_movie() function so the year entry is a valid integer that's greater than zero. Then, test this change.

3. Modify the write_movies() function so it also handles any OSError exceptions by displaying the class name and error message of the exception object and exiting the program.

4. Test this by using a raise statement in the try block that raises a BlockingIOError. This is one of the child classes of the OSError. Then, comment out the raise statement.

5. In the read_movies() function, comment out the two statements in the except clause for the FileNotFoundError. Instead, use this except clause to return the empty movies list that's initialized in the try block. This should cause the program to continue if the file can't be found by allowing the program to create a new file for the movies that the user adds.

6. Test this by first running the program with the movies.csv file. That should work as before. Then, change the filename in the global constant to movies_test.csv, run the program again, and add a movie. That should create a new file. If that works, run the program again to check whether it works with the new file.

LAB 9

Exercise 9-1 Enhance the Invoice program

In this exercise, you'll enhance a version of the Invoice program so two of the currency values are formatted correctly. You'll also add a shipping cost to the invoice:

```
Enter order total:            10554.23

Order total:                $10,554.23
Discount amount:               2110.85
Subtotal:                      8443.38
Shipping cost:                  717.69
Sales tax:                      422.17
Invoice total:               $9,583.24

Continue? (y/n):
```

Use the locale module for the currency values

1. In IDLE, open the invoice_decimal.py file that's in this folder:

    python/exercises/ch09

2. Modify this program so it displays the order total and invoice total as currency values in the United States. Note that because these values are now strings, they will need to be right aligned.

Add a shipping cost

3. Add a shipping cost as shown in the console data above. This charge should be .085 of the subtotal. As a result, it could cause a rounding error. To prevent this error from occurring, use the decimal module to make sure each monetary value has the correct number of decimal places.
4. Test and debug this program and make sure that it works with all input values.

Use variables for format specifications

5. Review the code that prints the order total and calculated values, and notice the format specifications used by these values. Create a variable for each different specification, and then use those variables to format the values. Test this to be sure it works correctly.
6. Create another variable with a format specification that can be used to set the width of the string literals that are displayed. Then, use this specification with the string literals. When you do that, you'll need to pay close attention to the use of quotes and braces. Test this to be sure it works correctly.

LAB 10

Exercise 10-1 Enhance the Create Account program

In this exercise, you'll enhance the Create Account program in figure 10-5 so it gets a valid email address and a valid phone number, as shown here:

```
Enter full name:                    Joel Murach
Enter password:                     Extra54321
Enter email address:                joelmurach.com
Please enter a valid email address.
Enter email address:                joel.murach@com
Please enter a valid email address.
Enter email address:                 joel@murach.com
Enter phone number:                 (555) 555-555
Please enter a 10-digit phone number.
Enter phone number:                 (555) 555-SPAM
Please enter a 10-digit phone number.
Enter phone number:                 (555) 555-5555
Hi Joel, thanks for creating an account.

We'll text your confirmation code to this number: 555.555.5555
```

1. In IDLE, open the create_account.py file that's in this folder:

   python/exercises/ch10

2. Create and use a function to get a valid email address. To be valid, the address has to contain an @ sign and end with ".com".

3. Create and use a function to get a valid phone number. To do that, remove all spaces, dashes, parens, and periods from the number. Then, check to make sure the number consists of 10 characters that are digits.

4. When all of the entries are valid, display the message shown above, including the phone number format that uses dots to group the digits.

Exercise 10-2 Enhance the Word Counter program

In this exercise, you'll enhance the Word Counter program in figure 10-9 so it also displays the number of sentences in the text string, as shown here:

```
The Word Counter program

Number of sentences = 11
Number of words = 260
Number of unique words = 142
Unique word occurrences:
        a = 7
        ...
```

1. In IDLE, open the word_counter.py file that's in this folder:

    python/exercises/ch10

2. Create and use a function to get the number of sentences. To keep this simple, this function can read the text file just as the function for getting the number of words does. Then, the main() function should display the number of sentences as shown above.

