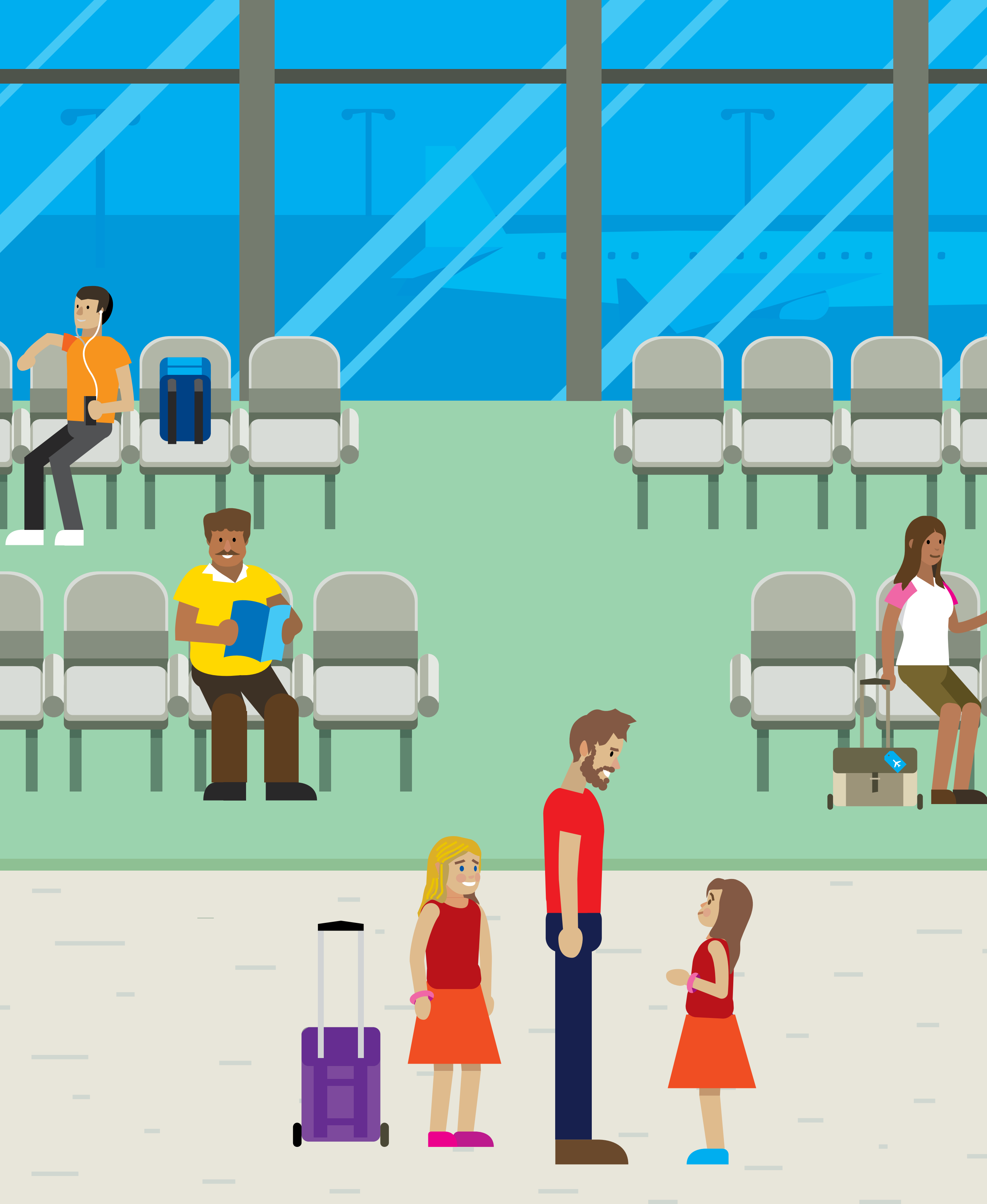
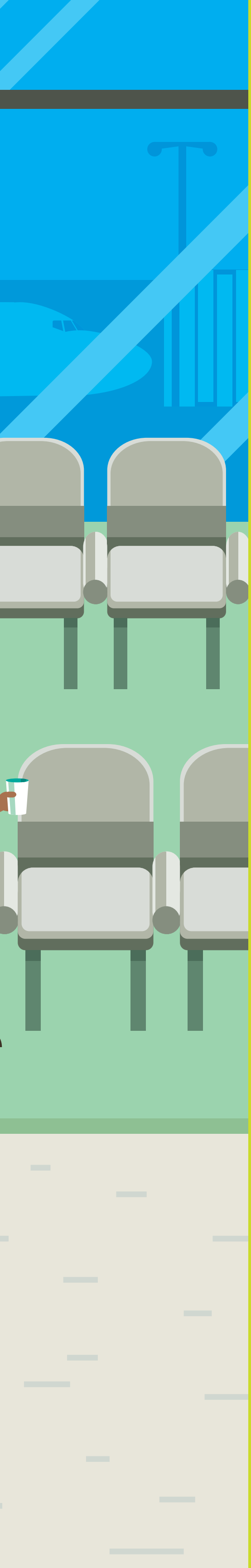




Part 3
Useful Python

Parts one and two gave you a good foundation in the Python language and a good understanding of software design. You've built some substantial applications, and hopefully you've built some of your own programs, too. You also know about the importance of testing and documentation and have seen the powerful Python tools that can help you with these tasks.

Now it's time to move on to the really cool stuff. In this third part, you'll learn how to make Python programs that have graphical user interfaces, talk to the Internet, and work over the network. Then, we'll round things off with an exploration of game development in Python.

In this part, the balance of the content changes slightly. There will be a bit less talking and a lot more doing. Expect to see more "Make Something Happen" sections as we explore how to build useful applications using popular Python frameworks. We'll also have more "Make Something Happen: Development Challenges" where you can take our example code and "run with it" to create programs of your own.

13
Python and Graphical User Interfaces

What you will learn

This chapter begins with a description of a new Integrated Development Environment for Python. The IDLE editor we've been using for the previous twelve chapters is a great place to learn to program, but we're starting to find limitations with the way it works, particularly now that we're creating applications that span several source files. The Visual Studio Code application provides a flexible and fast place to work with Python, so we'll look at how we can use it to develop our code.

Once we've sorted out our new workplace, we'll look at the creation of Python programs that have a Graphical User Interface (GUI). We'll use the popular Tkinter module that is supplied with the Python language and allows us to write easy-to-use programs that work on any device that supports Python. We'll discover how to create labels, text entry boxes, buttons, and list boxes. We'll then find out about events, create a drawing program, and finish off with a GUI version of the Fashion Shop application.

Visual Studio Code

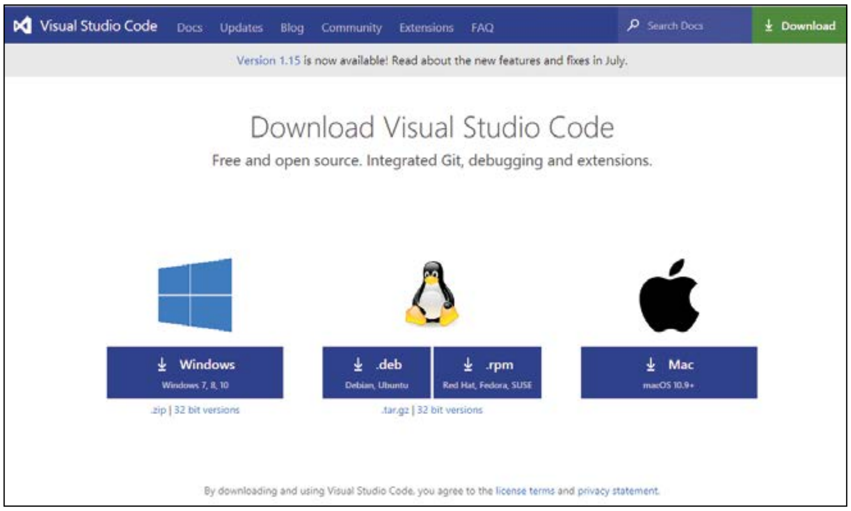
The IDLE program supplied with Visual Studio is a great place to learn how to program. However, as we begin to write larger programs, we start to notice that it has some limitations. If you want to make a program out of several Python source files (as we've begun to do now that we're using modules), the IDLE experience is not a good one. You must remember to save all open file windows before you run your program; otherwise, it might not incorporate all the latest changes to your code.

Visual Studio Code is a free and lightweight program editor from Microsoft. It's available for a wide range of operating systems, including Windows, Mac, and Linux. It's an open-source project, so you can even take a look inside the Visual Studio Code program source code and discover exactly how it works. Visual Studio Code is not tied to working with any specific programming language; it supports *plugins* that can be installed within the editor to customize it. We'll install Visual Studio Code and then use a very popular Python plugin from open-source contributor Don Jayamanne.

We'll still use IDLE from time to time, though, as it's still a great place to use the Python Command Shell.

Install Visual Studio Code

You can download a copy of Visual Studio Code for your machine from https://code.visualstudio.com/Download, which you can see in **Figure 13-1**.



Version 1.15 is now available! Read about the new features and fixes in July.

Download Visual Studio Code

Free and open source. Integrated Git, debugging and extensions.

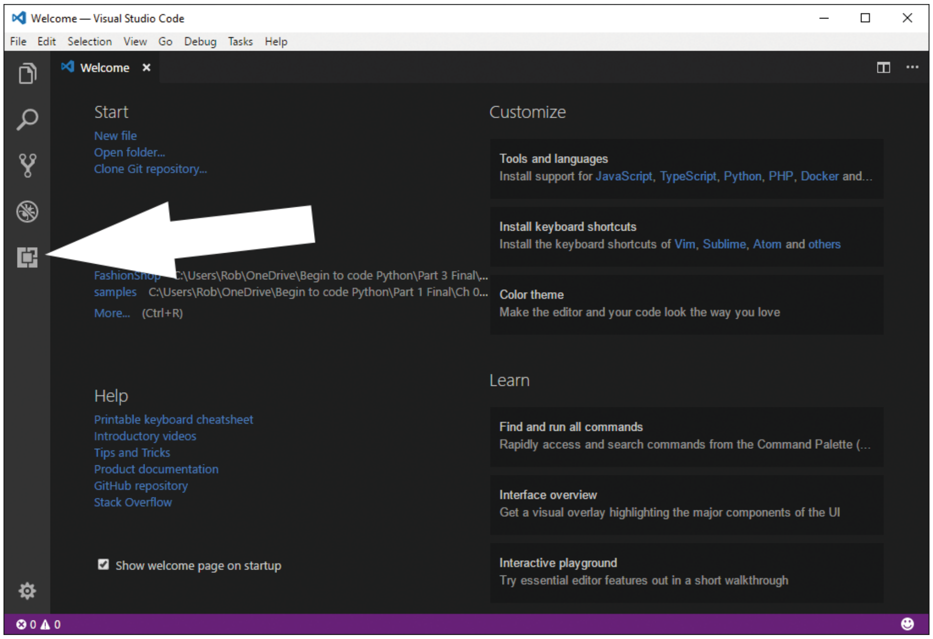
By downloading and using Visual Studio Code, you agree to the license terms and privacy statement.

Figure 13-1 Visual Studio Code downloads

Select the version of the program for your computer by clicking the appropriate button on the page. Follow the installation instructions to install Visual Studio Code on your machine.

Install the Python Extension in Visual Studio Code

Once we have Visual Studio Code installed, we next need to add the Python Extension that helps us work with Python programs. Open the Visual Studio Code application and click the Extensions icon indicated by the arrow in **Figure 13-2**.


Welcome — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
Welcome ×
Start
New file
Open folder...
Clone Git repository...
Customize

Tools and languages
Install support for JavaScript, TypeScript, Python, PHP, Docker and...

Install keyboard shortcuts
Install the keyboard shortcuts of Vim, Sublime, Atom and others

FashionShop C:\Users\Rob\OneDrive\Begin to code Python\Part 3 Final\...
samples C:\Users\Rob\OneDrive\Begin to code Python\Part 1 Final\Ch 0...
More... (Ctrl+R)

Color theme
Make the editor and your code look the way you love

Find and run all commands
Rapidly access and search commands from the Command Palette (...

Interface overview
Get a visual overlay highlighting the major components of the UI

Interactive playground
Try essential editor features out in a short walkthrough

Figure 13-2 Extensions selector

Visual Studio Code will now allow you to select extensions, showing you a list of available extensions. The Python environment we want to use should be near the top of the list, but if it's not visible, type **Python** into the search box at the top as shown in **Figure 13-3**. Once you've found the correct extension (you want the one by Don Jayamanne), click the Install button. Now that you have the extension installed, we can start writing some Python.

