

14

EXCEL SPREADSHEETS



Although we don't often think of spreadsheets as programming tools, almost everyone uses them to organize information into two-dimensional data structures, perform calculations with formulas, and produce output as charts. In the next two chapters, we'll integrate Python into two popular spreadsheet applications: Microsoft Excel and Google Sheets.

Excel is a popular and powerful spreadsheet application. The `openpyxl` module allows your Python programs to read and modify Excel spreadsheet files. For example, you might have the boring task of copying certain data from one spreadsheet and pasting it into another one. Or you might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria. Or you might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red. These are exactly the sorts of boring, mindless spreadsheet tasks that Python can do for you.

Although Excel is proprietary software from Microsoft, LibreOffice is a free alternative that runs on Windows, macOS, and Linux. LibreOffice Calc uses Excel's `.xlsx` file format for spreadsheets, which means the `openpyxl` module can work on spreadsheets from this application as well. You can download it from <https://www.libreoffice.org>. Even if you already have Excel installed on your computer, you may find this program easier to use. The screenshots in this chapter, however, are all from the cloud-based Office 365 Excel.

The `openpyxl` module operates on Excel files, and not the desktop Excel application or cloud-based Excel web app. If you're using the cloud-based Office 365, you must click **File ▶ Save As ▶ Download a Copy** to download a spreadsheet, run your Python script to edit the spreadsheet file, and then re-upload the spreadsheet to Office 365 to see

the changes. If you have the desktop Excel application, you must close the spreadsheet, run your Python script to edit the spreadsheet file, and then reopen it in Excel to see the changes.

Python does not come with `openpyxl`, so you'll have to install it. Appendix A has information on how to install third-party packages with Python's `pip` tool. You can find the full `openpyxl` documentation at <https://openpyxl.readthedocs.io/en/stable/>.

You'll use several example spreadsheet files in this chapter. You can download them from the book's online materials at <https://nostarch.com/automate-boring-stuff-python-3rd-edition>.

Reading Excel Files

First, let's go over some basic definitions. An Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the `.xlsx` extension.

Each workbook can contain multiple *sheets* (also called *worksheets*). The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*. Each sheet has *columns* (addressed by letters starting at *A*) and *rows* (addressed by numbers starting at 1). A box at a particular column and row is called a *cell*. Each cell can contain a number or text value. The grid of cells and their data makes up a sheet.

The examples in this chapter will use a spreadsheet named *example3.xlsx* stored in the current working directory. You can either create the spreadsheet yourself or download it from this book's online resources. Figure 14-1 shows the tabs for the three sheets named *Sheet1*, *Sheet2*, and *Sheet3*.

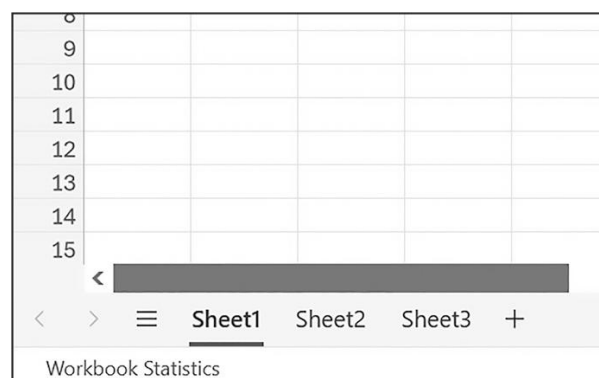


Figure 14-1: The tabs for a workbook's sheets are in the lower-left corner of Excel.

Sheet1 in the example file should look like Table 14-1. (If you didn't download *example3.xlsx*, you should enter this data into the sheet yourself.)

Table 14-1: The *example3.xlsx* Spreadsheet

| | A | B | C |
|---|----------------------|--------------|-----|
| 1 | 4/5/2035 1:34:02 PM | Apples | 73 |
| 2 | 4/5/2035 3:41:23 AM | Cherries | 85 |
| 3 | 4/6/2035 12:46:51 PM | Pears | 14 |
| 4 | 4/8/2035 8:59:43 AM | Oranges | 52 |
| 5 | 4/10/2035 2:07:00 AM | Apples | 152 |
| 6 | 4/10/2035 6:10:37 PM | Bananas | 23 |
| 7 | 4/10/2035 2:40:46 AM | Strawberries | 98 |

Now that we have our example spreadsheet, let's see how we can manipulate it with the `openpyxl` module.

Opening a Workbook

Once you've imported the `openpyxl` module, you'll be able to open *.xlsx* files with the `openpyxl.load_workbook()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

The `openpyxl.load_workbook()` function takes in the filename and returns a value of the `Workbook` data type. This `Workbook` object represents the Excel file, a bit like how a `File` object represents an opened text file.

Getting Sheets from the Workbook

You can get a list of all the sheet names in the workbook by accessing the `sheetnames` attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> wb.sheetnames    # The workbook's sheets'
```

```
names
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb['Sheet3'] # Get a sheet from
the workbook.
>>> sheet
<Worksheet "Sheet3">
>>> type(sheet)
<class
'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
# Get the sheet's title as a string.
'Sheet3'
>>> another_sheet = wb.active # Get the
active sheet.
>>> another_sheet
<Worksheet "Sheet1">
```

Each sheet is represented by a `Worksheet` object, which you can obtain by using the square brackets with the sheet name string, like a dictionary key. Finally, you can use the `active` attribute of a `Workbook` object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the `Worksheet` object, you can get its name from the `title` attribute.

Getting Cells from the Sheets

Once you have a `Worksheet` object, you can access a `Cell` object by its name. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1'] # Get a sheet from
the workbook.
>>> sheet['A1'] # Get a cell from the sheet.
<Cell 'Sheet1'.A1>
```

```

>>> sheet['A1'].value # Get the value from
the cell.
datetime.datetime(2035, 4, 5, 13, 34, 2)
>>> c = sheet['B1'] # Get another cell from
the sheet.
>>> c.value
'Apples'
>>> # Get the row, column, and value from
the cell.
>>> f'Row {c.row}, Column {c.column} is
{c.value}'
'Row 1, Column 2 is Apples'
>>> f'Cell {c.coordinate} is {c.value}'
'Cell B1 is Apples'
>>> sheet['C1'].value
73

```

The `Cell` object has a `value` attribute that contains, unsurprisingly, the value stored in that cell. It also has `row`, `column`, and `coordinate` attributes that provide location information for the cell. Here, accessing the `value` attribute of our `Cell` object for cell B1 gives us the string `'Apples'`. The `row` attribute gives us the integer 1, the `column` attribute gives us 2, and the `coordinate` attribute gives us `'B1'`.