Exercise 10-3 Enhance the Hangman game

In this exercise, you'll enhance the Hangman game in figure 10-11 so it displays a graphic hangman as shown here:

```
Play the H A N G M A N game
____
   |
   O
  \|/
   |
  / \
----------------------------------------------------------------------
____
   |
Word: _ _ _ _  Guesses: 0    Wrong: 0     Tried:
Enter a letter: b
----------------------------------------------------------------------
____
   |
   O
Word: _ _ _ _  Guesses: 1    Wrong: 1     Tried: B
Enter a letter: f
----------------------------------------------------------------------
          (The game continues)
----------------------------------------------------------------------
____
   |
   O
  \|/
   |
  / \
Word: _ A _ E         Guesses: 9     Wrong: 7     Tried: B F A E I C D G H
----------------------------------------------------------------------
Sorry, you lost.
The word was: RAKE
```

1. In IDLE, open the hangman.py file that's in this folder:

    python/exercises/ch10

2. Enhance this program so it displays a hangman graphic when the program starts as shown above. This hangman consists of the letter O, bars, slashes, and backslashes. To display it, create a draw_hangman() function that is called from the main() function.

3. Enhance this program so it adds one character to the hangman each time the player guesses wrong, starting with O for the head. To do this, enhance the draw_hangman() function so it gets the number of wrong guesses as its only argument. Then, it should display that many hangman characters. This function should be called by the draw_screen() function as well as the main() function. In this version of the game, the player only gets 7 wrong guesses. So, make sure to modify the program accordingly.

LAB 11

Exercise 11-1 Modify the Invoice Due Date program

In this exercise, you'll modify the Invoice Due Date program so it uses date objects, not datetime objects. In addition, you'll add some code that makes sure the user enters a valid date.

```
The Invoice Due Date program

Enter the invoice date (MM/DD/YYYY):      1-14-2021
Invalid date format! Try again.
Enter the invoice date (MM/DD/YYYY):      1/14/2099
Date must be today or earlier. Try again.
Enter the invoice date (MM/DD/YYYY):      1/14/2021

Invoice Date:  January 14, 2021
Due Date:      February 13, 2021
Current Date:  March 15, 20121

This invoice is 30 day(s) overdue.

Continue? (y/n):
```

Review the code and make sure it works correctly

1. In IDLE, open the invoice.py file that's in this folder:
   python/exercises/ch11
2. Review the code and run the program to see how it works.

Use date objects to store the three dates

3. Modify the get_invoice_date() function so it uses a four-digit year.

4. Modify the program so it uses date objects, not datetime objects, to store the three dates. To do that, you can create a date object from the parts of the datetime object that's created in the get_invoice_date() function.

Add validation

5. Run the program, and enter an invalid date. This should cause the program to crash. Then, add code to the get_invoice_date() function that prevents the program from crashing and makes sure the user enters a valid date.
6. Add code to the get_invoice_date() function that makes sure the user enters a date that's either today or in the past.

Exercise 11-2 Modify the Timer program

In this exercise, you'll modify the Timer program so it displays the time like this:

```
The Timer program

Press Enter to start...
Start time:            15:09:53.206009

Press Enter to stop...
Stop time:             15:12:24.982324

ELAPSED TIME
Hours/minutes: 00:02
Seconds: 31.776315
```

This makes it easier to read the timer for elapsed times that are less than one day.

Review the code and test it

1. In IDLE, open the timer.py file that's in this folder:

    python/exercises/ch11

2. Review the code and run the program to see how it works.

Modify the way the elapsed time is displayed

3. Modify the code that displays the start and stop times so it only displays the time part, not the date part.

4. Modify the code that displays the elapsed time so it only shows hours/minutes and seconds. Also, the program should only display the hours/minutes data if it contains a non-zero value.