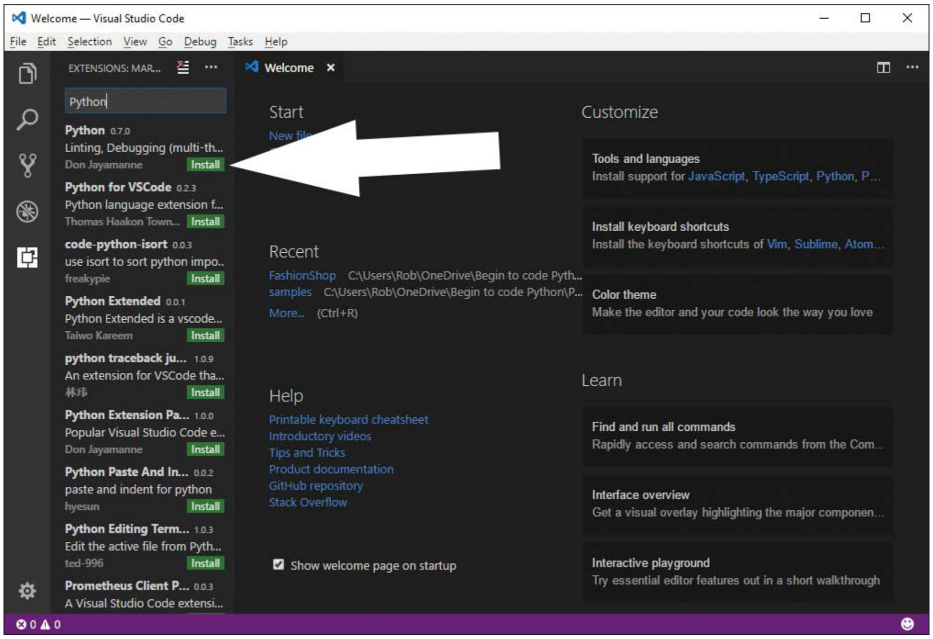

Welcome — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXTENSIONS: MAR...
Python
Python 0.7.0
Linting, Debugging (multi-th...
Don Jayamanne Install
Python for VSCode 0.2.3
Python language extension f...
Thomas Haakon Town... Install
code-python-isort 0.0.3
use isort to sort python impo..
freakypie Install
Python Extended 0.0.1
Python Extended is a vscode...
Taiwo Kareem Install
python traceback ju... 1.0.9
An extension for VSCode tha...
Install
Python Extension Pa... 1.0.0
Popular Visual Studio Code e...
Don Jayamanne Install
Python Paste And In... 0.0.2
paste and indent for python
hyesun Install
Python Editing Term... 1.0.3
Edit the active file from Pyth...
ted-996 Install
Prometheus Client P... 0.0.3
A Visual Studio Code extensi...
Welcome
Start
New file
Recent
FashionShop C:\Users\Rob\OneDrive\Begin to code Pyth...
samples C:\Users\Rob\OneDrive\Begin to code Python\P...
More... (Ctrl+R)
Help
Printable keyboard cheatsheet
Introductory videos
Tips and Tricks
Product documentation
GitHub repository
Stack Overflow
Show welcome page on startup
Customize
Tools and languages
Install support for JavaScript, TypeScript, Python, P...
Install keyboard shortcuts
Install the keyboard shortcuts of Vim, Sublime, Atom...
Color theme
Make the editor and your code look the way you love
Learn
Find and run all commands
Rapidly access and search commands from the Com...
Interface overview
Get a visual overlay highlighting the major componen...
Interactive playground
Try essential editor features out in a short walkthrough
0 0

Figure 13-3 Installing the Python Extension

Create a project folder

Visual Studio Code manages your work in folders. Each folder holds the program files for a specific project. When you're working on a project, you have the project's folder open in Visual Studio Code. To open the folder explorer view in Visual Studio Code, click on the folder icon as shown in **Figure 13-4**.

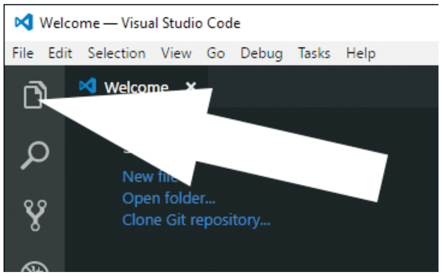

Welcome — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
Welcome
New file
Open folder...
Clone Git repository...

Figure 13-4 Opening the folder explorer

Visual Studio Code will tell you that you presently have no folders open and invites you to open one by clicking the Open Folder button shown in **Figure 13-5**.

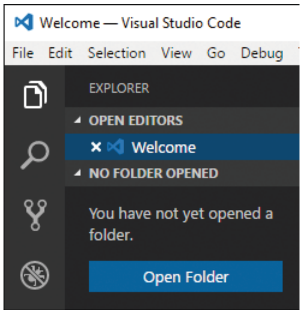

Welcome — Visual Studio Code
File Edit Selection View Go Debug
EXPLORER
OPEN EDITORS
Welcome
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder

Figure 13-5 The Open Folder button

When you click **Open Folder,** a dialog appears that you can use to create or select a folder. Make a new folder called **First Project** and select it. Note that the precise dialog you see at this point will depend on which operating system you're using. Once you've opened the folder, it will appear in the folder explorer in Visual Studio Code, as shown in **Figure 13-6** below.

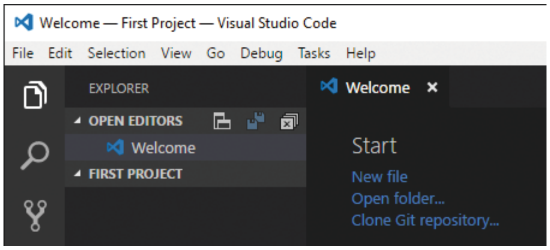


Figure 13-6 First Project in Visual Studio Code

Create a program file

Currently, the folder is empty. Now, let's make a Python program. Rest your mouse cursor over First Project in the Folder Explorer and click the New File icon that appears, as shown in **Figure 13-7**.

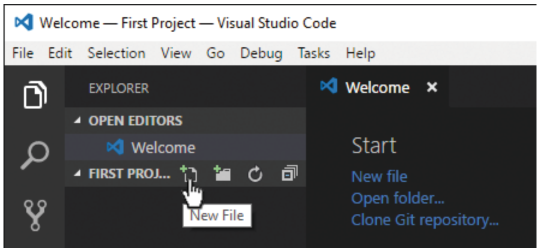


Figure 13-7 New File in Visual Studio Code

Give the new program file the name **myprog.py** and press **Enter**. The file will be created in the folder and opened for editing, as shown in **Figure 13-8**.

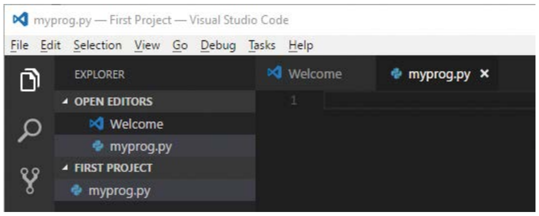

myprog.py — First Project — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXPLORER
Welcome
myprog.py
OPEN EDITORS
Welcome
myprog.py
FIRST PROJECT
myprog.py

Figure 13-8 File Editing in Visual Studio Code

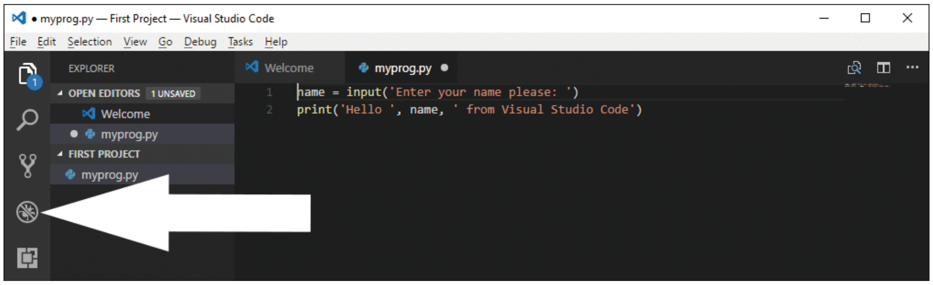
Now, type in the following tiny Python program:

```
name = input('Enter your name please: ')
print('Hello ', name, ' from Visual Studio Code')
```

As you type the program into Visual Studio Code, you'll notice some differences in the way it works compared with the IDLE editor. The editor will suggest Python elements as you type their names. Simply press **Enter** to accept the selected suggestion, or use the arrow keys to scroll down the suggested items to the word you want to select. You will also find help suggestions on functions that you call. If you move the cursor over particular words in the text, you'll find the same words highlighted on the page (which is very useful for seeing where you've used a variable). As you type, you'll also see a tiny map of your program appear in the top right corner of the screen. You can use this map to move rapidly through large files.

Debug a program

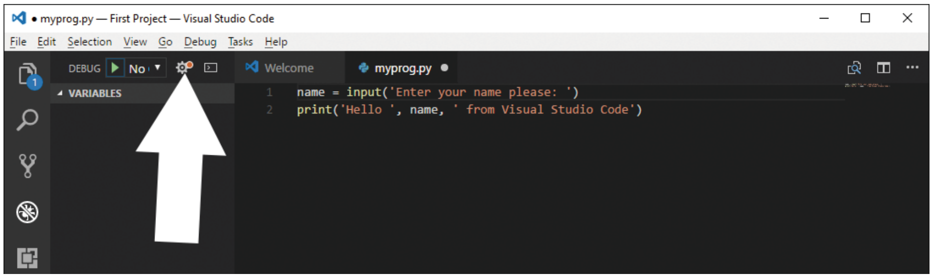
Once you've typed your program, you'll want to run it. Press the **Debug** button on the left-hand menu, as shown in **Figure 13-9** below.


● myprog.py — First Project — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXPLORER
Welcome
myprog.py ●
OPEN EDITORS 1 UNSAVED
Welcome
myprog.py
FIRST PROJECT
myprog.py

```
name = input('Enter your name please: ')
print('Hello ', name, ' from Visual Studio Code')
```

Figure 13-9 Starting Debug in Visual Studio Code

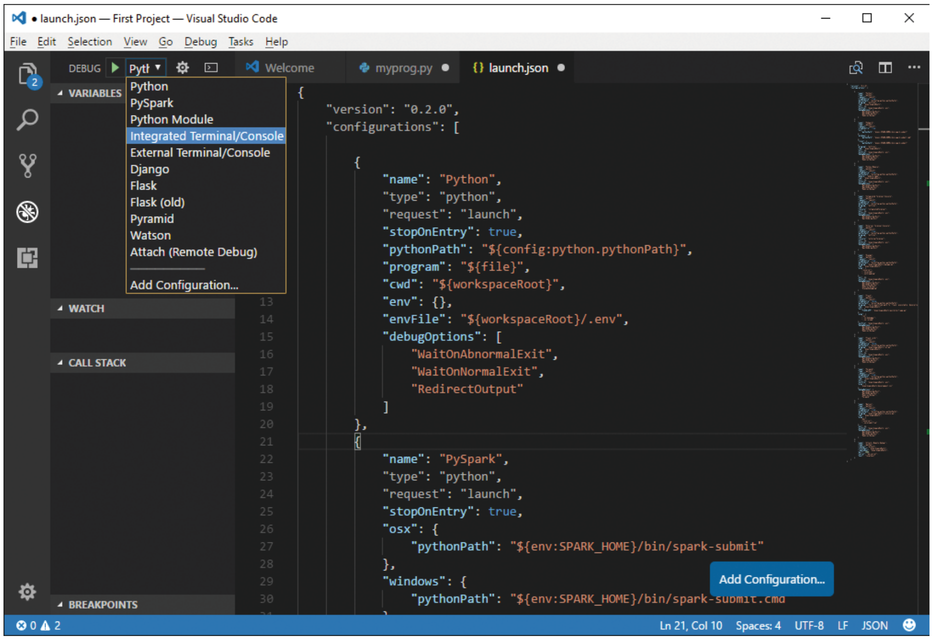
The screen will now change to Debug mode. The Folder Explorer view changes to one that will let us see the contents of variables in the program as we run it. Now we must do some configuration. Visual Studio Code stores some configuration data in each folder with which you work. This data tells the editor the type of program you're working with, and how the editor should behave when working with this project. Currently, we don't have any configuration data set for this folder, so we need to create some. Click on the gears as indicated by the arrow in **Figure 13-10** below to open the configuration file for this folder.


• myprog.py — First Project — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
DEBUG No
VARIABLES
Welcome
myprog.py

```
name = input('Enter your name please: ')
print('Hello ', name, ' from Visual Studio Code')
```

Figure 13-10 Set up Visual Studio Code options

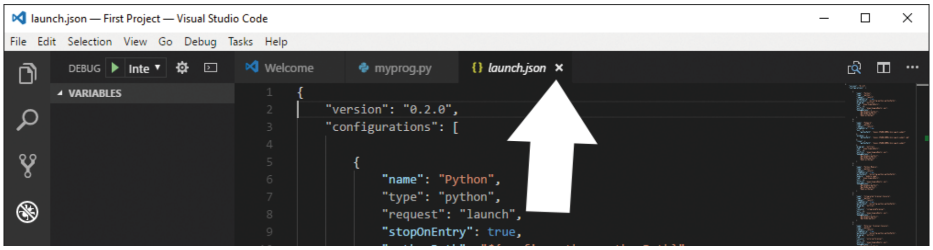
The configuration options are data files in the *JSON* format. We need to select a specific set of configuration options for this folder. Open the pull-down menu you see in **Figure 13-11** and select **Integrated Terminal/Console** from the options that appear.



```
{
    "version": "0.2.0",
    "configurations": [

        {
            "name": "Python",
            "type": "python",
            "request": "launch",
            "stopOnEntry": true,
            "pythonPath": "${config:python.pythonPath}",
            "program": "${file}",
            "cwd": "${workspaceRoot}",
            "env": {},
            "envFile": "${workspaceRoot}/.env",
            "debugOptions": [
                "WaitOnAbnormalExit",
                "WaitOnNormalExit",
                "RedirectOutput"
            ]
        },
        {
            "name": "PySpark",
            "type": "python",
            "request": "launch",
            "stopOnEntry": true,
            "osx": {
                "pythonPath": "${env:SPARK_HOME}/bin/spark-submit"
            },
            "windows": {
                "pythonPath": "${env:SPARK_HOME}/bin/spark-submit.cmd"
```

Figure 13-11 Configure Visual Studio Code options

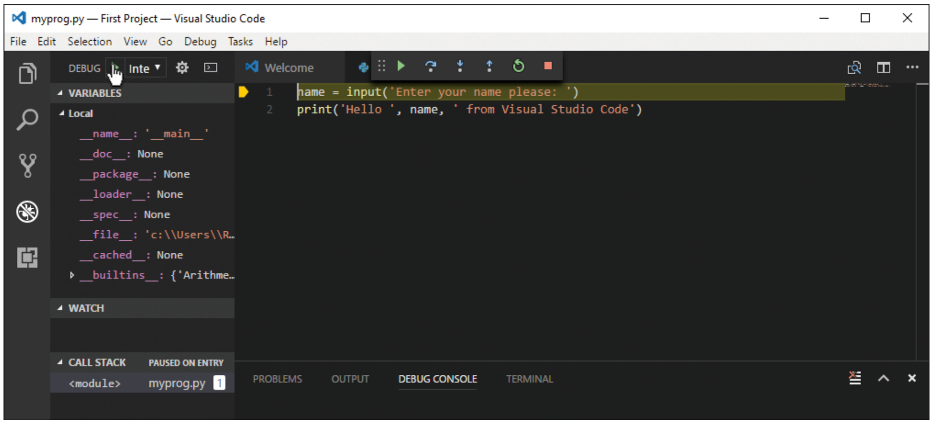
Once you've selected your options, close the configuration file by clicking the X near the file name in the editor, as shown in **Figure 13-12** below.


launch.json — First Project — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
DEBUG
Inte
VARIABLES
Welcome
myprog.py
launch.json

```
{
    "version": "0.2.0",
    "configurations": [

        {
            "name": "Python",
            "type": "python",
            "request": "launch",
            "stopOnEntry": true,
```

Figure 13-12 Close the configuration file

Now we can debug our program. Click the green arrow you see next to Debug in Figure 13-11 to start the Debug program. The program window now shows the statement about to be performed, as you can see in **Figure 13-13**.


myprog.py — First Project — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
DEBUG
Inte
Welcome
VARIABLES
WATCH
CALL STACK
PAUSED ON ENTRY
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL

```
name = input('Enter your name please: ')
print('Hello ', name, ' from Visual Studio Code')
```

```
Local
  __name__: '__main__'
  __doc__: None
  __package__: None
  __loader__: None
  __spec__: None
  __file__: 'c:\\Users\\R…
  __cached__: None
  __builtins__: {'Arithme…
```

```
<module>    myprog.py 1
```

Figure 13-13 Visual Studio Code debugging

At the left, you see all the local variables. Currently, these are the variables set by Python for a program. The debugger allows you to watch the contents of variables, which are also shown on the left side. There is also a panel called the Call Stack that shows you the functions and methods that have been called. At the top of the screen are controls that let you debug the program, statement by statement, as shown in **Figure 13-14**.