The `openpyxl` module will automatically interpret the dates in column A and return them as `datetime` values rather than strings. Chapter 19 explains the `datetime` data type further.

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the `sheet.cell()` method and passing integers for its `row` and `column` keyword arguments. The first row or column integer is 1, not 0. Continue the interactive shell example by entering the following:

```

>>> sheet.cell(row=1, column=2)
<Cell 'Sheet1'.B1>
>>> sheet.cell(row=1, column=2).value

```

```
'Apples'
>>> for i in range(1, 8, 2): # Go through
every other row.
...     print(i, sheet.cell(row=i,
column=2).value)
...
1 Apples
3 Pears
5 Apples
7 Strawberries
```

Using the sheet's `cell()` method and passing it `row=1` and `column=2` gets you a `Cell` object for cell B1, just like specifying `sheet['B1']` did.

By using this `cell()` method and its keyword arguments, we wrote a `for` loop to print the values of a series of cells. Say you want to go down column B and print the value in every cell with an odd row number. By passing 2 for the `range()` function's "step" parameter, you can get cells from every second row (in this case, all the odd-numbered rows). This example passes the `for` loop's `i` variable as the `cell()` method's `row` keyword argument, and uses 2 for the `column` keyword argument on each call of the method. Note that this method accepts the integer 2, not the string 'B'.

You can determine the size of the sheet with the `Worksheet` object's `max_row` and `max_column` attributes. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> sheet.max_row # Get the highest row
number.
7
>>> sheet.max_column # Get the highest
```

column number.

3

Note that the `max_column` attribute is an integer rather than the letter that appears in Excel.

Converting Between Column Letters and Numbers

To convert from numbers to letters, call the `openpyxl.utils.get_column_letter()` function. To convert from letters to numbers, call the `openpyxl.utils.column_index_from_string()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.utils import
get_column_letter, column_index_from_string
>>> get_column_letter(1)    # Translate column
1 to a letter.
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> get_column_letter(sheet.max_column)
'C'
>>> column_index_from_string('A') # Get A's
number.
1
```

```
>>> column_index_from_string('AA')
```

```
27
```

After you import these two functions from the `openpyxl.utils` module, you can call `get_column_letter()` and pass it an integer like 27 to figure out what the letter name of the 27th column is. The function `column_index_from_string()` does the reverse: you pass it the letter name of a column, and it tells you what number that column is. You don't need to have a workbook loaded to use these functions.

Getting Rows and Columns

You can slice `Worksheet` objects to get all the `Cell` objects in a row, column, or rectangular area of the spreadsheet. Then, you can loop over all the cells in the slice. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> sheet['A1':'C3'] # Get cells A1 to C3.
((<Cell 'Sheet1'.A1>, <Cell 'Sheet1'.B1>,
<Cell 'Sheet1'.C1>), (<Cell 'Sheet1'.A2>,
<Cell
'Sheet1'.B2>, <Cell 'Sheet1'.C2>), (<Cell
'Sheet1'.A3>, <Cell 'Sheet1'.B3>, <Cell
'Sheet1'.C3>))
>>> for row_of_cell_objects in
sheet['A1':'C3']: ❶
...     for cell_obj in row_of_cell_objects:
❷
...         print(cell_obj.coordinate,
cell_obj.value)
...     print('--- END OF ROW ---')
...
```



```
B1 Apples
C1 73
--- END OF ROW ---
A2 2035-04-05 03:41:23
B2 Cherries
C2 85
--- END OF ROW ---
A3 2035-04-06 12:46:51
B3 Pears
C3 14
--- END OF ROW ---
```

Here, we specify `['A1' : 'C3']` to get a slice of the `Cell` objects in the rectangular area from A1 to C3, and we get a tuple containing the `Cell` objects in that area.

This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the `Cell` objects in one row of our desired area, from the leftmost cell to the rightmost. So, overall, our slice of the sheet contains all the `Cell` objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.

To print the values of each cell in the area, we use two `for` loops. The outer `for` loop goes over each row in the slice ❶. Then, for each row, the nested `for` loop goes through each cell in that row ❷.

To access the values of cells in a particular row or column, you can also use a `Worksheet` object's `rows` and `columns` attributes. These attributes must be converted to lists with the `list()` function before you can use the square brackets and an index with them. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> list(sheet.columns)[1] # Get the second
column's cells.
(<Cell 'Sheet1'.B1>, <Cell 'Sheet1'.B2>,
```

```
<Cell 'Sheet1'.B3>, <Cell 'Sheet1'.B4>, <Cell  
'Sheet1'.B5>, <Cell 'Sheet1'.B6>, <Cell  
'Sheet1'.B7>)  
>>> for cell_obj in list(sheet.columns)[1]:  
...     print(cell_obj.value)  
...  
Apples  
Cherries  
Pears  
Oranges  
Apples  
Bananas  
Strawberries
```

Using the `rows` attribute on a `Worksheet` object, passed to `list()`, will give us a list of tuples. Each of these tuples represents a row and contains the `Cell` objects in that row. The `columns` attribute, passed to `list()`, also gives us a list of tuples, with each of the tuples containing the `Cell` objects in a particular column. For *example3.xlsx*, because there are seven rows and three columns, `list(sheet.rows)` gives us a list of seven tuples (each containing three `Cell` objects), and `list(sheet.columns)` gives us a list of three tuples (each containing seven `Cell` objects).

To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you'd use `list(sheet.columns)[1]`. To get the tuple containing the `Cell` objects in column A, you'd use `list(sheet.columns)[0]`. Once you have a tuple representing one row or column, you can loop through its `Cell` objects and print their values.

A REVIEW OF WORKBOOKS, SHEETS, AND CELLS

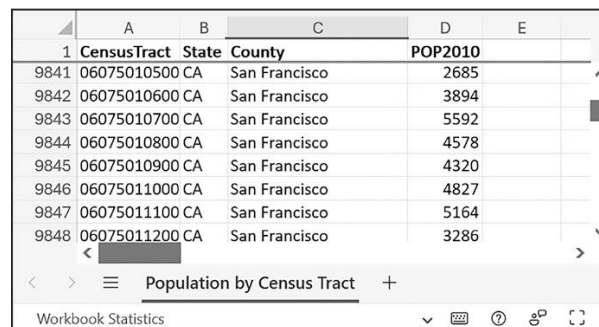
As a quick review, here's a rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

1. Import the `openpyxl` module.
2. Call the `openpyxl.load_workbook()` function to get a `Workbook` object.
3. Use the `active` or `sheetnames` attribute.