LAB 12

Exercise 12-1 Add a list method to the Book Catalog program

In this exercise, you'll enhance the Book Catalog program so it offers a list command that will list all of the books in the catalog, as shown here:

```
COMMAND MENU
list – List all books
show - Show book info
add - Add book
edit - Edit book
del - Delete book
exit - Exit program


Command: list
Title:          Slaughterhouse Five
Author:         Kurt Vonnegut
Pub year:       1969

Title:          The Hobbit
Author:         J. R. R. Tolkien
Pub year:       1937


Title:          Moby Dick
Author:         Herman Melville
Pub year: 1851


 Command:
```

1. In IDLE, open the book_catalog.py file that's in this folder:

   python/exercises/ch12

2. Review the code and run it.
3. Add the list command.

Exercise 12-2 Enhance the Movie List 2D program

In this exercise, you'll enhance the Movie List program of chapter 6 so it uses a list of dictionaries instead of a list of lists to store the data for the movies. This is similar to how the movie data would be stored if you were retrieving it from a database as shown in chapter 17.

Open and test the program

1. In IDLE, open the movie_list_2d.py file that's in this folder:

    python/exercises/ch12

2. Review the code and run it to refresh your memory about how it works.

Modify the code so it uses a dictionary

3. In the main() function, modify the code so it stores a list of dictionaries where each dictionary stores the data for a movie. Use "name" as the key for the movie's name and "year" as the key for the movie's year.

4. In the list() function, modify the code in the loop that displays each movie so it uses the correct key to get the name and year of the movie. Note that this makes the code easier to read and understand.

5. In the add() function, modify the code that stores the data for the movie so it uses a dictionary instead of a list. Note that you should still use a list to store the list of movies, but you should use a dictionary to store the data for each movie.

6. In the delete() function, modify the code so it works correctly now that the data for the movie is stored in a dictionary.

7. Run the program. It should work as it did before. However, it's now using a dictionary to store the data for each movie.

LAB 13

Exercise 13-1 Test the Fibonacci program

In this exercise, you'll test the recursive function that prints part of the Fibonacci series. Then, you'll test an iterative function that does the same thing, but more efficiently.

Open and test the recursive Fibonacci program

1. In IDLE, open the fibonacci_recursion.py file that's in this folder:

    python/exercises/ch13

2. Review the code and run it to make sure it works correctly. It should print the first 16 numbers of the Fibonacci series.

3. Modify the code so it calculates a Fibonacci series for 32 numbers, and run it again. Note how slowly it runs.

View the stack for the program

4. Modify the code so it calculates a Fibonacci series for 100 numbers, and run it again. While it's running press Ctrl+C on a Windows system or Command+C on a Mac to interrupt the calculation and view the stack trace. Note that the fib() function has been called numerous times.

5. From the IDLE shell, select Debug⍰Stack Viewer. In the Stack Viewer, expand a few of the fib() functions and note that they include the values for the local variable named n.

Open and test the iterative Fibonacci program

6. Open the fibonacci_loop.py file that's in the same folder.

7. Review the code and run it to make sure it works correctly.

8. Modify the code so it calculates a Fibonacci series for 100 numbers, and run it. Note how quickly it runs compared to the recursive program.

Exercise 13-2 Test the Towers of Hanoi puzzle

In this exercise, you'll test the Towers of Hanoi puzzle.

Open and test the program

1. In IDLE, open the towers_of_hanoi.py file that's in this folder:

    python/exercises/ch13

2. Review the code and run it for 3, 4, and 5 disks. Note that the pattern is slightly different depending on whether the number is even or odd.

3. Continue testing with larger numbers until the program begins running too slowly to be acceptable.

Add a print statement that traces the calls on the stack

4. Add a statement to the start of the move_disk() function that prints the name of the function and the values of the arguments that are being passed to it. When you're done, the console should look like this for 4 disks:

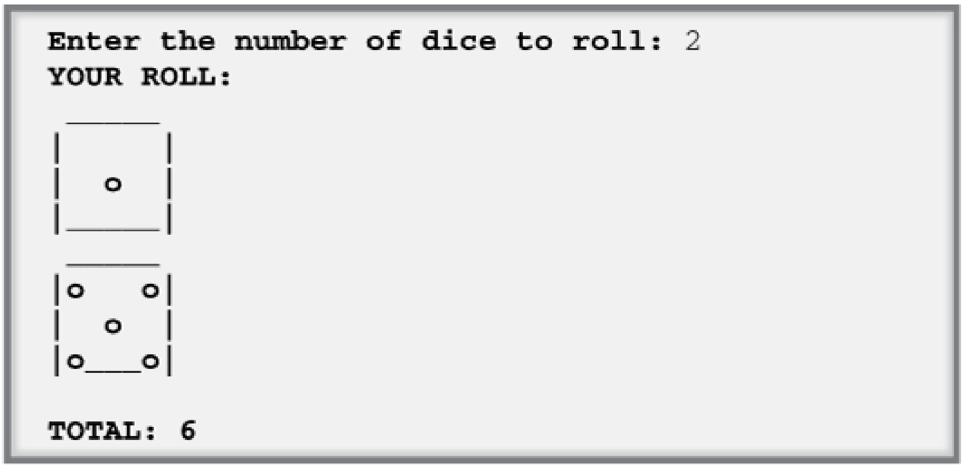
```
**** TOWERS OF HANOI ****
Enter number of disks: 4
        move_disk(n=4, src=A, dest=C, temp=B)
        move_disk(n=3, src=A, dest=B, temp=C)
        move_disk(n=2, src=A, dest=C, temp=B)
        move_disk(n=1, src=A, dest=B, temp=C)
        move_disk(n=0, src=A, dest=C, temp=B)
Move disk 1 from A to B
        move_disk(n=0, src=C, dest=B, temp=A)
Move disk 2 from A to C
        move_disk(n=1, src=B, dest=C, temp=A)
        move_disk(n=0, src=B, dest=A, temp=C)
Move disk 1 from B to C
        move_disk(n=0, src=A, dest=C, temp=B)
Move disk 3 from A to B
        move_disk(n=2, src=C, dest=B, temp=A)
        move_disk(n=1, src=C, dest=A, temp=B)
        move_disk(n=0, src=C, dest=B, temp=A)
...
```

5. Study the console and note how the move_disk() functions are pushed onto the stack and popped off the stack.

LAB 14

Exercise 14-1 Enhance the Dice Roller program

In this exercise, you'll enhance the Dice Roller program by making some improvements to its classes. When you're done, rolling two dice should look something like this:



```
Enter the number of dice to roll: 2
YOUR ROLL:
 _____
|     |
|  o  |
|_____|
 _____
|o   o|
|  o  |
|o___o|

TOTAL: 6
```

Open and test the program

1. In IDLE, open the dice.py and dice_roller.py files that are in this folder:

python/exercises/ch14/dice_roller

2. Review the code and run the program to make sure it works correctly. Note that it starts by displaying an image for each of the 6 possible die values.

Improve the Die class

3. In the Die class, modify the roll() method so it returns the __value attribute after it sets it to a random number from 1 to 6.

4. In the Die class, add a __post_int__() method that sets the __value attribute by calling the roll() method. This makes sure that the __value attribute for a new Die object stores a valid number for the die.

5. In the Die class, modify the setter for the value property so it doesn't allow a value greater than 6.

6. Run the dice module and use Python's interactive shell to make sure these changes work correctly.

7. In the Die class, add a read-only property named image that gets a string for the image that represents the die's current value.

Improve the Dice Roller program

8. Open the dice_roller.py file and run it to make sure the Dice Roller program still works correctly. Since you didn't change the interface for the Die class, it should.

9. Modify the code that displays the roll so it uses the new image property to display an image for each die instead of displaying the value.

10. At the start of the program, modify the code that displays the 6 die images so it uses a loop to create a Die object for each valid number and to display its image. This reduces code duplication since the code that defines the image is only stored in one place now, in the Die class.

Improve the Dice class

11. In the Dice class, add a method named getTotal() that gets the total value of all Die objects currently stored in the Dice object.

12. In the dice_roller.py file, add the code that displays the total each time the user rolls the dice.

Exercise 14-2 Create an object-oriented Movie List program

In this exercise, you'll convert the Movie List program presented in chapter 6 so it uses objects instead of storing the data for the movie in a list.