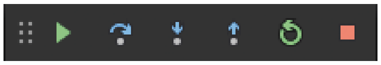
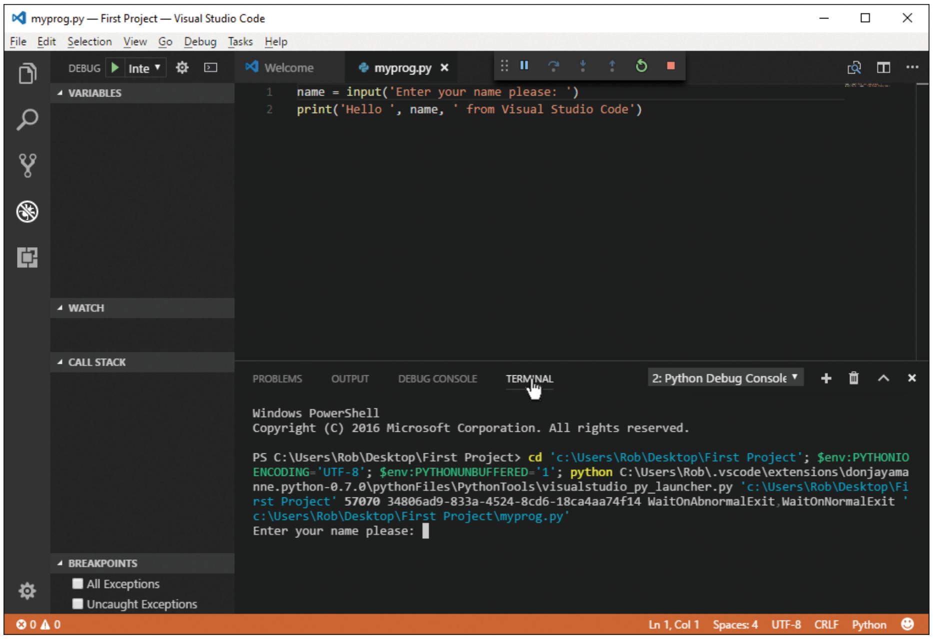


Figure 13-14 Run controls

From left to right, the controls are:

- **Move Panel** (six dots): Click and drag this to move the panel around.
- **Run/Continue** (green triangle): Run the program or continue from a breakpoint.
- **Step over** (Curved arrow over dot): Execute the present statement. If the statement is a method or function call, don't go into the method or function, just obey it. This is called "stepping over" a method or function.
- **Step into** (Arrow pointing down): Execute the present statement. If the statement is a method or function call, enter (or "step into") the method or function and start to step through it.
- **Step out** (Arrow pointing up): Complete the present method or function and "step out" of it.
- **Restart** (Counterclockwise arrow): Restart the program from the beginning.
- **Stop** (Square): Stop the program.

Press the Step Over control (the curved arrow over a dot), which will cause Visual Studio to perform the statement that calls the input function to read our name. The program uses the Terminal window at the bottom of the screen, so click that to open it, allowing you to enter your name, as shown in **Figure 13-15** below.



```python
name = input('Enter your name please: ')
print('Hello ', name, ' from Visual Studio Code')
```

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\Rob\Desktop\First Project> cd 'c:\Users\Rob\Desktop\First Project'; $env:PYTHONIO
ENCODING='UTF-8'; $env:PYTHONUNBUFFERED='1'; python C:\Users\Rob\.vscode\extensions\donjayama
nne.python-0.7.0\pythonFiles\PythonTools\visualstudio_py_launcher.py 'c:\Users\Rob\Desktop\Fi
rst Project' 57070 34806ad9-833a-4524-8cd6-18ca4aa74f14 WaitOnAbnormalExit,WaitOnNormalExit '
c:\Users\Rob\Desktop\First Project\myprog.py'
Enter your name please:
```

Figure 13-15 Entering your name

After you've entered your name, you are returned to the debugger. If you look at the Variables display on the left side of the screen, you'll find that a new variable, `name`, has been added to the list. If you press the **Run** button in the control panel, you'll see the program run, and Visual Studio Code will say hello to you.

You might think that we've done a lot of work only to slightly improve our working conditions. However, when you start typing your programs, you'll find that this text editor is a huge improvement over the one in IDLE. Visual Studio Code is very good at suggesting things based on what you're typing. If you type a variable name in one part of the program, you'll find that the name is suggested the next time you start to type it. There's a lot to explore in the commands, too. The editor is very good at various kinds of searching and replacing. It's very easy to set breakpoints in your program. Just click to the left of the statement at which you want your program to pause. There's no need to open a special debugging window, as with IDLE.

Visual Studio Code is extremely powerful and customizable, and there are lots more useful extensions you can add. You can also integrate many Python tools that can be used to check and test your program.

WHAT COULD GO WRONG

Selecting the right Python interpreter

Obviously, you must have Python installed before you try using the Python extension in Visual Studio Code. However, you might have a problem if your system has multiple versions of Python installed. You might have installed Python version 3.6 to work through the examples in this book, but your computer might also have Python 2.7 installed. When the Python extension for Visual Studio Code is installed, it might pick the wrong version of the program.

You can use the Command Palette in Visual Studio Code to select the command you need to fix this. Open the Command Palette from the View menu. Then type in:

```
Python:Select Workspace Interpreter
```

You won't need to type all the text, as the palette will find matching commands from which you can choose. Select the command "Python: Select Workspace Interpreter" and then pick the Python interpreter with version 3.6.

You can also use this command to select the interpreter to use if you installed Visual Studio Code before you installed Python on your machine.

Other Python editors

Visual Studio Code is my “weapon of choice” for writing Python. However, here are a couple other development tools you might like to check out.

Visual Studio 2017 Community Edition

Visual Studio is a heavyweight development tool that’s very popular in the software development industry. It’s available on both Windows and Mac platforms, but unfortunately, at the time of writing, only the Windows version of Visual Studio supports Python development. If you have a Windows PC, I strongly suggest that you look into Visual Studio. The Community Edition is a free download and is extremely powerful. You can find it at www.visualstudio.com.

Pycharm

Pycharm is not without its charms. It provides a nice place to work, and the Community Edition is a free download from www.jetbrains.com/pycharm.

Create a Graphical User Interface with Tkinter

The mainstay of our interactions with our programs has been the input and print functions provided with Python, along with the BTCInput module that we created to read numbers and text. Now we’ll find out how to use Python to create a Graphical User Interface (GUI). You should already be very familiar with GUIs, as most modern applications are controlled in this way.

A user interface is what people see when they use your program. A graphical user interface displays buttons, text fields, labels, and pictures that the user works with to get their job done. Part of the job of the programmer is to create this “front end” and then put the appropriate behaviors behind the screen that allow the user to drive the program and get what they want from it. In this section, we’ll find out how to create a program that uses a graphical user interface.

It should come as no surprise that a graphical user interface on the screen is represented by objects. When a program is working with items on the screen, it is calling methods in the object. For example, if we want to change the text displayed by a label on the screen, we would call a method on the object that is responsible for that label and tell it to change the text.

We'll use a module called Tkinter, which is shipped as part of the standard Python distribution and contains lots of different kinds of objects that represent the objects on the screen. It's also a very good example of a class hierarchy, in that particular display items (for example, buttons, blocks of text, and images) are represented by classes that are subclasses of parent items. Tkinter is actually a Python interface to a Graphical User Interface toolkit called Tk. Tk is available for many different hardware platforms including Windows, Mac, and Linux devices.

MAKE SOMETHING HAPPEN

Build our first user interface

The best way to find out about Tkinter is to play with it. So, let's do that. We can do so from the Python Command Shell in IDLE. So, let's start that up. The first thing we need to do is import all the resources from the Tkinter module. Give the following command and press **Enter**:

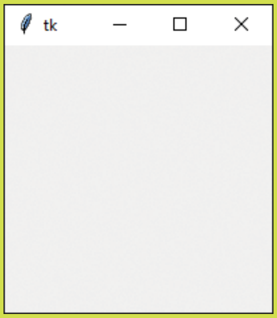
```
>>> from tkinter import *
```

This form of input is different from others we've used recently. It's a way of using the items in a module without having to put the module name in front of each item. You can find more discussion about this in Chapter 7 in the section "Convert our functions into a Python module."

Now that we've imported the module, we can use it. The first thing we'll do is create a "root" window, which will act as a container for all the elements on our display. Type the statement below and press **Enter**:

```
>>> root = Tk()
```

The statement creates a new window on the screen and sets the variable `root` to refer to the window. You should notice that a new window has appeared on your desktop. It should look like the one below.



Let's create a new Label and add it to the window. Label items are used to display text in a window. The user can't interact with a Label, but a program can change the text on the label to display results on the screen. Type in the following statement and press **Enter**.

```
>>> hello = Label(root, text='hello')
```

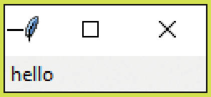
The initializer for a Label takes two parameters. The first is the *parent* display object, which is the object within which the Label will be displayed. You can put objects inside objects so that you can build up complex displays. We won't do that just yet; instead, we'll display the Label in the main window so we can pass in the value of root. The second parameter we're giving to the initializer is a keyword argument called text, which is the text that we want the Label to display.

If you look at the window on the screen, you'll notice, perhaps to your disappointment, that the label has not appeared. This is because the graphical user interface doesn't put anything on the screen until it knows where to put it.

There are two ways you can position things within a display. You can use a mechanism called pack, which, as the name implies, packs the elements together in the window. You can give pack hints such as "LEFT" or "TOP" to tell it to put the item in that part of the display. However, I suggest that you use a mechanism called grid. This lets you lay out your items in a grid. This means that you'll need to plan your screen layout before you write the program, but a bit of planning is never a bad thing in my experience. We tell our label to use the grid layout method by calling the grid method on the label. Type in the following statement and press **Enter**.

```
>>> hello.grid(row=0,column=0)
```

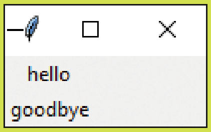
This tells the Label referred to by hello to use the grid layout and to put it at grid location (0,0). This is the top left corner of the screen. If you look at the display window, you should notice two things. First, the label is now displayed. Second, you should see that the window has now been shrunk to fit the label within it.


hello

Let's add another label. Enter the following two statements:

```
>>> goodbye = Label(root, text='goodbye')
>>> goodbye.grid(row=1, column=0)
```

The display will now contain two labels.



The labels seem to be aligned at the left edge of the window. But the hello text is indented slightly, as it is centered about the label portion of the window. We can use some settings to improve this. We can also specify margins around items we display. But that's for later. For now, let's add a button.

Buttons are one way that a user can initiate an action in our programs. The user presses a button when they want something to happen. So, we need a way of linking a button to some code in our application. This turns out to be very easy. We create a function, tell the button the name of the function, and then when the button is clicked the function is called. So, let's make a button function. Type the following text and press **Enter** after each statement, including the empty line after the print statement.

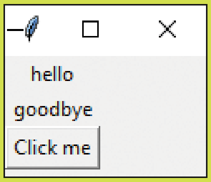
```
>>> def been_clicked():
            print('click')

>>>
```

We now have a function called been_clicked, which we can connect to our button when we create it. Let's do that now. Enter the following statement.

```
>>> btn = Button(root, text='Click me', command=been_clicked)
```

This creates a Button and sets the variable btn to refer to it. The second argument tells the Button to call the been_clicked function when the button is clicked. Now, let's place the button on the display. Enter the following statement to place the button at the bottom of the display.

```
>>> btn.grid(row=2, column=0)
```



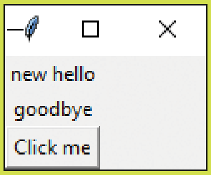
Now, you're really going to have to click the button.

```
>>> btn.grid(row=2, column=0)
>>> click
click
click
```

I clicked the button three times, as you can see above. Each time you click the button, the function been_clicked is called. Functions such as been_clicked are called *event handlers* because they are executed in response to an external event.

Next, we'll change the content of one of the labels on the display. Display elements provide a method called config, which can be used to configure them. We can set the text attribute of the label by using the config method. Type the statement below and press **Enter**.

```
>>> hello.config(text='new hello')
```

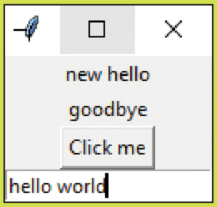


The content of the hello label changes to the new text.

The final thing we'll do is read some text from a display element, which is how we can read things entered by the user. If the user is only entering a single line of text, we can use the Entry component for this. Type in the following statements, pressing **Enter** after each of them.

```
>>> ent = Entry(root)
>>> ent.grid(row=3,column=0)
```

These statements create an Entry item at the bottom of our little program, which is referred to by a variable called ent. I've typed the universal computer greeting hello world into the text entry area, as you can see below. You can type in whatever text you like.



Now that we've managed to enter some text, the next thing to do is to try to read it from our program. We can use the `get` method on our text entry object to do this. Type in the following statement. The `get` method asks an element to return the text it is holding.

```
>>> print(ent.get())
```

When you press Enter, the `get` method runs on the `Entry` object, and it returns the string that was typed in. In my case, it shows `hello world`.