4. Get a `Worksheet` object.
5. Use indexing or the `cell()` sheet method with `row` and `column` keyword arguments.
6. Get a `Cell` object.
7. Read the `Cell` object's `value` attribute.

Project 9: Gather Census Statistics

Say you have a spreadsheet of data from the 2010 US Census and you've been given the boring task of going through its thousands of rows to count both the population and the number of census tracts for each county. (A *census tract* is simply a geographic area defined for the purposes of the census.) Each row represents a single census tract. We'll name the spreadsheet file *censuspopdata.xlsx*, and you can download it from this book's online resources. Its contents look like Figure 14-2.



| | A | B | C | D | E |
|------|-------------|-------|---------------|---------|---|
| 1 | CensusTract | State | County | POP2010 | |
| 9841 | 06075010500 | CA | San Francisco | 2685 | |
| 9842 | 06075010600 | CA | San Francisco | 3894 | |
| 9843 | 06075010700 | CA | San Francisco | 5592 | |
| 9844 | 06075010800 | CA | San Francisco | 4578 | |
| 9845 | 06075010900 | CA | San Francisco | 4320 | |
| 9846 | 06075011000 | CA | San Francisco | 4827 | |
| 9847 | 06075011100 | CA | San Francisco | 5164 | |
| 9848 | 06075011200 | CA | San Francisco | 3286 | |

Figure 14-2: The *censuspopdata.xlsx* spreadsheet

Even though Excel can automatically calculate the sum of multiple selected cells, you'd still have to first manually select the cells for each of the 3,000-plus counties. Even if it takes just a few seconds to calculate a county's population by hand, this would take hours to do for the whole spreadsheet.

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds.

This is what your program does:

- Reads the data from the Excel spreadsheet
- Counts the number of census tracts in each county
- Counts the total population of each county
- Prints the results

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module.

- Calculate all the tract and population data and store it in a data structure.
- Write the data structure to a text file with the `.py` extension using the `pprint` module so that it can be imported later.

Step 1: Read the Spreadsheet Data

There is just one sheet in the *censuspopdata.xlsx* spreadsheet, named 'Population by Census Tract', and each row in the sheet holds the data for a single census tract. The columns are the tract number (A), the state abbreviation (B), the county name (C), and the population of the tract (D).

Open a new file editor tab and enter the following code, then save the file as *readCensusExcel.py*:

```
# readCensusExcel.py - Tabulates county
population and census tracts

❶ import openpyxl, pprint
   print('Opening workbook...')

❷ wb =
   openpyxl.load_workbook('censuspopdata.xlsx')

❸ sheet = wb['Population by Census Tract']
   county_data = {}

# TODO: Fill in county_data with each
county's population and tracts.
print('Reading rows...')

❹ for row in range(2, sheet.max_row + 1):
    # Each row in the spreadsheet has data
    for one census tract.
        state = sheet['B' + str(row)].value
        county = sheet['C' + str(row)].value
        pop = sheet['D' + str(row)].value

# TODO: Open a new text file and write the
contents of county_data to it.
```

This code imports the `openpyxl` module, as well as the `pprint` module that you'll use to print the final county data ❶. Then, it opens the `censuspopdata.xlsx` file ❷, gets the sheet with the census data ❸, and begins iterating over its rows ❹.

Note that you've also created a variable named `county_data`, which will contain the populations and number of tracts you calculate for each county. Before you can store anything in it, though, you should determine exactly how you'll structure the data inside it.

Step 2: Populate the Data Structure

In the United States, states have two-letter abbreviations and are further split into counties. The data structure stored in `county_data` will be a dictionary with state abbreviations as its keys. Each state abbreviation will map to another dictionary, whose keys are strings of the county names in that state. Each county name will in turn map to a dictionary with just two keys, `'tracts'` and `'pop'`. These keys map to the number of census tracts and the population for the county. For example, the dictionary will look similar to this:

```
{ 'AK': { 'Aleutians East': { 'pop': 3141,
    'tracts': 1},
      'Aleutians West': { 'pop': 5561,
    'tracts': 2},
      'Anchorage': { 'pop': 291826,
    'tracts': 55},
      'Bethel': { 'pop': 17013, 'tracts':
3},
      'Bristol Bay': { 'pop': 997, 'tracts':
1},
      --snip--
```

If the previous dictionary were stored in `county_data`, the following expressions would evaluate like this:

```
>>> county_data['AK']['Anchorage']['pop']
291826
```

```
>>> county_data['AK']['Anchorage']['tracts']  
55
```

More generally, the `county_data` dictionary's keys will look like this:

```
county_data[state abbrev][county]['tracts']  
county_data[state abbrev][county]['pop']
```

Now that you know how `county_data` will be structured, you can write the code that will fill it with the county data. Add the following code to the bottom of your program:

```
# readCensusExcel.py - Tabulates county  
population and census tracts  
  
--snip--  
  
for row in range(2, sheet.max_row + 1):  
    # Each row in the spreadsheet has data  
    for one census tract.  
        state = sheet['B' + str(row)].value  
        county = sheet['C' + str(row)].value  
        pop    = sheet['D' + str(row)].value  
  
        # Make sure the key for this state  
exists.  
        ❶ county_data.setdefault(state, {})  
        # Make sure the key for this county in  
this state exists.  
        ❷ county_data[state].setdefault(county,  
{ 'tracts': 0, 'pop': 0 })  
  
        # Each row represents one census tract,
```

so increment by one.

```
    ❸ county_data[state][county]['tracts'] += 1
    # Increase the county pop by the pop in
    this census tract.
```

```
    ❹ county_data[state][county]['pop'] +=
    int(pop)
```

```
# TODO: Open a new text file and write the
contents of county_data to it.
```

The last two lines of code perform the actual calculation work, incrementing the value for `tracts` ❸ and increasing the value for `pop` ❹ for the current county on each iteration of the `for` loop.

The other code is there because you cannot add a county dictionary as the value for a state abbreviation key until the key itself exists in `county_data`. (That is, `county_data['AK']['Anchorage']['tracts'] += 1` will cause an error if the 'AK' key doesn't exist yet.) To make sure the state abbreviation key exists in your data structure, you need to call the `setdefault()` method to set a value if one does not already exist for `state` ❶.

Just as the `county_data` dictionary needs a dictionary as the value for each state abbreviation key, each of *those* dictionaries will need its own dictionary as the value for each county key ❷. And each of *those* dictionaries in turn will need the keys 'tracts' and 'pop' that start with the integer value 0. (If you ever lose track of the dictionary structure, look back at the example dictionary at the start of this section.)

Since `setdefault()` will do nothing if the key already exists, you can call it on every iteration of the `for` loop without a problem.