Open and test the program

1. In IDLE, open the movie.py file that's in this folder:

    python/exercises/ch14/movies

2. Review the code and note how it uses a list to store the data for each movie.

3. Run the code to make sure it works correctly.

Define a Movie object that can store the data for each movie

4. Add a module named objects to the program's folder.

5. In the object module, write the code for a class named Movie that defines a Movie object that stores the name and year of a movie. This class should include a getStr() method that returns the name of the movie followed by its year in parentheses.

6. Run the module. This should display the interactive shell. Then, use the shell to test the class by creating a Movie object and printing it to the console.

Modify the program so it uses the Movie object

7. Switch back to the movie.py file, and add a statement that imports the Movie class.
8. Modify the main() function so it creates a list of Movie objects instead of a list of lists.
9. Modify the list() function so it uses the getStr() method to display each movie. Note how this simplifies the code.
10. Modify the add() function so it creates a Movie object from the user input. Note how this simplifies the code.
11. Modify the delete() function so it uses the Movie object to display the name of the movie that's deleted. Note how this makes the code easier to read and understand.

Exercise 14-3 Create an object-oriented Temperature Converter

In this exercise, you'll modify the Temperature Converter program presented in chapter 4 so it uses object-oriented programming instead of procedural programming.

Open and test the program

1. In IDLE, open the two files stored in this folder:

   python/exercises/ch14/temperature

2. Review the code for the program and note how it uses the functions in the temperature module to convert the temperatures.

3. Run the code to make sure it works correctly.

Define a Temp object that can store a temperature

4. Switch to the temperature.py file. At the top of this module, define a class named Temp that defines two hidden attributes to store the degrees Fahrenheit and Celsius.

5. In the Temp class, define the methods that set the hidden attributes. When you set one unit of temperature, it should also calculate and set the other unit of temperature. For example, when you set degrees Fahrenheit, it should also calculate and set degrees Celsius.

6. In the Temp class, define the methods that get the hidden attributes. These methods should round the number that it returns to 2 decimal places.

7. Delete the to_fahrenheit() and to_celsius() functions.

8. In the temperature module, modify the code in the main() function so it tests the Temp class.

9. Run the temperature module. It should successfully convert the temperatures specified by the loops.

Modify the program so it uses the Temp object

10. Switch back to the convert_temperatures.py file, and modify the import statement so it imports the Temp class from the temperature module.

11. In the convert_temp() function, create a Temp object and use it to set and get the temperatures.

LAB 15

Exercise 15-1 Enhance the Product Viewer program

In this exercise, you'll enhance the Product Viewer program shown in this chapter so it provides for one more type of product: a music album. When you enter the product number for a music album, it should print the data to the console like this:

```
Enter product number: 4


PRODUCT DATA
Name:                 Rubber Soul
Artist:               The Beatles
Format:                CD
Discount price:       10.00
```

Open and test the program

1. In IDLE, open the objects.py and product_viewer.py files in this folder:

    python/exercises/ch15/product_viewer

2. Review the code and run the program to make sure it works correctly. Note that the Movie class displays the format, which is DVD, as part of the name of the product.

Improve the Movie and Book classes

3. In the Movie class, add an attribute named format that stores the format of the product. For example, the format could be DVD, streaming, and so on.

4. In the product_viewer module, modify the code that creates the Movie object so it stores "DVD" as the format attribute instead of appending this data to the end of the name attribute. Then, modify the code that displays the Movie object so it displays the format on a separate line, after the year of the movie.

5. Repeat steps 3 and 4 for the Book class. You can use "Hardcover" as the format for the book. Or, if you prefer, you can specify a different type of book format such as "Paperback" or "ebook".

Add an Album class

6. In the objects module, add a class named Album that inherits the Product class. The Album class should add two attributes: one for storing the artist and another for storing the format.

7. In the product_viewer.py file, modify the code that creates the objects so it includes a fourth object, an Album object. This object should contain the data for a music album that you like. Then, add code that displays the Album object as shown at the beginning of this exercise.