```
>>> print(ent.get())
hello world
```

You can type in some more text and read it again, just to prove that it works.

CODE ANALYSIS

Building a graphical user interface

You might have some questions about the user interface we just created.

Question: What happens if we change the size of the window on the desktop?

Answer: We haven't given the graphical user interface any special instructions about what to do if the size of the window is changed, so if we use the mouse to grab hold of the edges of the window and change its size, we'll find that we can make the window far too big, and we can also make it smaller than the components it is displaying. However, we can set attributes on the window to make it impossible to change its size:

```
root.resizable(width=False, height=False)
```

The `resizable` method on the `root` display element lets us determine how the user can change the size of its window on the screen. You can try it now with the window we created in the previous "Make Something Happen."

We can also make it possible for the user to resize the window and have the size and position of components change automatically.

Question: What happens when I close the window we just created?

Answer: Because we have used the IDLE Command Shell to create the window, the window will disappear when you close it on the desktop. However, when we create a program that creates a graphical user interface, we'll discover a way that our program can get control when the user closes the window.

Question: Will the window look the same on different machines?

Answer: Mostly. Because the Tk graphical toolkit uses the windowing system of the host computer to display its output, you'll find that the window will look like a window on the host machine.

Question: What happens if an event handler function connected to a button takes a long time to complete?

Answer: The function connected to a button will run when the button is pressed. The button will be "stuck down" until the function completes. I actually tested this by creating a version of `been_clicked` that contained a call to the `sleep` function from the time module that made the function pause for ten seconds. When an event handler is running in response to one event, all the other controls on the application will be unresponsive.

You should take care to make sure that event handler functions are completed as quickly as possible. Fortunately, the kind of actions that we'll perform when buttons are pressed are not going to cause a problem because they all complete very quickly.

Python supports a mechanism called *threading*. An application can contain several *threads of execution* that execute simultaneously. Each thread could run a different program. An application could respond to a button press by starting a new program.

Creating and managing threads is beyond the scope of this book, but if you do want to perform an action that will take more than a second or so, you should look at how to use threading to perform the action.

Question: What happens if I place a large amount of text in a label?

Answer: The default behavior (that is, unless we specify otherwise) is for `Label` to expand to fit the text inside it. So, the window in the screen would grow to hold this text.

Question: What happens if I put two items in the same cell in the grid?

Answer: The most recently added item will be drawn in preference to the "older" one. In other words, a new item will "block out" an older one. It's best not to do this.

Question: Can we update the contents of elements on the screen from within an event handler function?

Answer: Absolutely. In fact, this is how applications work.

We now know nearly all we need to know to create applications that use a graphical user interface. The most important thing to remember is that events generated by the user (for example, clicking on buttons) will end up as calls to functions inside our application. In the example program above, the function `been_clicked` will never be called by any code that we write. It will be called by the button when the user clicks the button. If we create an application that contains multiple buttons, we can connect each button to a different event handler. If we have two distinct ways to select a particular action (perhaps from a button or from a menu), we can connect both display elements to the same event handler function.

This form of application creation is a bit like “wiring up” electronic devices. We create a user interface design and then connect each of the user interface components to an event handler function. Note that events can be generated from actions such as mouse movements as well as key presses.

Create a graphical application

Now that we know how to create a graphical user interface, we can make our first application that works this way. This application won’t do much, but it will show us how to create applications that work via a GUI. It’s a simple adding machine. **Figure 13-16** shows what it will look like. The user will type in two numbers, press the Add numbers button, and the result will magically appear underneath the button.

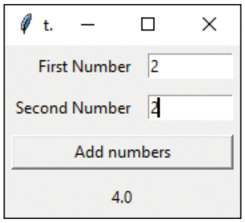


Figure 13-16 Adding machine

This application looks deceptively simple, but there’s quite a lot to learn from building it. Let’s start with the application itself. I’ve created a class called `Adder` that will contain the application. The class will contain a method called `display` that will display the application:

```
class Adder(object):
    '''
    Implements an adding machine using a Tkinter GUI
    Call the method display to initiate the display
    '''
    def display(self):
        '''
        Display the user interface
        Returns when the interface is closed by the user
        '''
```

In the Adder.py source file, I've added some Python code that will run the adding machine if the Adder.py file is executed as a program:

```
if __name__ == '__main__':
    app = Adder()
    app.display()
```

We've seen this arrangement of code before. The file can be opened as a module (for example, by pydoc for producing documentation), but it will only run as a program if it is the main module. Now we must create the contents of the display method that will implement our adding machine.

Lay out a grid

We'll use the grid layout to place the elements in our display area. **Figure 13-17** shows the display with a grid laid over the top to show where each display element will go. The label "Second Number" is at location row=1 and column=0. Some of the items (the Add numbers button and the result value) seem to straddle two columns; we will discover how to do this in the next section.

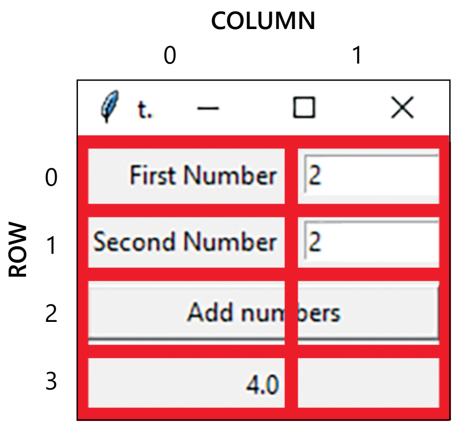


Figure 13-17 Adding machine layout

The items that display “First Number,” “Second Number,” and the result (in this case “4.0”) are all Label elements. We also have a Button to trigger the add numbers behavior and two Entry items to receive the two numbers that are typed in. Let’s start positioning components:

```
first_number_label = Label(root, text='First Number')
first_number_label.grid(sticky=E, padx=5, pady=5, row=0, column=0)
```

These are the statements that create and position the first number label at the top left corner of the display. This is in the element at row=0. There are a few extra arguments to the grid call that we haven’t seen before: sticky, padx, and pady. Let’s look at those.

Use sticky formatting

Quite often, when laying things out in a grid, we’ll find two items of different sizes in the same column. We can see this above, in that the label “Second Number” is slightly longer than the label “First Number”. The layout process will always size a row or column to the largest item in that row or column, which means there will be items placed in grid cells that are larger than they are. By default (that is, unless we state otherwise), an item is placed in the center of a larger cell.

For the first_number_label I want the label to be close to the Entry it is labeling. So, I’ve made the item “sticky” in an “easterly” direction. This means that the label will try to “stick” to an item on its east side. If I wanted to move the label all the way to the left, I’d make it sticky toward the west. If I want the item to “stretch” to fill a cell, I can make it sticky in two directions. This is how I made the “Add numbers” button stretch to fill the entire width of the display area. We’ll see this later in this section. If you’re not sure about compass directions, you can find a handy reference in **Figure 13-18** below.

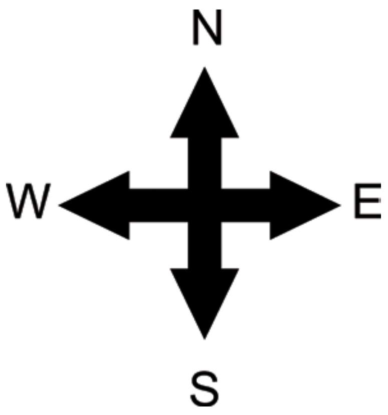


Figure 13-18 Compass points

Use padding

Padding is extra space placed around the component to "pad" it out. Otherwise, the component will be drawn right up to the edge of the cell in which it is being drawn. I like to add around 5 pixels or so of padding around items on the screen. You can specify the amount of padding in the x and y directions. I put 5 pixels around each of the items on the display in both directions.

Span grid cells

The grid must be two elements wide so that I can display the label and the entry boxes for both numbers that the user will enter. However, I'd like the button and the result to be drawn across the full width of the display area. I can do this by merging the cells into which an element is to be drawn. Look at the definition of the `add_button` below.

```
add_button = Button(root, text='Add numbers', command=do_add)
add_button.grid(sticky=E+W, row=2, column=0, columnspan=2,  padx=5, pady=5)
```

The first statement creates the `Button` instance, sets the text to be displayed on the button, and tells the button to call the function `do_add` when the button is clicked. The second statement places the `Button` in the grid in row 2, column 0. However, it also contains the argument `columnspan=2`. This means that the button will be drawn in a cell that spans two columns. This case means that the button will be the full width of the display because the display is two columns wide.

Note also that I've made the button "sticky" in both easterly and westerly directions so that it will be stretched across the entire display when it's drawn. I use the same technique to position the result label. Below are all the statements that position and set up the items on the form.

```
first_number_label = Label(root, text='First Number')
first_number_label.grid(sticky=E, padx=5, pady=5, row=0, column=0)

first_number_entry = Entry(root, width=10)
first_number_entry.grid(padx=5, pady=5, row=0, column=1)

second_number_label = Label(root, text='Second Number')
second_number_label.grid(sticky=E, padx=5, pady=5, row=1, column=0)

second_number_entry = Entry(root, width=10)
```

```
second_number_entry.grid(padx=5, pady=5, row=1, column=1)
add_button = Button(root ,text='Add numbers', command=do_add)
add_button.grid(sticky=E+W,row=2, padx=5, pady=5, column=0, columnspan=2)

result_label = Label(root, text='Result')
result_label.grid(sticky=E+W, padx=5, pady=5, row=3, column=0,  columnspan=2)
```

Create an event handler function

We know that we can connect a function to a button so that when the button is clicked the event handler runs. For the adding machine, the event handler should read the text out of the two `Entry` objects into which the user has (hopefully) entered some numbers. The event handler should then convert the text into numbers, add the two numbers, and then display the result in the result label. Below you can see my version of the event handler for the adding machine.

```
# EG13.01 First Adding machine
class Adder(object):
    '''
    Implements an adding machine using a Tkinter GUI
    Call the method display to initiate the display
    '''
    def display(self):
        # create all the screen elements here
        def do_add():
            first_number_text = first_number_entry.get()
            first_number = float(first_number_text)

            second_number_text = second_number_entry.get()
            second_number = float(second_number_text)

            result = first_number + second_number
            result_label.config(text = str(result))
```

Method called to generate the user interface

Event handler for the add button

Convert the second number text into a floating-point number

Calculate the result

Convert the result into a string and display it

Get the text out of the Entry for second number

Convert the first number text into a floating-point number

Get the text out of the Entry for the first number

CODE ANALYSIS

Writing an event handler

You might have some questions about the event handler we just created.

Question: Why is the event handler defined inside the display function?

Answer: We've seen before that Python will allow programmers to define functions inside other functions. The event handler function needs access to the result label and the two Entry variables that the user uses to enter the two numbers to be added. Code running inside a function has access to the variables in the enclosing namespace (we saw this in Chapter 7), and so the do_add function can use variables declared in the display function.

I could have created do_add as a method in the Adder class (event handlers can be methods as well as functions), but then I would've had to make all the display elements attributes of the Adder class so that the do_add method could access them, which would have meant a bit more typing.

Question: What happens if the user doesn't type in a valid number before pressing the Add numbers button?

Answer: Good question. The answer is that the float function in the do_add event handler will be unable to convert the text in the Entry into a floating-point number. It will fail by raising an exception that will cause the do_add method to be abandoned when the exception is raised. You might notice the exception being raised if you're debugging the program, but when the program is running the user will not see any errors at all.

Entering invalid numbers will not stop the program from running, but the user will not see an error; they'll just notice that the result display will not be updated. In an upcoming section, we'll discover how a program can display a pop-up warning if the user does something like this.

Create a mainloop

When we created a display window from the Python Command Shell in IDLE, we found that it just worked. This is because the shell was running. If we had exited the shell program, we'd have found that the display window disappears as well. If a program just created the display components and then ended, we'd find that the interface would flash up on the screen for a fraction of a second and then vanish when the program ended. To keep the display active, the Tkinter module provides a method called mainloop which a program should call once it has set up its display components:

```
root.mainloop()
```

The name mainloop describes what this method does. It repeatedly fetches events and sends them on to functions that have been created to deal with the events. When the user closes the window on the screen (in Windows 10, they would click the X in the top right corner of the window), the mainloop method ends. As the mainloop method is frequently the last method call in a program that uses a graphical user interface when mainloop ends the program probably ends, too.

You can find a complete implementation of the adder program in the example file **EG13-01 First Adding machine** in the downloadable sample files for this chapter.

Handle errors in a graphical user interface

The adding machine we've created works quite well. However, it does have problems if the user enters invalid text rather than numbers. **Figure 13-19** shows what can happen. The user has typed in two text strings. When they press the Add numbers button, the result display is not updated because the do_add event handler fails with an exception.

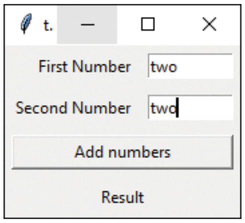


Figure 13-19 Text as numbers

One way to deal with this would be to catch the exceptions and change the result string to reflect the issue. We saw how to catch exceptions in Chapter 6. The code below is part of an improved do_add function that catches exceptions if either of the number conversions performed by the float method fail.