Step 3: Write the Results to a File

After the `for` loop has finished, the `county_data` dictionary will contain all of the population and tract information keyed by county and state. At this point, you could program more code to write this data to a text file or another Excel spreadsheet. For now, let's just use the `pprint.pformat()` function to write the `county_data` dictionary value as a massive string to a file named *census2010.py*. Add the following code to the bottom of your program (making sure to keep it un-indented so that it stays outside the `for` loop):

```
# readCensusExcel.py - Tabulates county
population and census tracts.
```

```
--snip--
```

```
# Open a new text file and write the contents
of county_data to it.
print('Writing results...')
result_file = open('census2010.py', 'w')
result_file.write('allData = ' +
pprint.pformat(county_data))
result_file.close()
print('Done.')
```

The `pprint.pformat()` function produces a string that itself is formatted as valid Python code. By outputting it to a text file named *census2010.py*, you've generated a Python program from your Python program! This may seem complicated, but the advantage is that you can now import *census2010.py* just like any other Python module. In the interactive shell, change the current working directory to the folder with your newly created *census2010.py* file and then import it:

```
>>> import census2010
>>> census2010.allData['AK']['Anchorage']
{'pop': 291826, 'tracts': 55}
>>> anchorage_pop = census2010.allData['AK']
['Anchorage']['pop']
>>> print('The 2010 population of Anchorage
was ' + str(anchorage_pop))
The 2010 population of Anchorage was 291826
```

The *readCensusExcel.py* program was throwaway code: once you have its results saved to *census2010.py*, you won't need to run the program again. Whenever you need the county data, you can just run `import census2010`.

Calculating this data by hand would have taken hours; this program did it in a few seconds. Using `openpyxl`, you'll have no trouble extracting information saved to an Excel spreadsheet and performing calculations on it. You can download the complete program from the book's online resources.

Ideas for Similar Programs

Many businesses and offices use Excel to store various types of data, and it's not uncommon for spreadsheets to become large and unwieldy. Any program that parses an Excel spreadsheet has a similar structure: it loads the spreadsheet file, preps some variables or data structures, and then loops through each of the rows in the spreadsheet. Such a program could do the following:

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.
- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

Writing Excel Documents

The `openpyxl` module also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, creating spreadsheets with thousands of rows of data is simple.

Creating and Saving Excel Files

Call the `openpyxl.Workbook()` function to create a new, blank `Workbook` object. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()    # Create a
blank workbook.
>>> wb.sheetnames
# The workbook starts with one sheet.
['Sheet']
>>> sheet = wb.active
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'    #
```

Change the title.

```
>>> wb.sheetnames  
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named *Sheet*. You can change the name of the sheet by storing a new string in its `title` attribute.

Anytime you modify the `Workbook` object or its sheets and cells, the spreadsheet file will not be saved until you call the `save()` workbook method. Enter the following into the interactive shell (with *example3.xlsx* in the current working directory):

```
>>> import openpyxl  
>>> wb =  
openpyxl.load_workbook('example3.xlsx')  
>>> sheet = wb['Sheet1']  
>>> sheet.title = 'Spam Spam Spam'  
>>> wb.save('example3_copy.xlsx')  
# Save the workbook.
```

Here, we change the name of our sheet. To save our changes, we pass a filename as a string to the `save()` method. Passing a different filename than the original, such as `'example3_copy.xlsx'`, saves the changes to a copy of the spreadsheet.

Whenever you edit a spreadsheet you've loaded from a file, you should always save the new, edited spreadsheet with a different filename than the original. That way, you'll still have the original spreadsheet file to work with in case a bug in your code caused the new, saved file to contain incorrect or corrupted data. Also, the `save()` method won't work if the spreadsheet is currently open in the Excel desktop application. You must first close the spreadsheet and then run your Python program.

Creating and Removing Sheets

You can create or delete sheets from a workbook with the `create_sheet()` method and `del` operator. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.sheetnames
['Sheet']
>>> wb.create_sheet() # Add a new sheet.
<Worksheet "Sheet1">
>>> wb.sheetnames
['Sheet', 'Sheet1']
>>> # Create a new sheet at index 0.
>>> wb.create_sheet(index=0, title='First
Sheet')
<Worksheet "First Sheet">
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle
Sheet')
<Worksheet "Middle Sheet">
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Middle Sheet',
'Sheet1']
```

The `create_sheet()` method returns a new `Worksheet` object named `SheetX`, which by default is the last sheet in the workbook. Optionally, you can specify the index and name of the new sheet with the `index` and `title` keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Middle Sheet',
'Sheet1']
>>> del wb['Middle Sheet']
>>> del wb['Sheet1']
>>> wb.sheetnames
['First Sheet', 'Sheet']
```

You can use the `del` operator to delete a sheet from a workbook, just like you can use it to delete a key-value pair from a dictionary.

Remember to call the `save()` method to save the changes after adding sheets to or removing sheets from the workbook.

Writing Values to Cells

Writing values to cells is much like writing values to keys in a dictionary. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 'Hello, world!'
# Edit the cell's value.
>>> sheet['A1'].value
'Hello, world!'
```

If you have the cell's coordinate as a string, you can use it just like a dictionary key on the `Worksheet` object to specify which cell to write to.

Project 10: Update a Spreadsheet

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their prices. Download this *produceSales3.xlsx* spreadsheet from the book's online resources. Figure 14-3 shows what the spreadsheet looks like.

| | A | B | C | D | E |
|----|----------------|-----------------------|--------------------|--------------|---|
| 1 | PRODUCE | COST PER POUND | POUNDS SOLD | TOTAL | |
| 2 | Potatoes | 0.86 | 21.6 | 18.58 | |
| 3 | Okra | 2.26 | 38.6 | 87.24 | |
| 4 | Fava beans | 2.69 | 32.8 | 88.23 | |
| 5 | Watermelon | 0.66 | 27.3 | 18.02 | |
| 6 | Garlic | 1.19 | 4.9 | 5.83 | |
| 7 | Parsnips | 2.27 | 1.1 | 2.5 | |
| 8 | Asparagus | 2.49 | 37.9 | 94.37 | |
| 9 | Avocados | 3.23 | 9.2 | 29.72 | |
| 10 | Celery | 3.07 | 28.9 | 88.72 | |
| 11 | Okra | 2.26 | 40 | 90.4 | |

Figure 14-3: A spreadsheet of produce sales

Each row represents an individual sale. The columns are the type of produce sold (A), the cost per pound of that produce (B), the number of pounds sold (C), and the total revenue from the sale (D). The TOTAL column is set to an Excel formula like `=ROUND(B2*C2, 2)`, which multiplies the row's cost per pound by the number of pounds sold and rounds the result to the nearest cent. With this formula, the cells in the TOTAL column will automatically update themselves if there is a change in the COST PER POUND and POUNDS SOLD columns.

Now imagine that the prices of garlic, celery, and lemons were entered incorrectly, leaving you with the boring task of going through thousands of rows in this spreadsheet to update the cost per pound for any celery, garlic, and lemon rows. You can't do a simple find-and-replace for the price, because there might be other items with the same price that you don't want to mistakenly "correct." For thousands of rows, this would take hours to do by hand. But you can write a program that can accomplish this in seconds.

Your program should do the following:

- Loop over all the rows.
- If the row is for celery, garlic, or lemons, change the price.

This means your code will need to do the following:

- Open the spreadsheet file.
- For each row, check whether the value in column A is Celery, Garlic, or Lemon.
- If it is, update the price in column B.
- Save the spreadsheet to a new file (so that you don't lose the original spreadsheet, just in case).

Step 1: Set Up a Data Structure with the Updated Information

The prices that you need to update are as follows:

- Celery: 1.19
- Garlic: 3.07
- Lemon: 1.27

You could write code to set these new prices, like this:

```
if produce_name == 'Celery':  
    cell_obj = 1.19  
if produce_name == 'Garlic':  
    cell_obj = 3.07  
if produce_name == 'Lemon':  
    cell_obj = 1.27
```

But hardcoding the produce and updated price data like this is a bit inelegant. If you needed to update the spreadsheet again with different prices or different produce, you would have to change a lot of the code. Every time you change code, you risk introducing bugs.

A more flexible solution is to store the corrected price information in a dictionary and write your code to use this data structure. In a new file editor tab, enter the following code:

```
# updateProduce.py - Corrects costs in  
produce sales spreadsheet  
  
import openpyxl  
  
wb =  
openpyxl.load_workbook('produceSales3.xlsx')  
sheet = wb['Sheet']  
  
# The produce types and their updated prices  
PRICE_UPDATES = {'Garlic': 3.07,  
                  'Celery': 1.19,
```

```
# TODO: Loop through the rows and update the
prices.
```

Save this as *updateProduce.py*. If you need to update the spreadsheet again, you'll need to update only the `PRICE_UPDATES` dictionary, not any other code.

Step 2: Check All Rows and Update Incorrect Prices

The next part of the program will loop through all the rows in the spreadsheet. Add the following code to the bottom of *updateProduce.py*:

```
# updateProduce.py - Corrects costs in
produce sales spreadsheet

--snip--

# Loop through the rows and update the
prices.
❶ for row_num in range(2, sheet.max_row +
1): # Skip the first row.
    ❷ produce_name = sheet.cell(row=row_num,
column=1).value
    ❸ if produce_name in PRICE_UPDATES:
        sheet.cell(row=row_num,
column=2).value = PRICE_UPDATES[produce_name]

❹ wb.save('updatedProduceSales3.xlsx')
```

We loop through the rows starting at row 2, as row 1 is just the header ❶. The cell in column 1 (that is, column A) will be stored in the variable `produce_name` ❷. If `produce_name` exists as a key in the `PRICE_UPDATES` dictionary ❸, you know this row needs its price

corrected. The correct price will be in `PRICE_UPDATES[produce_name]`.

Notice how clean using `PRICE_UPDATES` makes the code. It uses only one `if` statement, rather than a separate line like `if produce_name == 'Garlic':` for every type of produce to update. And since the code uses the `PRICE_UPDATES` dictionary instead of hardcoding the produce names and updated costs into the `for` loop, you can modify only the `PRICE_UPDATES` dictionary, and not the rest of the code, if the produce sales spreadsheet needs additional changes.

After going through the entire spreadsheet and making changes, the code saves the `Workbook` object to *updatedProduceSales3.xlsx* ④. It doesn't overwrite the old spreadsheet, in case there's a bug in the program and the updated spreadsheet is wrong. After checking that the updated spreadsheet looks right, you can delete the old spreadsheet.

Ideas for Similar Programs

Since many office workers use Excel spreadsheets all the time, a program that can automatically edit and write Excel files could be really useful. Such a program could do the following:

- Read data from one spreadsheet and write it to parts of other spreadsheets.
- Read data from websites, text files, or the clipboard and write it to a spreadsheet.
- Automatically “clean up” data in spreadsheets. For example, it could use regular expressions to read multiple formats of phone numbers and edit them to a single, standard format.

Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. In the produce spreadsheet, for example, your program could apply bold text to the potato, garlic, and parsnip rows. Or perhaps you want to italicize every row with a cost per pound greater than \$5. Styling parts of a large spreadsheet by hand would be tedious, but your programs can do it instantly.

To customize font styles in cells, import the `Font()` function from the `openpyxl.styles` module:

```
from openpyxl.styles import Font
```

Importing the function in this way allows you to write `Font()` instead of `openpyxl.styles.Font()`. (See “Importing Modules” on Chapter 3 for more information.)

The following example creates a new workbook and sets cell A1 to a 24-point italic font:

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
❶ >>> italic_24_font = Font(size=24,
    italic=True)
❷ >>> sheet['A1'].font = italic_24_font
>>> sheet['A1'] = 'Hello, world!'
>>> wb.save('styles3.xlsx')
```

In this example, `Font(size=24, italic=True)` returns a `Font` object, which we store in `italic_24_font` ❶. The keyword arguments to `Font()` configure the object’s styling information, and assigning `sheet['A1'].font` the `italic_24_font` object ❷ applies all of that font-styling information to cell A1.

To set `font` attributes, pass keyword arguments to `Font()`. Table 14-2 shows the possible keyword arguments for the `Font()` function.