Add a Media class

8. In the objects module, add a class named Media that inherits the Product class. This class should add a format attribute to the Product class.

9. Modify the Movie, Book, and Album classes so they inherit the Media class, not the Product class. This should create a class hierarchy that looks like this:

```
Product
    Media
        Movie
        Book
        Album
```

10. In the product_viewer.py file, modify the code that displays a product so it only displays the format attribute for Media objects. Note how this reduces code duplication.

Exercise 15-2 Work with Author and Authors objects

In this exercise, you'll work with a Book object that uses an Authors object to store one or more Author objects.

Open and test the program

1. In IDLE, open the objects.py and authors_tester.py files that are in this folder:

    python/exercises/ch15/authors

2. Review the code and note how the Book object uses an Authors object to store one or more Authors.

3. Run the code to see how it works. At this point, it doesn't display the book or author information correctly, but you'll fix that later in this exercise.

Improve the Author, Authors, and Book classes

4. In the Author class, add a __str__() method that returns the first and last name of the author, separated by a space.

5. In the Authors class, add a __str__() method that returns the name of each author, separating multiple authors with a comma and a space.

6. In the Book class, add a __str__() method that returns the title of the book, followed by the word "by" and one or more authors, with each author separated by a comma.

7. Run the authors_tester module again. This time, it should display the correct data for the book and author information.

Make sure the program works correctly for a single author

8. In the authors_tester module, comment out the statement that adds the second author. Then, run the module again. This should work, but the last line says "Authors" where it should say "Author".

9. Modify the code so it uses the count property of the Authors object to display the correct label for the author or authors depending on the number of authors for the book.

Define and use an iterator for the Authors object

10. In the Authors class, add the __iter__() method that makes it possible to use a for statement to loop through all Author objects stored in the Authors object.

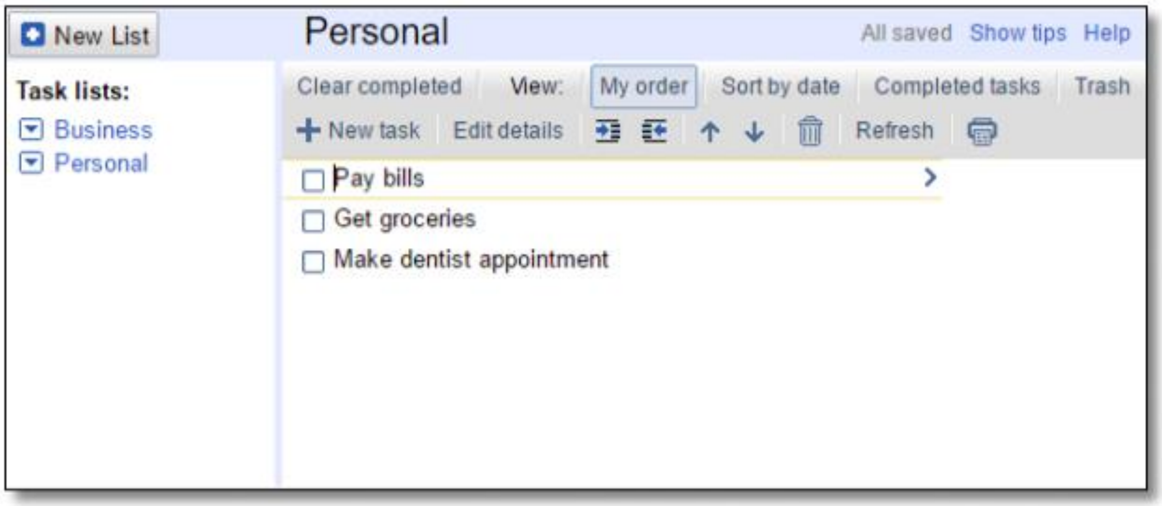
11. In the authors_tester module, add a for statement that loops through each author in the Authors object and prints each author to the console. When you're done, running this module should display this data to the console:

```
The Authors Tester program

BOOK DATA - SINGLE LINE
The Gilded Age by Mark Twain, Charles Warner

BOOK DATA - MUTLIPLE LINES
Title:          The Gilded Age
Authors:        Mark Twain, Charles Warner

AUTHORS
Mark Twain
Charles Warner
```