```
# EG13-02 Exception handler with messages

def do_add():
    first_number_text = first_number_entry.get()
    try:
        first_number = float(first_number_text)
    except ValueError:
        result_label.config(text='Invalid first number')
        return

    second_number_text = second_number_entry.get()
```

Get the number text from the Entry on the screen

Start of exception handler

Statement that might throw an exception

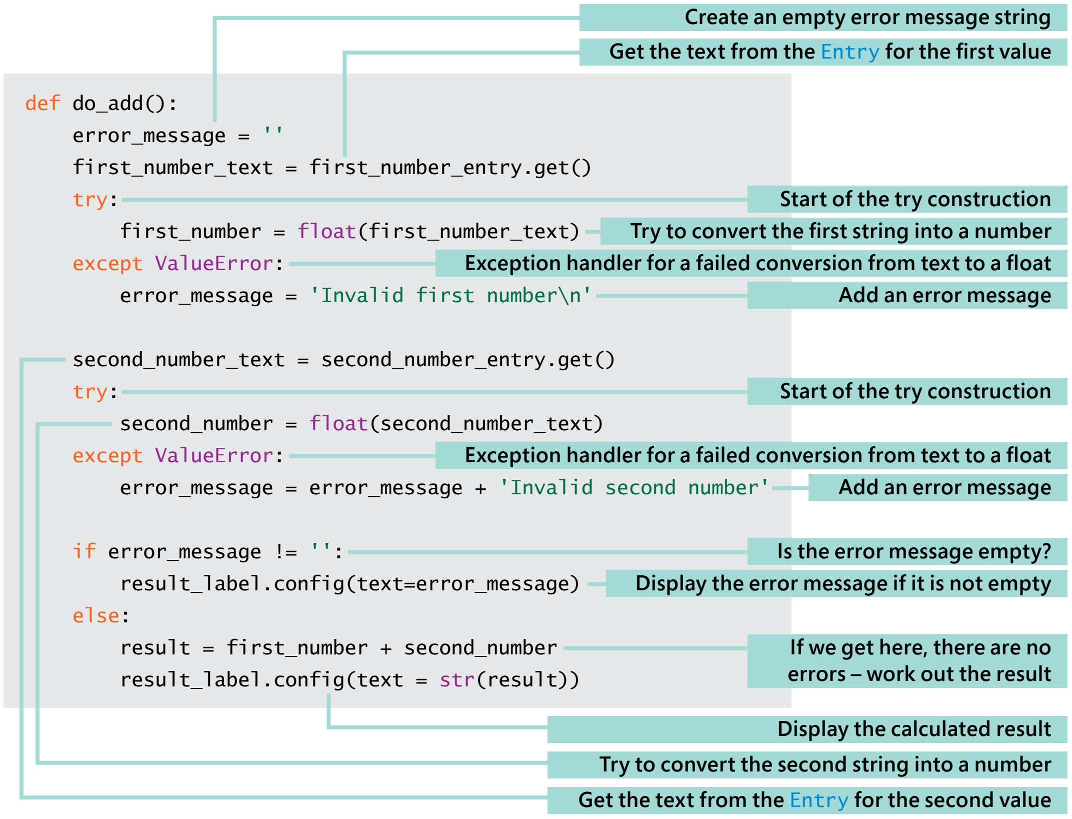
Handler for ValueError exceptions

Set the result to indicate that an error has occurred

Return from the method if the first number is not valid

```
    try:
        second_number = float(second_number_text)
    except ValueError:
        result_label.config(text='Invalid second number')
        return
```

The code above works well, but it is not perfect. If the user enters invalid text in both Entry objects on the screen, the program will only tell the user about the first error, not the second. We can improve this code so that it builds an error string and displays it if any errors are detected:



```
def do_add():
    error_message = ''
    first_number_text = first_number_entry.get()
    try:
        first_number = float(first_number_text)
    except ValueError:
        error_message = 'Invalid first number\n'

    second_number_text = second_number_entry.get()
    try:
        second_number = float(second_number_text)
    except ValueError:
        error_message = error_message + 'Invalid second number'

    if error_message != '':
        result_label.config(text=error_message)
    else:
        result = first_number + second_number
        result_label.config(text = str(result))
```

Create an empty error message string
Get the text from the Entry for the first value
Start of the try construction
Try to convert the first string into a number
Exception handler for a failed conversion from text to a float
Add an error message
Start of the try construction
Exception handler for a failed conversion from text to a float
Add an error message
Is the error message empty?
Display the error message if it is not empty
If we get here, there are no errors – work out the result
Display the calculated result
Try to convert the second string into a number
Get the text from the Entry for the second value

This version of the do_add function is much better. It uses a technique I've used many times when dealing with user errors. It starts with an empty error string. Each time code in the function finds something wrong, it will add text describing the error to the error string. If at the end of the function, the error string is empty, it means that there are no errors and the function can complete. Otherwise, it displays the error message. You can find this version of the error handler in the sample file **EG13-03 Adder with sensible messages**.

There is considerable scope for making this method even better. Tkinter provides methods to set the foreground and background colors of items on the screen, so you could make the do_add function indicate invalid user entries by changing the background color of invalid entries to red. The statement below shows how you can change the color of an item in the program. It configures first_number_entry so that the background of the Entry on the screen is red and the foreground (the color of the text in the entry box) is blue.

```
first_number_entry.config(background='red', foreground='blue')
```

Display a message box

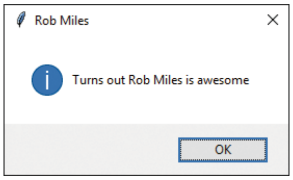
Another way to inform the user of an error is to pop up a message box. This technique has the advantage that the user must see and acknowledge the error before they can continue. Tkinter provides a message box, and it's very easy to use. The first thing the program must do is import the messagebox module:

```
from tkinter import messagebox
```

The messagebox module contains three functions that can display messages: showinfo, showwarning, and showerror. All the message boxes have the same format, but a different icon is used for each. The user interface for the program displaying the message (in our case, the Adder program) will be locked until the user clears the error message by clicking OK or closing the message box. Each of the message functions accepts two arguments, a title and a message. Both are strings. Below we can see how we could use the showinfo function to show some information:

```
messagebox.showinfo('Rob Miles', 'Turns out Rob Miles is awesome')
```

Figure 13-20 shows the output of this important message. To display a warning, use the showwarning method. To display an error, use the showerror method. You can find a version of the Adder program that displays a message box to indicate user error in the sample file **EG13-04 Adder with message box**


Rob Miles

Turns out Rob Miles is awesome

Figure 13-20 Important message from showinfo

We can make a version of the Adder program that displays a message box by replacing the statement that sets the result label to the error with one that generates a message box. The code below shows the part of do_add that handles errors. If the error message is not empty (in other words, something bad has happened), the message box will be displayed to indicate this.

```
if error_message != '':
    messagebox.showerror(title='Adder',message=error_message)
```

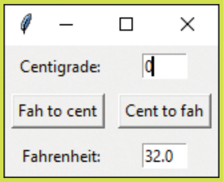
Is the error message empty?

Display an error message box

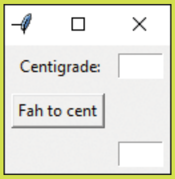
MAKE SOMETHING HAPPEN

Fahrenheit to centigrade. And back.

In this challenge, I'll give you a half-finished program to complete. This is something that happens surprisingly frequently in the software industry. When you get your first programming job, it's likely that you'll start by modifying an existing program rather than being asked to create an all-new program. The program you're working on is the ultimate temperature converter. The user can convert from Fahrenheit to centigrade or back. They type their conversion value into one box and, depending on which button they press, the other box will show the converted value. The program is supposed to look like this:



Unfortunately, the programmer hired to create the program has taken his job much too seriously, and has gone away to Hawaii, supposedly to test the program in higher temperatures. He has left behind a program that looks like this:



```python
'''
Display a graphical user interface that lets users convert from temperature scales
'''

from tkinter import *

class Converter(object):
    '''
    Displays a Tkinter user interface to convert between Fahrenheit and centigrade
    Call the display function to display the converter on the screen
    '''

    def display(self):
        '''
        Displays the converter window
        When the window is closed, this method completes
        '''

        root = Tk()

        cent_label = Label(root, text='Centigrade:')
        cent_label.grid(row=0, column=0, padx=5, pady=5, stick=E)

        cent_entry = Entry(root, width=5)
        cent_entry.grid(row=0, column=1, padx=5, pady=5)

        fah_entry = Entry(root, width=5)
        fah_entry.grid(row=2, column=1, padx=5, pady=5)

        def fah_to_cent():
            '''
            Convert from Fahrenheit to centigrade and display the result
            '''
            fah_string = fah_entry.get()
            fah_float = float(fah_string)
            result = (fah_float - 32) / 1.89
            cent_entry.delete(0, END) # remove the old text
            cent_entry.insert(0, str(result)) # insert the new text

        def cent_to_fah():
            '''
            Convert from centigrade to Fahrenheit and display the result
            '''
            cent_string = cent_entry.get()
            cent_float = float(cent_string)
            result = cent_float * 1.8 + 32

        fah_to_cent_button = Button(root, text='Fah to cent', command=fah_to_cent)
        fah_to_cent_button.grid(row=1, column=0, padx=5, pady=5)

        root.mainloop()



if __name__ == '__main__':
    app = Converter()
    app.display()
```

The programmer has used a feature of Tkinter that we haven't seen before. When the program has calculated a new result, it must display it in a text entry field. Updating the text in an Entry is slightly more complicated than just changing the text in a Label. There are very powerful editing features available, but we just want to replace the text with new text. The two statements below show how this is done. The first statement deletes all the text from cent_entry. The first argument to the delete method is the position to start deleting (0 means the beginning of the string). The second argument to the delete method is the position to stop deleting. The variable END is declared in the Tkinter module and means "the end of the line."

The second statement inserts a string containing the result into cent_entry starting at the location 0 (the beginning of the string).

```
cent_entry.delete(0, END) # remove the old text
cent_entry.insert(0, str(result)) # insert the new text
```

You can find the starter code in the folder **EG13-05 TemperatureConverter Starter** in the sample code for this chapter. The folder is all set up for use with Visual Studio Code. If you want to "skip to the end," you can find a complete version of the program in the folder **EG13-06 TemperatureConverter Complete**. However, even the complete version could use some attention; currently, it doesn't handle invalid inputs.

You can use this program as the basis for any conversion you like, such as ounces to grams, mph to kph, or dollars to euros.

Draw on a Canvas

We can also use graphical interfaces to allow the user to draw with the mouse on the screen. We do this by creating a drawing area that sends our program an event each time the user moves their mouse. If this event performs a drawing operation, we have an instant drawing program. Let's look at how we can get events from areas of the screen.

MAKE SOMETHING HAPPEN

Investigate events and drawing

We can investigate Tkinter events from the Python Command Shell in IDLE. So, let's start that up. As before, the first thing we need to do is import all the resources from the Tkinter module. Give the following command and press **Enter**:

```
>>> from tkinter import *
```

Next, we need to create a window on the screen. Enter the following statement and press **Enter**:

```
>>> root = Tk()
```

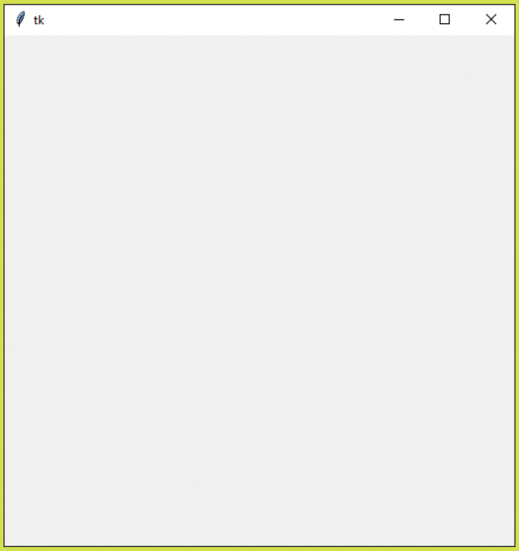
Now we'll create a Canvas. A Canvas is a display component that can act as a container for lots of other display elements. We can draw and position these elements inside the canvas. When you create a Canvas, you can tell the graphical user interface the size of the Canvas in pixels. Enter the following statement to create a Canvas that is 500 pixels square.

```
>>> c = Canvas(root, width=500, height=500)
```

To get the Canvas displayed, we need to specify where to place it. It will be the only item on the display, so we can place it at row 0 and column 0.

```
>>> c.grid(row=0, column=0)
```

If you look at the window that's been created, you should see that the program is now displaying a square window.



Now we need to connect a function to the events that Tkinter generates when a mouse is moved over the Canvas. Let's write the function first. Enter the following function. Enter a blank line after the print statement to end the function.

```
>>> def mouse_move(event):
        print(event.x,event.y)

>>>
```

The function is supplied with a single parameter, which is a reference to an event object. This object has two attributes, which are the x and y positions of the mouse pointer at the time the event occurred. The method above just prints these positions on the screen.

Now we need to connect this function to the event generated when the mouse is moved with a button pressed. This will give us the movement detection that will make our drawing program work, which is called *binding* the function to the event. Objects on the display provide a bind method that programs can use to connect functions to events. Each event has a unique name. Type in the following statement and press **Enter**. The statement calls the bind method on the canvas and links the <B1-Motion> event (that is, mouse motion with button 1 pressed down) to the function mouse_move. Each time the Canvas detects a mouse movement with the button pressed, it will call the method.

The bind method returns a string that describes the binding that has taken place. A program could use this string to identify the binding and disconnect the connection later, but we can ignore this string for now. Note that because the description string is supposed to be unique on a specific machine, you may find that the string displayed on your machine differs from the one shown below.

```
>>> c.bind('<B1-Motion>', mouse_move)
'2886099647752mouse_move'
>>>
```

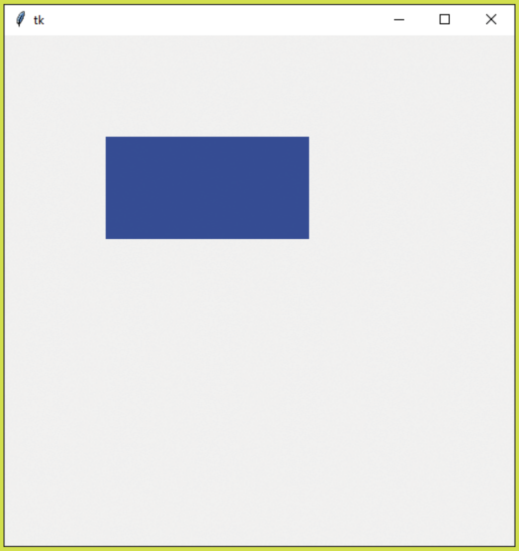
Now for the fun bit. Move your mouse to the window displaying the canvas, hold down the left (or only) button on the mouse and drag it. Watch the Python Command Shell in IDLE. You should see a stream of numbers being generated. Below you can see some of the numbers that I saw. If you drag the mouse up to the top left corner of the canvas, you should see the numbers getting smaller. This is because the origin of the coordinates (the point 0,0) is the top left corner of the canvas. This should not come as a surprise; it is the same way that the grids are numbered.

```
>>> 283 277
290 297
290 306
289 307
```

Printing coordinates is nice enough, but we are making a drawing program, and we need to draw a dot. The Canvas object provides a method called create_rectangle that should do the trick. Tear yourself away from dragging the mouse around your canvas and enter the following statement. This will draw a blue rectangle. The top left corner of the rectangle will be at coordinate (100,100). The bottom right corner of the rectangle will be at coordinate (300,200). The outline argument sets the color of the outline of the rectangle; the fill argument sets the color used to fill in the block. Unless you specify otherwise, the outline color will be black.