Table 14-2: Keyword Arguments for `Font` Objects

| Keyword argument | Data type | Description |
|---------------------|-----------|---|
| <code>name</code> | String | The font name, such as 'Calibri' or 'Times New Roman' |
| <code>size</code> | Integer | The point size |
| <code>bold</code> | Boolean | <code>True</code> , for bold font |
| <code>italic</code> | Boolean | <code>True</code> , for italic font |

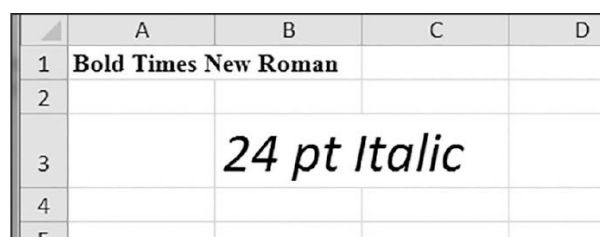
You can call `Font()` to create a `Font` object and store that `Font` object in a variable. You then assign that variable to a `Cell` object’s `font` attribute. For example, this code creates various font styles:

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> bold_font = Font(name='Times New Roman',
bold=True)
>>> sheet['A1'].font = bold_font
>>> sheet['A1'] = 'Bold Times New Roman'

>>> italic_font = Font(size=24, italic=True)
>>> sheet['B3'].font = italic_font
>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles3.xlsx')
```

Here, we store a `Font` object in `bold_font` and then set the A1 Cell object's `font` attribute to `bold_font`. We repeat the process with another `Font` object to set the font of a second cell. After you run this code, the styles of the A1 and B3 cells in the spreadsheet will have custom font styles, as shown in Figure 14-4.



| | A | B | C | D |
|---|-----------------------------|---------------------|---|---|
| 1 | Bold Times New Roman | | | |
| 2 | | | | |
| 3 | | <i>24 pt Italic</i> | | |
| 4 | | | | |
| 5 | | | | |

Figure 14-4: A spreadsheet with custom font styles

For cell A1, we set the font name to 'Times New Roman' and set `bold` to `true`, so the text appears in bold Times New Roman. We didn't specify a point size, so the text uses the `openpyxl` default, 11. In cell B3, the text is italic, with a point size of 24. We didn't specify a font name, so the text uses the `openpyxl` default, Calibri.

Formulas

Excel formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells. In this section, you'll use the

openpyxl module to programmatically add formulas to cells, just as you would add any normal value. Here is an example:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

This code will store the formula `=SUM(B1:B8)` in B9, setting the cell's value to the sum of values in cells B1 to B8. You can see this in action in Figure 14-5.

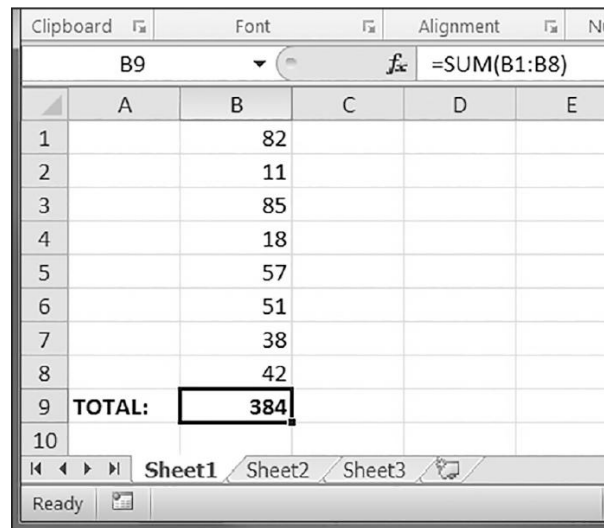


Figure 14-5: Cell B9 contains the formula to add the values in cells B1 to B8.

You can set an Excel formula just like any other text value in a cell. For instance, enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 200
>>> sheet['A2'] = 300
>>> sheet['A3'] = '=SUM(A1:A2)' # Set the
formula.
>>> wb.save('writeFormula3.xlsx')
```

The cells in A1 and A2 are set to 200 and 300, respectively. The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spreadsheet is opened in Excel, A3 will display its value as 500.

The `openpyxl` module doesn't have the ability to calculate Excel formulas and populate cells with the results. However, if you open this *writeFormula3.xlsx* file in Excel, Excel itself will populate the cells with the formula results. You can save the file in Excel, and then open it while passing the `data_only=True` keyword argument to `openpyxl.load_workbook()`, and the cell values should show the calculation results instead of the formula string:

```
>>> # Be sure to open writeFormula3.xlsx in
Excel and save it first.
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('writeFormula3.xlsx')
# Open without data_only.
>>> wb.active['A3'].value # Get the formula
string.
'=SUM(A1:A2) '
>>> wb =
openpyxl.load_workbook('writeFormula3.xlsx',
data_only=True) # Open with data_only.
>>> wb.active['A3'].value # Get the formula
result.
500
```

Again, you'll only see the 500 result in the spreadsheet file if you opened and saved it in Excel so that Excel could run the formula calculation and store the result in the spreadsheet file. This is the value `openpyxl` reads when you pass `data_only=True` to `openpyxl.load_workbook()`.

Excel formulas offer a level of programmability for spreadsheets, but they can quickly become unmanageable for complicated tasks. For example, even if you're deeply familiar with Excel formulas, it's a headache to try to decipher what `=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!A1:B10000, 2, FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!A1:B10000, 2, FALSE), " ", "")), ""))` actually does. Python code is much more readable.

Adjusting Rows and Columns

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header. But if you need to set the size of a row or column based on its cells' contents, or if you want to set sizes in a large number of spreadsheet files, it's much quicker to write a Python program to do it.

You can also hide rows and columns from view, or “freeze” them in place so that they're always visible on the screen, appearing on every page when you print the spreadsheet (which is handy for headers).

Setting Row Height and Column Width

A `Worksheet` object has `row_dimensions` and `column_dimensions` attributes that control row heights and column widths. For example, enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 'Tall row'
>>> sheet['B2'] = 'Wide column'
>>> sheet.row_dimensions[1].height = 70
>>> sheet.column_dimensions['B'].width = 20
>>> wb.save('dimensions3.xlsx')
```

A sheet's `row_dimensions` and `column_dimensions` are dictionary-like values; `row_dimensions` contains `RowDimension` objects, and `column_dimensions` contains `ColumnDimension` objects. In `row_dimensions`, you can access one of the objects using the number of the row (in this case, 1 or 2). In `column_dimensions`, you can access one of the objects using the letter of the column (in this case, A or B).

The *dimensions3.xlsx* spreadsheet looks like Figure 14-6.

| | A | B | |
|---|----------|-------------|--|
| | | | |
| 1 | Tall row | | |
| 2 | | Wide column | |
| 3 | | | |

Figure 14-6: Row 1 and column B set to larger heights and widths

The default width and height of cells varies between versions of Excel and `openpyxl`.

Merging and Unmerging Cells

You can merge a rectangular group of cells into a single cell with the `merge_cells()` sheet method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet.merge_cells('A1:D3') # Merge all
these cells.
>>> sheet['A1'] = 'Twelve cells merged
together.'
>>> sheet.merge_cells('C5:D5')
# Merge these two cells.
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged3.xlsx')
```

The argument to `merge_cells()` is a single string of the top-left and bottom-right cells of the rectangular area to be merged: `'A1:D3'` merges 12 cells into a single cell. To set the value of these merged cells, simply set the value of the top-left cell of the merged group.