LAB 16

Exercise 16-1 Design a Task List program and implement your design

In this exercise, you'll design the business objects for a program that stores one or more lists of tasks. It should be a simplified version of the following screen capture:



Note that this program shows two task lists (Business and Personal), but the user could add more lists, and each list can contain zero or more tasks. For the first three steps, you can use a word processor or text editor. The only deliverable will be the UML diagram that you prepare in steps 4 and 5.

Design the business objects for the program

1. Identify the data attributes.

2. Subdivide each attribute into its smallest useful components.

3. Identify the classes by sorting the data attributes into categories.

4. Identify the methods and properties by drawing a UML diagram for the class. To do that, you can use a software tool (many are available for free), or you can sketch the diagram on a piece of paper.

5. Refine the classes, attributes, methods, and properties. When you're done, you should be ready to start coding and testing the business objects for the program.

Implement the business objects and test them

6. Write the code for the business objects.

7. Test the business objects. To do that, you can use the interactive shell, or you can write code that tests these objects.

8. As you work on the previous two steps, you can further refine the classes, attributes, methods, and properties for the business objects.

Implement the user interface and test it

9. Write the code that implements the user interface. When you're done, it should look something like this:

```
COMMAND MENU
List        - List all tasks
add         - Add a task
complete    - Complete a task
Delete      - Delete a task
switch      - Switch selected task list
exit        - Exit program

TASK LISTS
1. Personal
2. Business

Enter number to select task list: 1
Personal task list was selected.

Command: add
Description: Pay bills

Command: add
Description: Get groceries

Command: list
1. Pay bills
2. Get groceries

Command: complete
Number: 1

Command: list
1. Pay bills (DONE!)
2. Get groceries

Command: switch
Enter number to select task list: 2
Business task list was selected.

Command:
```

To get this to work, you can hard code a list that stores the names of the two task lists (Personal and Business).

Implement the database tier and test it

10. Write the code that writes and reads the business objects to one or more files. For now, you can store the names of the task lists in one file, and you can store the data for each task list in a separate file. In the next chapter, you'll learn how to store data in a database, which has many advantages over storing it in one or more files.

LAB 17

Exercise 17-1 Review a SQLite database and test some SQL statements

This exercise gives you a chance to use DB Browser for SQLite to review the SQLite database that's used by the Movie List program that's presented in this chapter. You will also use DB Browser to test some SQL statements against this database.

Use DB Browser for SQLite to connect to the movies database

1. If you haven't already installed DB Browser for SQLite, do that now. For instructions, please see the appendix for your operating system.

2. Start DB Browser for SQLite. Then, use DB Browser to open the database named movies.sqlite that's in this directory:

    murach/python/_db

Review the tables in the movies database

3. Use the Browse Data tab to review the data in the Movie table.

4. Use the Browse Data tab to review the data in Category table.

Run SQL statements against the movies database

5. Use the Execute SQL tab to enter a query that selects all columns from the Movie table where the category ID is 2, and click the Run SQL button to execute this statement. This should display a result set.

6. Modify the value for the category ID in the query so it selects movies that have a category ID of 3. Then, run this query and view the result set.