```
>>> c.create_rectangle(100,100,300,200,outline='blue',fill='blue')
1
>>>
```

If you look at the output window for your program, you should see that a blue rectangle has duly appeared.



You may be wondering why the value 1 was displayed when we created the blue rectangle. This is because when you create an object on a canvas, the method that creates it will return a value that identifies this object. If we just call a method, Python will just display the value returned by it, which in this case was 1 because we have just created object number 1 on the `Canvas`.

A canvas manages each object by its number. We can ask the canvas to remove an object from the display by using the `delete` method. Type the following command and press **Enter**.

```
>>> c.delete(1)
```

You should see the blue rectangle disappear. This is a very powerful feature of the `Canvas`. Every single element on the screen is a separate object that we can find and manipulate after we've drawn it.

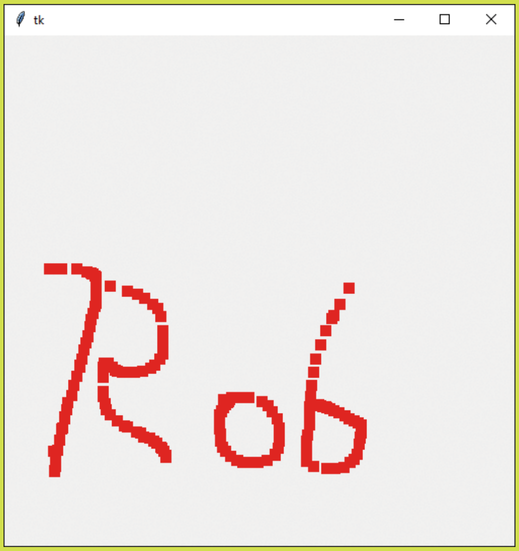
Now we need to use the drawing method to allow us to draw with the mouse. We can create a new function that draws a block at the position a mouse event was detected. Enter the statements below. Add an empty line after the call of `create_rectangle` to end the function definition.

```
>>> def mouse_move_draw(event):
            c.create_rectangle(event.x-5,event.y-5,event.x+5,event.y+5,
              fill='red', outline='red')

>>>
```

This method creates two points that define the rectangle to be drawn. The first point is five pixels to the left and above the mouse position, the second point is five pixels to the right and below the mouse position. The result is that the function will draw a ten-pixel square block centered around the position of the mouse. Now, all we need to do is bind this new draw function to the event generated when the mouse is moved with the pointer held down.

```
>>> c.bind('<B1-Motion>', mouse_move_draw)
'2886099651528mouse_move_draw'
```

Once you have bound the function to the event, you should be able to draw on the canvas by clicking the left mouse button and dragging it over the canvas.



Above, you can see my not very artistic attempts at drawing. You can almost certainly do better. You should also notice that the program no longer prints the mouse position in the IDLE output window, which is because only one function can be bound to a particular event.

Tkinter events

Tkinter events are very powerful and flexible. Let's look at the event we've been using for drawing. Below is the statement we used to link the `mouse_move` function to the event where the mouse is moved with a button held down.

```
c.bind('<B1-Motion>', mouse_move)
```

The event identifier is the string `'<B1-Motion>'`. We can break this string down into two components. The first part is called the *modifier*. You can think of this as a condition that must be satisfied for the event to be generated. In our case, the condition is that mouse button 1 is pressed. The second part is called the *detail*. This is the thing that will produce the events. If we left the modifier off, and just bound a handler to an event identified by the string `'<Motion>'` we would get events produced every time the mouse was moved, which is more events than we really want. Here are a few of the most useful events and modifiers:

MODIFIER	ACTION	DETAIL	ACTION
Control	Control key pressed	Motion	Mouse moved
Shift	Shift key pressed	ButtonPress	Mouse button pressed
B1 – B4	Corresponding mouse button pressed	ButtonRelease	Mouse button released
		KeyPress	Key pressed
		KeyRelease	Key released
		MouseWheel	Mouse wheel moved

Note that the different actions may deliver different event information when their action is called. In other words, the events delivered when a key is pressed contain the key information, rather than mouse coordinates. You can create more complex events if you wish with multiple modifiers.

Create a drawing program

We can use events to create a simple drawing program. The user can draw with the mouse and select colors with the keyboard. They can also clear the canvas and start a new drawing.

```
'''
Provides a simple drawing app
Hold down the left button to draw
Provides some single key commands:
R-red G-green B-blue
C-clear
'''

from tkinter import *
```

```
class Drawing(object):

    def display(self):
        root = Tk()                                                   Create the display root

        canvas = Canvas(root, width=500, height=500)                  Create the canvas to draw on
        canvas.grid(row=0, column=0)                                  Position the canvas in the display

        draw_color = 'red'                                            Set the draw color to red

        def mouse_move(event):                                        Event handler for the mouse movement
            '''
            Draws a 10-pixel rectangle centered about the mouse
            position
            '''
            canvas.create_rectangle(event.x-5, event.y-5,
            event.x+5, event.y+5, fill=draw_color, outline=draw_color)

        canvas.bind('<B1-Motion>', mouse_move)                        Bind the event handler to the mouse
                                                                      movement

        def key_press(event):
            nonlocal draw_color       Make sure we use the draw_color in the enclosing namespace
            ch = event.char.upper()   Get the character pressed and convert it into uppercase
            if ch == 'C':                                             Is the character a C?
                canvas.delete('all')                                  Delete all the objects on the canvas
            elif ch == 'R':                                           Is the character an R?
                draw_color = 'red'                                    Set the draw color to red
            elif ch == 'G':                                           Is the character a G?
                draw_color = 'green'                                  Set the draw color to green
            elif ch == 'B':                                           Is the character a B?
                draw_color = 'blue'                                   Set the draw color to blue

        canvas.bind('<KeyPress>', key_press)                          Bind the event handler for keypresses
        canvas.focus_set()                                            Set the keyboard focus to the canvas

        root.mainloop()                                               Main loop for Tkinter

if __name__ == '__main__':                                            Are we being run as a program?
    app = Drawing()                                                   Create a drawing instance
    app.display()                                                     Start the display on the drawing
```

CODE ANALYSIS

Drawing on a canvas

In the above program, I've used some features of Python that you haven't seen before. You might have some questions about the program.

Question: What is the draw_color variable used for?

Answer: As its name implies, the draw_color variable holds the color to be used for draw actions. The Tkinter system can recognize a large range of colors by name. You can find a chart giving all the available colors here: http://wiki.tcl.tk/37701.

If you want to specify your own colors, you can do so by giving a string that contains three two-digit hexadecimal values, one each for the amount of red, green, and blue, respectively.

```
draw_color = '#FFFF00'
```

This would set the draw color to yellow (all the red, all the green and none of the blue).

In the program, the draw color is set to red when the program starts and then changes when the user presses the R, G, or B keys.

Question: How do you clear the canvas?

Answer: We saw above that we can delete items we've drawn if we know their ID. The drawing program above doesn't store the ID values of the items it draws (although it could). The delete method can be given with the argument 'all' if you want your program to delete everything that's been drawn. This has the effect of clearing the display.

```
canvas.delete('all')
```

The statement above is obeyed when the user presses C.

Question: In the key_press function, you've created a "nonlocal" variable called draw_color.

```
def key_press(event):
    nonlocal draw_color
```

What does this mean?

Answer: The key_press function needs to be able to change the value of the draw_color variable when the user presses a key to select a different drawing color. The variable draw_color is declared in the function that contains the key_press function. In Chapter 7, in the section "Global variables in Python programs," we saw how a function could access variables that were not created within the function by telling Python that the variable is "global." However, the variable draw_color is not global (global variables are declared outside any function); it just isn't local to the key_press function. The nonlocal statement is used in this situation. In other words, saying that a variable is nonlocal means "I'd like to use the variable with this name from an enclosing namespace please."

Question: What does the call of focus_set do?

Answer: When you move the mouse pointer over a specific item on the screen, Python knows that the item is the one that should receive any motion events. However, when the user presses a key on the keyboard, Python has no way of knowing which component in the application is supposed to receive a message.

The focus_set method lets a component say, "Please give me all the keyboard events." Note that this action is independent of what the user is doing. The user may have selected (given focus to) the window containing your Python program, but keyboard events will only be passed to a component if it has acquired focus using this method.

MAKE SOMETHING HAPPEN

Make the drawing program draw ovals

In this development challenge, you'll have to do some detective work to find out how some of the Tkinter functions work. The Canvas object provides a method called create_oval, which can be used to draw ovals. It has a different set of arguments from the create_rectangle method. Find out what the arguments are and make a version of the drawing program you can find in the sample folder **EG13-07 Drawing program** that draws ovals. You could even allow the artist to swap between brushes by pressing S for a square brush and O for an oval brush.

Enter multi-line text

We've seen that you can use a Tkinter Entry object to allow the user to enter a single line of text into the user interface, but this would not be useable if we wanted to create a text editor. The Tkinter framework provides an object called Text that allows a user to enter pages of text. It works in a very similar way to the Entry object, but there are some differences.

MAKE SOMETHING HAPPEN

Investigate the Text object

We can investigate the Text object from the Python Command Shell in IDLE. So, let's start that up. As usual, the first thing we need to do is import all the resources from the Tkinter module. Give the following command and press **Enter**:

```
>>> from tkinter import *
```

Next, we need to create a Tkinter window on the screen. Enter the statement below to create a new window and set the variable root to refer to it.

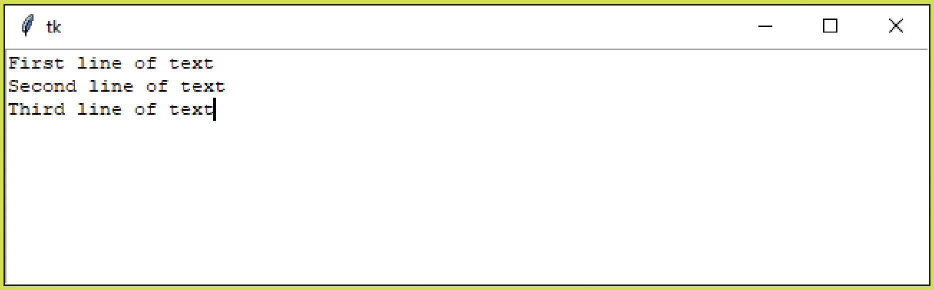
```
>>> root = Tk()
```

Now we'll create a Text object. Type the following statement and press **Enter**.

```
>>> t = Text(width=80, height=10)
```

The statement above creates a Text object and sets the variable t to refer to it. If the width and height values seem a bit smaller than we are used to (our drawing screen was 500 pixels in size), this is because the width of the text area is given in characters and the height is given in lines. As usual, the object will not be drawn until we've told Tkinter how to position it on the screen. Enter the following statement and press **Enter**.

```
>>> t.grid(row=0, column=0)
```



```
First line of text
Second line of text
Third line of text
```

The screenshot above shows the Text component in action. I've typed in a couple of lines. You should do the same.

The Text object allows a Python program a lot of control over the contents of the text window. For now, we just want to be able to read text back from a Text object. We can do this in a similar fashion to how we got text from the Entry object earlier in this chapter. However, we must work a little harder to address the text area that we want to read because we can refer to characters in the text in terms of their row and column positions. Enter the following statement and press **Enter**.

```
>>> t.get('1.0',END)
'First line of text\nSecond line of text\nThird line of text\n'
```

This statement gets all the text out of the Text object, starting at row 1 (the first row of the text), column 0 (the first column of the text). The value END specifies the end of the text, but you can specify a position in the text for the endpoint if you wish. If you just want to read the second line of text, you could use the following:

```
>>> t.get('2.0', '3.0')
'Second line of text\n'
```

We can use the delete method to delete portions of text from the Text object. Enter the following statement and press **Enter** to clear the text display.

```
>>> t.delete('1.0', END)
```

We can add text by stating the start position and then giving the text to be added. Enter the following statement to do just this:

```
>>> t.insert('1.0', 'New line 1\nNew line 2')
```

This inserts text into the Text area, starting at the beginning of the area. Note that the new line character '\n' is used to split lines on the display.

Group display elements in frames

A grid provides a way for you to design a layout for a complete window on the screen, but you often want to lay out subcomponents that you want to add to the window. We can do this by using a Frame. A Frame can act as a root for a set of elements displayed within it. We could use a frame to create a layout for the editing of a StockItem from our Fashion Shop application. Once we've created the Frame object, we can then include this in other display elements.

Using frames is very easy. We simply create the frame and then use the frame as the root object for all the items to be displayed within it:

```
frame = Frame(root)
stock_ref_label = Label(frame, text='Stock ref:')
stock_ref_label.grid(sticky=E, row=0, column=0, padx=5, pady=5)
```

Create a new frame

Add a label to the frame

Place the label in a grid inside the frame

The `stock_ref_label` is now part of the frame and will be positioned in the top left corner of the frame. Frames work well if you want to display the same information in several different applications.

Create an editable StockItem using a GUI

Now we can put these elements together to create an editable `StockItem` for use in a version of the Fashion Shop application that uses a graphical user interface. We'll create an object that will support the following three behaviors:

- Clear the editor display
- Put a `StockItem` on display for the user to edit
- Load a `StockItem` from the display after editing

We can call this object `StockItemEditor`, and it will contain methods for each of the behaviors above. Below, you can find an "empty" implementation of the class. It contains methods that currently just contain the empty statement `pass`. Next, we'll fill in these methods.

```
class StockItemEditor(object):
    '''
    Provides an editor for a StockItem
    The frame property gives the Tkinter frame
    that is used to display the editor
    '''

    def __init__(self,root):
        '''
        Create an instance of the editor. root provides
        the Tkinter root frame for the editor
        '''
        pass
    def clear_editor(self):
        '''
        Clears the editor window
        '''
        pass

    def load_into_editor(self, item):
        '''
        Loads a StockItem into the editor display
        item is a reference to the StockItem
        being loaded into the display
        '''
        pass

    def get_from_editor(self,item):
        '''
        Gets updated values from the screen
        item is a reference to the StockItem
        that will get the updated values
        Will raise an exception if the price entry
        cannot be converted into a number
        '''
        pass
```

We can create the initializer first. This is the method that sets up the object. It must create all the display objects and add them to the frame. Note that we don't create the editor when we want to edit a `StockItem`; we create it when the program starts. The editor provides the place where `StockItems` will be loaded to be edited.