When you run this code, *merged.xlsx* will look like Figure 14-7.

| | A | B | C | D | E |
|---|-------------------------------|---|-------------------|---|---|
| 1 | | | | | |
| 2 | | | | | |
| 3 | Twelve cells merged together. | | | | |
| 4 | | | | | |
| 5 | | | Two merged cells. | | |
| 6 | | | | | |
| 7 | | | | | |

Figure 14-7: Merged cells in a spreadsheet

To unmerge cells, call the `unmerge_cells()` sheet method:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('merged3.xlsx')
>>> sheet = wb['Sheet']
>>> sheet.unmerge_cells('A1:D3')    # Split
these cells up.
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('unmerged3.xlsx')
```

If you save your changes and then take a look at the spreadsheet, you'll see that the merged cells have gone back to being individual cells.

Freezing Panes

For spreadsheets that are too large to be displayed all at once, it's helpful to “freeze” a few of the top rows or leftmost columns onscreen. Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as *freeze panes*.

In `openpyxl`, each `Worksheet` object has a `freeze_panes` attribute that you can set to a `Cell` object or a string of a cell's coordinates. Note that this attribute will freeze all rows above this cell and all columns to the left of it, but not the row and column of the cell itself. To unfreeze all panes, set `freeze_panes` to `None` or `'A1'`. Table 14-3 shows which rows and columns get frozen for some example settings of `freeze_panes`.

Table 14-3: Frozen Pane Examples

| freeze_panes setting | Rows and columns frozen |
|--|--------------------------------|
| <code>sheet.freeze_panes = 'A2'</code> | Row 1 (no columns frozen) |

| freeze_panes setting | Rows and columns frozen |
|---|----------------------------------|
| <code>sheet.freeze_panes = 'B1'</code> | Column A (no rows frozen) |
| <code>sheet.freeze_panes = 'C1'</code> | Columns A and B (no rows frozen) |
| <code>sheet.freeze_panes = 'C2'</code> | Row 1 and columns A and B |
| <code>sheet.freeze_panes = 'A1'</code> or <code>sheet.freeze_panes = None</code> | No frozen rows or columns |

Download another copy of the *produceSales3.xlsx* spreadsheet, then enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb =
openpyxl.load_workbook('produceSales3.xlsx')
>>> sheet = wb.active
>>> sheet.freeze_panes = 'A2' # Freeze the
rows above A2.
>>> wb.save('freezeExample3.xlsx')
```

You can see the result in Figure 14-8.

| | A | B | C | D | E | F |
|------|---------------|----------------|-------------|--------|---|---|
| 1 | FRUIT | COST PER POUND | POUNDS SOLD | TOTAL | | |
| 1591 | Fava beans | 2.69 | 0.7 | 1.88 | | |
| 1592 | Grapefruit | 0.76 | 28.5 | 21.66 | | |
| 1593 | Green peppers | 1.89 | 37 | 69.93 | | |
| 1594 | Watermelon | 0.66 | 30.4 | 20.06 | | |
| 1595 | Celery | 3.07 | 36.6 | 112.36 | | |
| 1596 | Strawberries | 4.4 | 5.5 | 24.2 | | |
| 1597 | Green beans | 2.52 | 40 | 100.8 | | |

Figure 14-8: Freezing row 1

Because you set the `freeze_panes` attribute to `'A2'`, row 1 will remain visible, no matter where the user scrolls in the spreadsheet.

Charts

The `openpyxl` module supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

1. Create a `Reference` object from a rectangular selection of cells.
2. Create a `Series` object by passing in the `Reference` object.
3. Create a `Chart` object.
4. Append the `Series` object to the `Chart` object.
5. Add the `Chart` object to the `Worksheet` object, optionally specifying which cell should be the top-left corner of the chart.

The `Reference` object requires some explaining. To create `Reference` objects, you must call the `openpyxl.chart.Reference()` function and pass five arguments:

- The `Worksheet` object containing your chart data.
- The column and row integer of the top-left cell of the rectangular selection of cells containing your chart data: the first integer in the tuple is the row, and the second is the column. Note that 1 is the first row, not 0.
- The column and row integer of the bottom-right cell of the rectangular selection of cells containing your chart data: the first integer in the tuple is the row, and the second is the column.

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> for i in range(1, 11): # Create some
data in column A.
...     sheet['A' + str(i)] = i * i
...
>>>
ref_obj = openpyxl.chart.Reference(sheet, 1,
1, 1, 10)

>>> series_obj =
```

```
openpyxl.chart.Series(ref_obj, title='First
series')
```

```
>>> chart_obj = openpyxl.chart.BarChart()
>>> chart_obj.title = 'My Chart'
>>> chart_obj.append(series_obj)
```