7. Modify the query so it only returns the name and year columns. Then, run this query and view the result set.

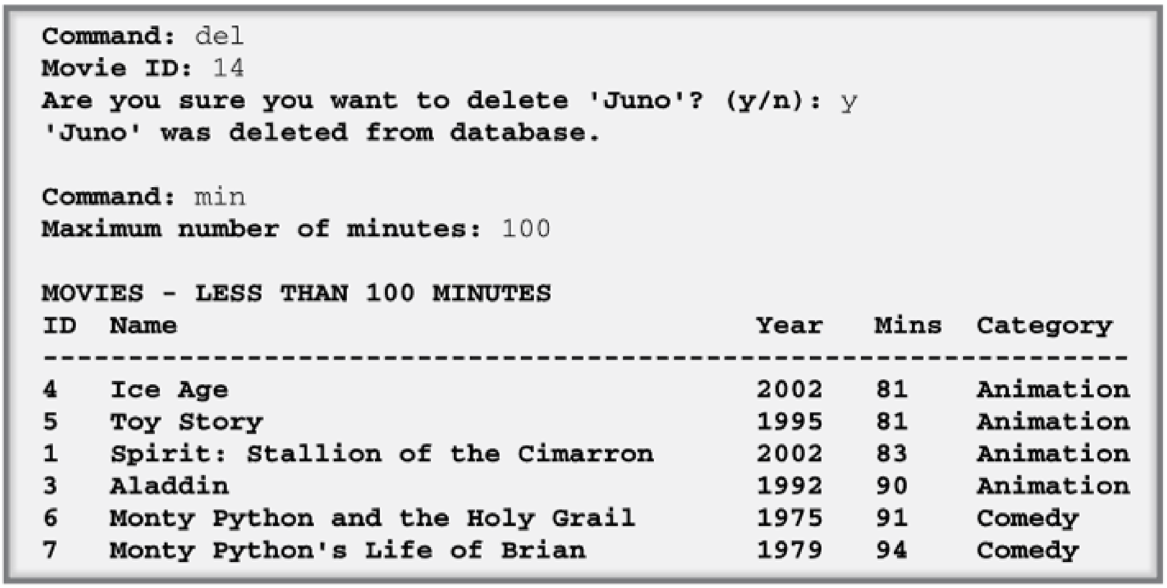
8. Modify the query so it sorts the result set by year in descending order.

9. Enter an INSERT statement that inserts a new row into the Movie table. Then, run this SQL statement. This shouldn't display a result set, but it should add a new row to the Movie table.
10. Use the Browse Data tab to browse the Movie table and view the new row.
11. Use the Execute SQL tab to run a DELETE statement that deletes the new row.
12. Use the Browse Data tab to make sure the row was deleted.

13. Continue to experiment until you're sure that you understand the SQL state-ments that are used by the Movie List program in this chapter.

Exercise 17-2 Enhance the Movie List program

In this exercise, you will enhance the Movie List program by improving its delete command and by adding a min command that lets the user view movies with run times that are less than a specific number of minutes.



```
Command: del
Movie ID: 14
Are you sure you want to delete 'Juno'? (y/n): y
'Juno' was deleted from database.

Command: min
Maximum number of minutes: 100

MOVIES - LESS THAN 100 MINUTES
ID  Name                                Year   Mins  Category
----------------------------------------------------------------
4   Ice Age                             2002   81    Animation
5   Toy Story                           1995   81    Animation
1   Spirit: Stallion of the Cimarron    2002   83    Animation
3   Aladdin                             1992   90    Animation
6   Monty Python and the Holy Grail     1975   91    Comedy
7   Monty Python's Life of Brian        1979   94    Comedy
```

Open and test the program

1. In IDLE, open the objects.py, db.py, and ui.py files that are in this folder:

python/exercises/ch17/movies

2. Review the code and note how the ui module uses the db module and the Movie class from the objects module. Then, run the program.

Improve the del command

3. In the db module, add a get_movie() function that gets a Movie object for the specified movie ID.
4. In the ui module, modify the delete_movie() function so it gets a Movie object for the specified ID and asks whether you are sure you want to delete the movie as shown above. This code should only delete the movie if the user enters "y" to confirm the operation.

Add the min command

5. In the db module, add a get_movies_by_minutes() function that gets a list of Movie objects that have a running time that's less than the number of minutes passed to it as an argument.

6. In the ui module, add a display_movies_by_minutes() function that calls the get_int() function to get the maximum number of minutes from the user and displays all selected movies. This should sort the movies by minutes in ascending order.

7. Modify the main() function and the display_menu() function so they provide for the min command.