```
class StockItemEditor(object):

    def __init__(self,root):
        self.frame = Frame(root)
```

Pass the constructor the root of the display for the frame

Create the frame to hold the editor

```
        stock_ref_label = Label(self.frame, text='Stock ref:')
        stock_ref_label.grid(sticky=E, row=0, column=0, padx=5, pady=5)
        self._stock_ref_entry = Entry(self.frame, width=30)
        self._stock_ref_entry.grid(sticky=W, row=0, column=1, padx=5, pady=5)
```

Stock reference editor

```
        price_label = Label(self.frame, text='Price:')
        price_label.grid(sticky=E, row=1, column=0, padx=5, pady=5)
        self._price_entry = Entry(self.frame, width=30)
        self._price_entry.grid(sticky=W, row=1, column=1, padx=5, pady=5)
```
Price editor

```
        self._stock_level_label = Label(self.frame,text='Stock level: 0')
        self._stock_level_label.grid( row=2, column=0, columnspan=2, padx=5, pady=5)
```
Stock level display

```
        tags_label = Label(self.frame,text='Tags:')
        tags_label.grid(sticky=E+N, row=3, column=0, padx=5, pady=5)
        self._tags_text = Text(self.frame, width=50, height=5)
        self._tags_text.grid(row=3, column=1, padx=5, pady=5)
```
Tags editor

In our application, we will create a new StockItemEditor and place it on the screen as follows:

```
from tkinter import *

root = Tk()

stock_frame = StockItemEditor(root)
stock_frame.frame.grid(row=0, column=0)
```
Import the Tkinter library

Create the root display

Create the StockItemEditor

Place the frame from the StockItemEditor on the display

CODE ANALYSIS

Creating a StockItemEditor

There are no new features being used in this initializer, but you might have some questions.

Question: Why do only some of the display elements have the self in front of them?

Answer: This is because not all the items on the display will be used after the display has been created. Consider the following:

```
stock_ref_label = Label(self.frame, text='Stock ref:')
stock_ref_label.grid(sticky=E, row=0, column=0, padx=5, pady=5)
self._stock_ref_entry = Entry(self.frame, width=30)
self._stock_ref_entry.grid(sticky=W, row=0, column=1, padx=5, pady=5)
```

These are the screen objects that provide access to the stock reference. The first object is the Label that appears on the display next to the item. The second is the Entry object that is used to display and enter the stock reference information. The object doesn't need to use the label once it has been created, so there's no point in making it an attribute of the class. The program simply uses a variable that will be local to the __init__ method and discarded when the method ends.

However, the Entry object will be changed when we display a StockItem, and so it must be stored as an attribute so that it can be used by other methods in the StockItemEditor class.

Question: What is the frame attribute of the StockItemEditor class used for?

Answer: The StockItemEditor class creates a frame that contains the objects that perform the editing. The program creating the display needs to have access to this frame so that it can be positioned on the display. So, the StockItemEditor class provides an attribute, called frame, that provides this value. You can see it used in the statement that positions the StockItemEditor on the display:

```
stock_frame.frame.grid(row=0, column=0)
```

The variable stock_frame refers to the StockItemEditor that's just been created. The statement above gets the frame attribute out of this object and calls the grid method on the frame to position the StockItemEditor at row 0 and column 0 on the display.

Now we can look at the method that will clear the display. We will use this in two situations: when we are loading a new element for editing (to get rid of any text that might be there) and when we have finished editing.

```
def clear_editor(self):
    '''
    Clears the editor window
    '''
    self._stock_ref_entry.delete(0, END)
    self._price_entry.delete(0, END)
    self._tags_text.delete('0.0', END)
    self._stock_level_label.config(text = 'Stock level : 0')
```

This method just clears all display items and changes the text on the stock level label to indicate that there are no items in stock. The next method we can examine in the StockItemEditor is the one that takes a StockItem and makes it available for editing. The values in the StockItem must be copied onto the editing objects. I've called the method load_into_editor.

```
def load_into_editor(self, item):
    clear_editor()
    self._stock_ref_entry.insert(0, item.stock_ref)
    self._price_entry.insert(0, str(item.price))
    self._stock_level_label.config(text = 'Stock level : ' + str(item.stock_level))
    self._tags_text.insert('0.0', item.text_tags)
```

item is a reference to the stock item being edited

Clear the editor

Insert the stock reference from the stock item

Display the list of tags as a text string

Display the stock level as a label

Convert the price value into a string and display it

We can get a StockItem object ready for editing by calling this method and passing the stockitem into it. The listing below does just that. Note that this is just test code; in the finished application, the item to be edited will be one of the items in the stock of the shop.

```
item = StockItem(stock_ref='D001', price=120,
                    tags='dress,color:red,loc:shop
window,pattern:swirly,size:12,evening,long')

stock_frame.load_into_editor(item)
```

Create a test StockItem

Send the StockItem to the edit frame

CODE ANALYSIS

The load_into_editor method

You might have some questions about load_into_editor.

Question: For what is this method used?

Answer: We will call this method when the user has selected a StockItem that they want to edit. In the Command Shell version of the program, we would use the print function to ask the user to give new values and the input function to read them back. We did this in Chapter 9 in the section "Editing a contact" for our contacts store.

An editor that uses a graphical user interface must work differently. It must display the StockItem and then allow the user to edit it. You use this way of working every time you edit a document using a word processor. The word processor loads the document, lets you edit it, and then saves the document. We have just written the load behavior for our "StockItem processor."

Question: Why are some of the items converted to a string before editing?

Answer: The price of an item is held as an integer. We need to convert the integer into a string so that the user can edit it. When we get the items back from the editor, we'll have to convert them from a string back into an integer.

Question: What is the text_tags attribute of a StockItem?

> **Answer:** The StockItem holds a set of tags that are used by the fashion shop owner to locate stock items with which she wants to work. The text_tags attribute is a property that converts this set of tags into a string of text that can be displayed and edited. There's nothing special about the code that implements the property; it's a variant of the code we used in Chapter 10 when we converted a list of Session objects into a text report. Look in the section "The Python join method" for more details.

The next method we need is the one that fetches an edited StockItem from the frame. The method is called get_from_editor and is used to complete the editing of a StockItem. This will happen when the user presses a Save button on the user interface. You can think of this method as the reverse of load_into_editor.

```
def get_from_editor(self,item):
    item.set_price(int(self._price_entry.get()))
    item.stock_ref = self._stock_ref_entry.get()
    item.text_tags = self._tags_text.get('1.0',END)
```

Convert the price string into an int and store it

Put the stock reference back into the stock item

Set the tags to the edited string

This code will run when the user presses a button to indicate that they've finished editing. The code below shows the save_edit function and a button that can be pressed to save the edited StockItem.

```
def save_edit():
    stock_frame.get_from_editor(item)
    stock_frame.clear_editor()

save_button = Button(root, text='Save', command=save_edit)
save_button.grid(row=1, column=0)
```

Called to save the edited stock item

Get the stock item from the editor

Clear the editor

CODE ANALYSIS

The get_from_editor method

You might have some questions about get_from_editor.

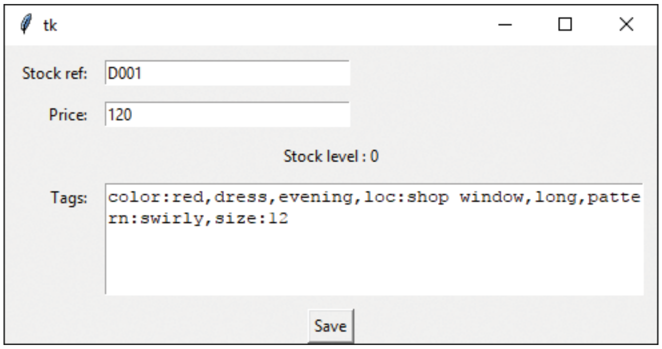
Question: What is the purpose of this method?

> **Answer:** This is the method that takes the edited StockItem details and puts them back into a StockItem. You can think of this as the Fashion Shop equivalent of the code that takes your edited text and stores it when you press Save in a word processor.

Question: Can this method fail?

Answer: Yes, it can. If the user doesn't enter a valid number into the price `Entry`, it will not be possible for the number to be converted, and the save method will raise an exception. A user of this method would have to take this into account when they write their program. Otherwise, there is the danger that the fashion shop owner might be left thinking that a save had succeeded when it had failed.

We can use the `load_into_editor`, `get_from_editor` and `clear_editor` methods to create a test editor for `StockItems`. The user interface will appear as in **Figure 13-21**.



```
color:red,dress,evening,loc:shop window,long,pattern:swirly,size:12
```

Figure 13-21 Editing a `stockitem`

The program below creates a test `StockItem` and allows the user to edit it. The user can finish the edit by pressing the Save button. When Save is pressed, the updated values are loaded from the edit window, and then the updated `StockItem` is printed. Finally, the edit window is cleared. This version is very basic (it doesn't do any checking for errors), but it does show how well this works. You can find the example in the folder **EG13-08 StockEditDemo** in the sample code for this chapter. You can open the folder using Visual Studio Code and run the file **StockItemEditDemo**, or you can open the same file and run it from IDLE.

```
# EG13.08 StockItemEditDemo

from tkinter import *
from StockItem import StockItem
from StockItemEditor import StockItemEditor

item = StockItem('D001', 120,
                'dress,color:red,loc:shop
window,pattern:swirly,size:12,evening,long')
```

Import the items we're using

Create a test `stockitem`

```
root = Tk()                                            Start Tkinter running

stock_frame = StockItemEditor(root)                    Create a stock editor frame
stock_frame.frame.grid(row=0, column=0)                Place the editor at the top of the window

def save_edit():                                       Function that saves the edited stock item
    stock_frame.get_from_editor(item)                  Get the item back from the editor
    print(item)                                        Print the edited item
    stock_frame.clear_editor()                         Clear the editor

save_button = Button(root, text='Save', command=save_edit)   Create a Save button
save_button.grid(row=1, column=0)                      Put the Save button on the display

stock_frame.load_into_editor(item)                     Load the stockitem we're editing

root.mainloop()                                        Start the display
```

CODE ANALYSIS

Editing Stock Items

You might have some questions about this code.

Question: Would it not make sense to put the editing behavior inside the StockItem class?

Answer: Good question. We've been talking about the importance of making objects that can just look after themselves, and you might think it would make sense to put the frame editor into the StockItem class. However, I don't think this is a particularly good idea. Another principle of object orientation is that an object should have a single purpose. The job of a StockItem object is to hold the data about an item of stock. It is not the job of the StockItem object to edit itself. We're designing our application so that we can use the same StockItem objects to store stock details, but the task of editing is quite different from storing.

So, a separate StockItemEditor class is a better idea. Another way to consider this would be to consider what would happen if we added the frame editor into the StockItem class and then made a version of the program that used the command shell user interface. We would have a lot of code floating around in the StockItem class that was never used.

Create a Listbox selector

We now know just about everything we need to know to create our graphical user interface version of the Fashion Shop application. We can put buttons on the screen to initiate actions, and we can edit and store StockItem objects. The last thing we need to discover is an easy way of allowing the fashion shop owner to find and select her stock items. We could ask her to type in the stock reference of an item for which she wishes to search, and then press a Find button to search for the item with that stock reference. This would work, but when we discuss this idea with our customer, she doesn't sound very keen on the idea. What she wants is the ability to pick stock items out of a list. It turns out that Tkinter has a Listbox object that allows us to do this kind of thing, so we agree to take on the project.

MAKE SOMETHING HAPPEN

Investigating the Listbox object

We can investigate the Listbox object from the Python Command Shell in IDLE. So, let's start that up. Just like the last few investigations, the first thing we need to do is import all the resources from the Tkinter module and create a root window. Give the following commands and press **Enter** after each:

```
>>> from tkinter import *
>>> root = Tk()
```

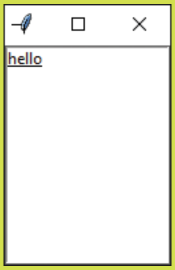
Next, we need to create a Listbox object on the screen. Type the statements below and press **Enter** after each one.

```
>>> lb = Listbox(root)
>>> lb.grid(row=0, column=0)
```

These statements create a Listbox and set the variable lb to refer to it. The Listbox is then displayed in the window. You should now see an empty Listbox in the window. We can add some items to the Listbox using the insert method. Type in the following and press **Enter**.

```
>>> lb.insert(0, 'hello')
```

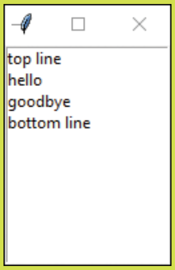
The first argument to the insert call is the position in the Listbox where we want to insert the item. The second argument is the text to insert in the list. You should see the item appear in the Listbox.


hello

Let's add some more items. Type in the following statements, pressing **Enter** after each one.

```
>>> lb.insert(1,'goodbye')
>>> lb.insert(0,'top line')
>>> lb.insert(END, 'bottom line')
```

The entry 'goodbye' is inserted after hello at position 1, whereas the entry 'top line' is inserted right at the top. The location END means the end of the list, so you should find that your Listbox looks like this:


top line
hello
goodbye
bottom line

We can work through the StockItem objects and use the stock reference of each item to build up a Listbox. Now we need to know how the user can select items in the box. This is another event to which we can bind a function. Let's write the function first. Type in the following statements, pressing **Enter** after each statement and remembering to enter a blank line at the end.

```
>>> def on_select(event):
        lb = event.widget
        index = int(lb.currentselection()[0])
        print(lb.get(index))
```

This function will run when the user clicks on one of the items in the Listbox. The first statement gets the object that caused the event. This is provided by the widget attribute of the event supplied as a parameter. We know that this is the Listbox, so we ask the Listbox to give us the index of the currentselection. Available options allow a user to select multiple items in a Listbox (although we're not using these), so the currentselection method returns a tuple that contains all the selected items. We're selecting only one item, so we can just get the first item (the one at element 0) in the tuple. We can then use this index in the get method on the Listbox to get that item from the Listbox.

The result of these three statements is that the method will find the selected item in the Listbox and then print it. Next, we need to bind this event handler to the "event selected" event in the Listbox. Type in the following statement and press **Enter**.