```
>>> sheet.add_chart(chart_obj, 'C5')
>>> wb.save('sampleChart3.xlsx')
```

This code produces a spreadsheet that looks like Figure 14-9.

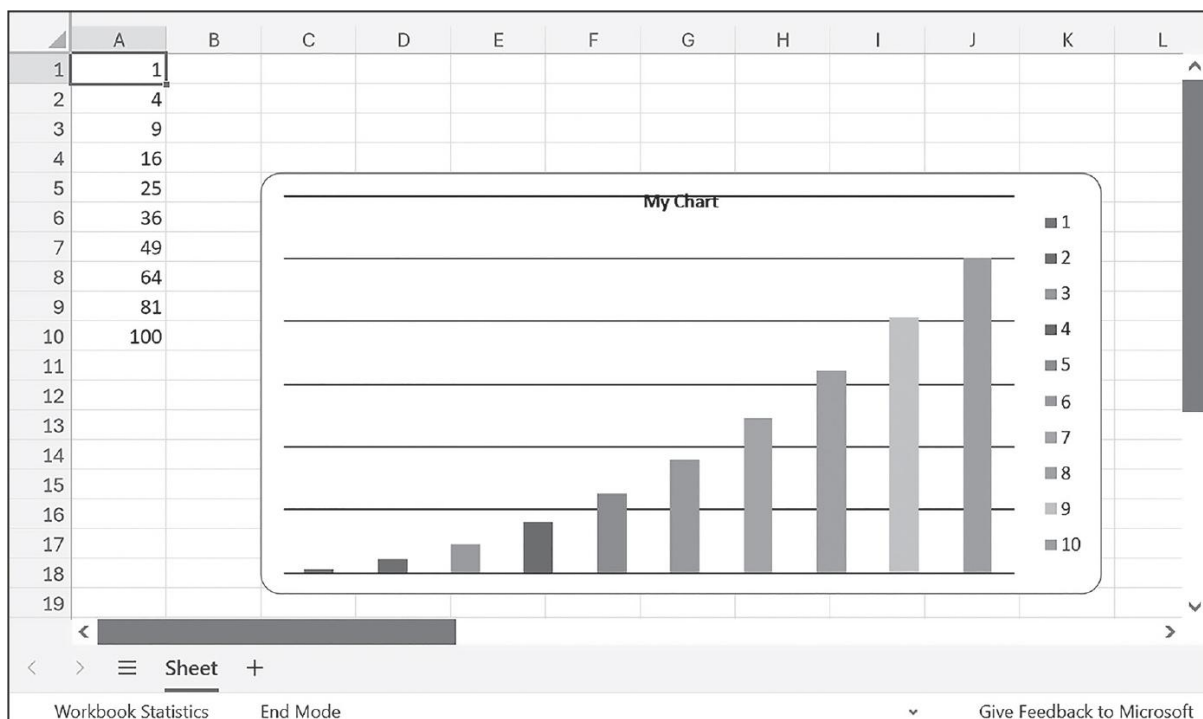


Figure 14-9: A spreadsheet with a chart added

We've created a bar chart by calling `openpyxl.chart.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.chart.LineChart()`, `openpyxl.chart.ScatterChart()`, and `openpyxl.chart.PieChart()`.

Summary

Often, the hard part of processing information isn't the processing itself but simply getting the data in the right format for your program. But

once you've loaded your Excel spreadsheet into Python, you can extract and manipulate its data much faster than you could by hand.

You can also generate spreadsheets as output from your programs. So, if colleagues need a text file or PDF of thousands of sales contacts transferred to a spreadsheet file, you won't have to tediously copy and paste it all into Excel. Equipped with the `openpyxl` module and some programming knowledge, you'll find processing even the biggest spreadsheets a piece of cake.

In the next chapter, we'll take a look at using Python to interact with another spreadsheet program: the popular online Google Sheets application.

Practice Questions

For the following questions, imagine you have a `Workbook` object in the variable `wb`, a `Worksheet` object in `sheet`, and a `Cell` object in `cell`.

1. What does the `openpyxl.load_workbook()` function return?
2. What does the `wb.sheetnames` workbook attribute contain?
3. How would you retrieve the `Worksheet` object for a sheet named `'Sheet1'`?
4. How would you retrieve the `Worksheet` object for the workbook's active sheet?
5. How would you retrieve the value in cell C5?
6. How would you set the value in cell C5 to `"Hello"`?
7. How would you retrieve the cell's row and column as integers?
8. What do the `sheet.max_column` and `sheet.max_row` sheet attributes hold, and what is the data type of these attributes?
9. If you needed to get the integer index for column `'M'`, what function would you need to call?
10. If you needed to get the string name for row 14, what function would you need to call?
11. How can you retrieve a tuple of all the `Cell` objects from A1 to F1?
12. How would you save the workbook to the filename `example3.xlsx`?
13. How do you set a formula in a cell?
14. If you want to retrieve the result of a cell's formula instead of the cell's formula itself, what must you do first?
15. How would you set the height of row 5 to 100?
16. How would you hide column C?
17. What is a freeze pane?

18. What five functions and methods do you have to call to create a bar chart?

Practice Programs

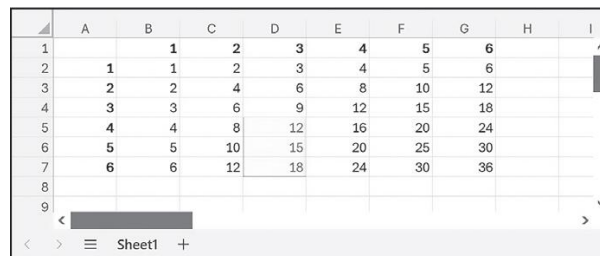
For practice, write programs to do the following tasks.

Multiplication Table Maker

Create a program *multiplicationTable.py* that takes a number N from the command line and creates an $N \times N$ multiplication table in an Excel spreadsheet. For example, when the program is run like this

```
py multiplicationTable.py 6
```

it should create a spreadsheet that looks like Figure 14-10.



| | A | B | C | D | E | F | G | H | I |
|---|---|----------|----------|----------|----------|----------|----------|---|---|
| 1 | | 1 | 2 | 3 | 4 | 5 | 6 | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | | |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | | |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | | |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | | |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | | |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | | |
| 8 | | | | | | | | | |
| 9 | | | | | | | | | |

Figure 14-10: A multiplication table generated in a spreadsheet

Row 1 and column A should contain labels and be in bold.

Blank Row Inserter

Create a program *blankRowInserter.py* that takes two integers and a filename string as command line arguments. Let's call the first integer N and the second integer M . Starting at row N , the program should insert M blank rows into the spreadsheet. For example, when the program is run like this

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

the “before” and “after” spreadsheets should look like Figure 14-11.

| | A1 Potatoes | | | | | |
|---|-------------|-----------|------------|------------|------------|---------|
| | A | B | C | D | E | F |
| 1 | Potatoes | Celery | Ginger | Yellow per | Green bea | Fava be |
| 2 | Okra | Okra | Corn | Garlic | Tomatoes | Yellow |
| 3 | Fava bean | Spinach | Grapefruit | Grapes | Apricots | Papaya |
| 4 | Watermel | Cucumber | Ginger | Watermel | Red onion | Butterr |
| 5 | Garlic | Apricots | Eggplant | Cherries | Strawberri | Apricot |
| 6 | Parsnips | Okra | Cucumber | Apples | Grapes | Avocad |
| 7 | Asparagus | Fava bean | Green cab | Grapefruit | Ginger | Butterr |
| 8 | Avocados | Watermel | Eggplant | Grapes | Strawberri | Celery |

| | A1 Potatoes | | | | | |
|---|-------------|----------|------------|------------|------------|---------|
| | A | B | C | D | E | F |
| 1 | Potatoes | Celery | Ginger | Yellow per | Green bea | Fava be |
| 2 | Okra | Okra | Corn | Garlic | Tomatoes | Yellow |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | Fava bean | Spinach | Grapefruit | Grapes | Apricots | Papaya |
| 6 | Watermel | Cucumber | Ginger | Watermel | Red onion | Butterr |
| 7 | Garlic | Apricots | Eggplant | Cherries | Strawberri | Apricot |
| 8 | Parsnips | Okra | Cucumber | Apples | Grapes | Avocad |

Figure 14-11: Before (left) and after (right) the two blank rows are inserted at row 3

You can write this program by reading in the contents of the spreadsheet. Then, when writing out the new spreadsheet, use a `for` loop to copy the first N lines. For the remaining lines, add M to the row number in the output spreadsheet.