```
>>> lb.bind('<<ListboxSelect>>', on_select)
```

This statement should be familiar. It is how we connected event handlers in our drawing application. Now, when you click on an object in the Listbox, the selected item is printed on the console. The Fashion Shop application will use the selected stock reference to locate and display the item to which it refers.

Create a StockItem selector

We can use a Listbox to allow the user of the Fashion Shop application to select an item from its stock reference. Now we'll create a class called StockItemSelector that we can use to generate a Frame that can be displayed in the GUI for our Fashion Shop. When I make the StockItemSelector class, I'll follow the same pattern as for the StockItemEditor class by deciding what the StockItemSelector class needs to do and then filling in the methods. The two things I think the StockItemSelector class needs to do are:

- Accept some StockItems from which to select
- Tell me when an item has been selected from the list

The first of these actions seems to make sense. We just need to create a method in the StockItemSelector class that can be called to tell the StockItemSelector to populate the Listbox. However, the second action is a bit trickier. We're quite happy with the idea of calling objects to make them do things for us, and we've done this a lot. We call a method in the StockItem class to add stock, and another method to tell the StockItem that stock has been sold. But how do we make an object tell us things? Programmers call this part of development *message passing*. One object is sending a message to another. In this case, the StockItemSelector class wants to send a message to an object to tell it that a StockItem has been selected.

It's actually very easy. We just give the sender object a reference to the receiver object and then when we want to deliver a message to the object, the code in the StockItemSelector just calls a method on that reference. We can give this reference when we initialize the StockItemSelector class.

```
class StockItemSelector(object):
    '''
    Provides a frame that can be used to select
    a given stock item reference from a list
    of stock items
    The stock item list is delivered to the
    class via the populate_listbox method
    Selection events will trigger a call
    of got_selection in the object provided
    as the receiver of selection messages
    '''
    def __init__(self, root, receiver):
        '''
        Create an instance of the editor. root provides
        the Tkinter root frame for the editor
        receiver is a reference to the object that
        will receive messages when an item is selected
        The event will take the form of a call
        to the got_selection method in the
        receiver
        '''
        pass

    def populate_listbox(self, items):
        '''
        Clears the selection Listbox and then
        populates it with the stock_ref values
        in the collection of items that have
        been supplied
        '''
        pass
```

This is the empty class that contains the methods that need to be filled in. Let's look at the `__init__` method first.

```
def __init__(self, root, receiver):
    self.receiver = receiver

    self.frame = Frame(root)

    self.listbox = Listbox(self.frame)
    self.listbox.grid(row=0, column=0)

    def on_select(event):
        '''
        Bound to the selection event in the Listbox
        Finds the selected text and calls
        the message receiver to deliver the name
        that has been selected
        '''
        lb = event.widget
        index = int(lb.curselection()[0])
        receiver.got_selection(lb.get(index))

    self.listbox.bind('<<ListboxSelect>>', on_select)
```

- `def __init__(self, root, receiver):` — Initialize the `StockItemSelector`
- `self.receiver = receiver` — Store the reference to the receiver so that we can deliver results to it
- `self.frame = Frame(root)` — Create the frame that we will use to store the controls
- `self.listbox = Listbox(self.frame)` — Create a `Listbox` in the frame
- `self.listbox.grid(row=0, column=0)` — Place the `Listbox` in the frame
- `lb = event.widget` — Gets the `Listbox` that produced the event
- `index = int(lb.curselection()[0])` — Get the index of the selected item
- `receiver.got_selection(lb.get(index))` — Call the `got_selection` method in the message receiver object
- `self.listbox.bind('<<ListboxSelect>>', on_select)` — Bind the `got_selection` event handler to the `Listbox`

CODE ANALYSIS

Selecting Stock Items

You might have some questions about this code.

Question: What are we doing in this method?

Answer: We are setting up an instance of the `StockItemSelector` class that can be used to display a `Listbox` of stock item references. When the user selects one of these references, we want to tell another object that this has happened. The `__init__` method accepts two parameters: the root frame for the window that will be used to display this frame, and a reference to the object that will receive a message each time the user selects a stock item.

The `__init__` method stores a reference to the message receiver, builds a `Listbox`, and then creates an event handler that will run when the user selects something from the list.

Question: What happens if the receiver doesn't have a got_selection method?

Answer: Good question. The idea is that the StockItemSelector will call the got_selection method on the receiver object when the user selects an item in the Listbox. If there is no method in the receiver object, the program will fail at this point with an exception. Fortunately, Python provides a built-in function that can be used to determine whether a particular object has a given attribute, so we could add an assert to test that a given object will work:

```
assert hasattr(receiver, 'got_selection')
```

The hasattr function accepts two arguments: a reference to an object, and a string. It returns True if the object has an attribute with the given name. The above statement (which we should add to __init__) will cause the program to raise an exception if the receiver (which is supposed to have a method called got_selection) does not have a got_selection method.

The second method in the StockItemSelector class accepts some StockItems to display in the Listbox.

```
def populate_listbox(self, items):
    self.listbox.delete(0, END)
    for item in items:
        self.listbox.insert(END,item.stock_ref)
```

Add the items to the Listbox

Clear the Listbox of previous values

Iterate through each item that has been supplied

Add the stock_item attribute of the item to the end of the Listbox

Now that we have our selection class, we can create a program that will test it. We can create a class that contains a got_selection method and then connect an instance of that class to the selector object.

```
# EG13.09 StockSelectDemo

from tkinter import *

from StockItem import StockItem
from StockItemSelector import StockItemSelector

class MessageReceiver(object):

    def got_selection(self, stock_ref):
```

Import all the required items

Class that will act as the receiver of the selection messages

Method that will be called when an item is selected

```
        print('Stock item selected :', stock_ref)


stock_list = []

for i in range(1,100):
    stock_ref = 'D' + str(i)
    item = StockItem(stock_ref, 120,
                     'dress,color:red,loc:shop
window,pattern:swirly,size:12,evening,long')
    stock_list.append(item)

receiver = MessageReceiver()

root = Tk()

stock_selector = StockItemSelector(root, receiver)
stock_selector.populate_listbox(stock_list)
stock_selector.frame.grid(row=0, column=0)

root.mainloop()
```

Print a message to show that the selection has taken place

Create a test stock list

Create 100 test stock items

Create a stock reference for this item

Create a test stock item

Add the test stock item to the list

Create an instance of the message receiver class

Create the display

Create a `StockItemSelector` instance

Populate the `StockItemSelector` with the sample stock list

Add the `StockItemSelector` frame to the display

Start the display loop

The program above is a demonstration of how the `StockItemSelector` is used. It creates 100 sample stock items and uses these to create a stock selector. When a stock item is selected, the stock reference of the selected item is printed. **Figure 13-22** below shows the output from the program. You can find the entire sample program in the folder **EG13-09 StockSelectDemo** with the sample program files for this chapter. Run the program **StockItemSelectorDemo.py** to see the demonstration.

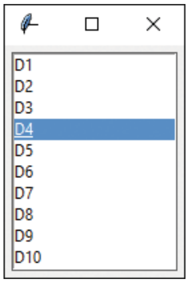
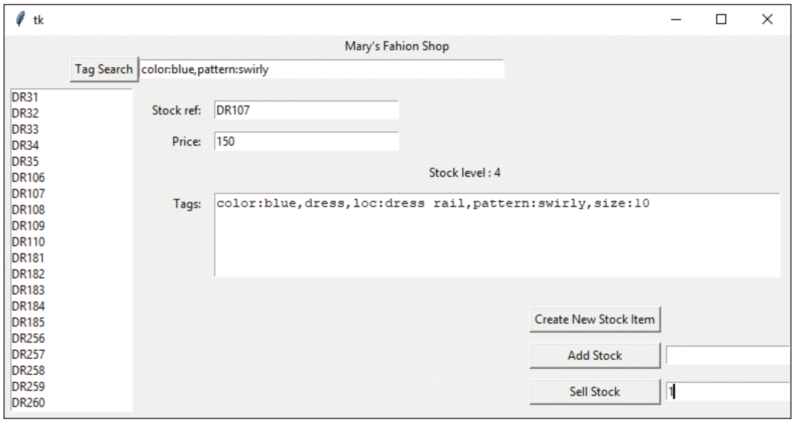

D1
D2
D3
D4
D5
D6
D7
D8
D9
D10

Figure 13-22 Testing the StockItemSelector

An application with a graphical user interface

Figure 13-23 shows the completed Fashion Shop with a graphical user interface. On the left, you can see the `StockItemSelector` in action, and at the right of the frame, you can see the `StockItem` editor. The remaining elements on the screen are buttons wired into the graphical user interface. They send commands to the various elements in the application, which seems to work. On the top, I've added a Search button. The fashion shop owner can enter search tags and the press the Search button to filter the selection of stock that is shown. The application is presently showing all the blue items with a swirly pattern.



color:blue,pattern:swirly

DR31
DR32
DR33
DR34
DR35
DR106
DR107
DR108
DR109
DR110
DR181
DR182
DR183
DR184
DR185
DR256
DR257
DR258
DR259
DR260

Stock ref: DR107

Price: 150

Stock level : 4

Tags: color:blue,dress,loc:dress rail,pattern:swirly,size:10

Sell Stock: 1

Figure 13-23 A Fashion Shop application with a graphical user interface

The user can add and sell amounts of stock by entering a number and pressing the appropriate button. The selected stock item is then updated. The user can also edit the details of a stock item. The changes are stored when the user navigates away from that item onto another. To create a new item, the user presses the **Create New Stock Item** button and then enters the new stock item details. When they move off that item, it is automatically saved in the application. When the user closes the application, the shop data is automatically stored in a file using pickling. This would serve as the basis of a working stock management system.

You can learn a lot by going through this code. You can find it in the folder **EG13-10 FashionShop** in the sample programs for this chapter. If you start the **FashionShopShellUIApp** program, you get a Fashion Shop that you can manage via the Command Shell. If you start the **FashionShopGUIApp** application, you get a Fashion Shop that you can manage via a graphical user interface. However, both programs use the same stock management classes.

CODE ANALYSIS

Complete Fashion Shop application

You might have some questions about my Fashion Shop application.

Question: Can we change the size of the text on the screen?

> **Answer:** Yes. When you create a `Label` object, you can set the font and text size to be used for the label. You can even create labels that contain images. The Tkinter framework is extremely powerful, and it is well worth finding out more about it.

Question: Can we stop the Fashion Shop application from displaying the Command Shell each time it runs?

> **Answer:** Yes, you can. You do this by changing the file extension of the Python program from .py (which means "contains a Python program") to .pyw (which means "contains a Python windows program"). I've done this for the **FashionShopGUIApp** in the folder **EG13-10 FashionShop**.

PROGRAMMER'S POINT

Always try using the programs you've written

This sounds like a stupid observation. Of course, you should try to use a program that you just wrote. But what I mean is that you should try to use it properly. You should try entering ten items of stock and find out if there's anything annoying about the way your program works. My first version of the Fashion Shop above displayed a message box each time an item was edited or saved. I thought this was a nice idea, but it turns out that it's a pain to keep clearing message box items after every action, so I changed it to now only display a message if something goes wrong.

When I was teaching programming, I'd watch people laboriously demonstrate programs they had written that were obviously horrible to use. I'd ask them afterward how they would ever expect their customer to use them when even the developer had a tough time making them work. I'm fairly happy with the Fashion Shop application, but I'm also fairly sure that after a day spent using it, I'd make a few changes to the way it works.

MAKE SOMETHING HAPPEN

Build your own application

The Fashion Shop program is a great jumping-off point for any application that you might like to write to store information about items. Think of something you'd like to store data about—perhaps favorite football players, recipes, monster trucks, or whatever—identify the items about each that you'd like to store, and then use the Fashion Shop code as the basis of an application that can manage that data.

What you have learned

In this chapter, you started by learning a bit about Visual Studio Code, a development tool that makes creating programs made from multiple components easier. Then you found out about graphical user interfaces. These are made up of objects that represent items on the screen—for example, labels, text to be entered, and buttons to be pressed. The screen display serves as a container for these objects, which can be positioned on the screen using a grid to lay them out. Each display object is placed at a specific location (a cell) in the grid and can be made to span one or more grid cells. An object can be positioned within the cell using "sticky" points of the compass. If an object is made to stick to both sides of a cell (for example the "east" and the "west" of the cell), then it is stretched to fill the cell boundaries.

Objects on the screen can generate events, which are mapped onto calls to a Python function or method in a class. An example of this behavior is the Button display component, which calls a command method when the button is pressed by the user. However, a program can bind to events generated by all components. The events can be originated by mouse, keyboard, or screen events. We saw these in action and learned how to draw graphics on a canvas when we made a simple drawing program.

You also extended the event mechanism into your own programs, where a stock item selector was made to generate an event in the Fashion Shop user interface when the user of the program selects an item.

Here are some points to ponder about graphical user interfaces.

Is Tkinter the only way to create Graphical User Interfaces in Python?

No. I like Tkinter because it is part of Python (and therefore available everywhere), easy to start with, and it does what I want. However, there are lots of other systems that your program can use to create a graphical user interface. If you want to try something different, try Kivy (kivy.org/#home) or PyQT (wiki.python.org/moin/PyQt). The thing to remember is that having used Tkinter you now know the fundamentals of graphical user interface construction and you can apply this knowledge to other libraries that you might want to use in the future.

Are programs with a graphical user interface easier to create than those that use a Command Shell?

This is a very good question. When we were writing the programs that used the Python Command Shell, the program had to ask the user questions and then make sense of the replies. But with a GUI, we can just provide buttons for the user to press. A program with a GUI doesn't need to worry about what to do if the user enters an invalid command, because all the user can do is press the buttons on the screen.

This seems to imply that programs with a GUI might be easier to create, and in some ways, they are. However, you need to spend time making sure that what happens when buttons are pressed are the right actions, which can be tricky and will test your organizational skills.

Is a program with a GUI still a "data processing" program?

This is a very good question. When we started programming, we had this model of a computer program as something that takes in some data, does something with it, and then produces an output. A program with a GUI doesn't seem to work this way. The user will type in some data in one place and then press a button to perform an action.

I find it best to think of the event handlers that run inside a program with a graphical user interface as tiny programs that all cooperate to make the system work. The programmer just needs to ensure that the actions fit together to make a complete system. At the beginning of this book, I said "If you can plan a birthday party, you can write a program."

When you're creating a program that uses software components and a graphical user interface, you find yourself in the role of an organizer as much as a programmer, as you seek to ensure that messages from one source are used to trigger actions in components to produce the results that the user wants. From a design perspective, it's also a good idea to separate the classes that deal with the user interface from those that store the data. We've seen that this gives flexibility, in that we have created a Fashion Shop application with a Graphical User Interface that uses exactly the same data storage code as our previous text version of the application.