

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.





## 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`.

[Click here to view code image](#)

```
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.

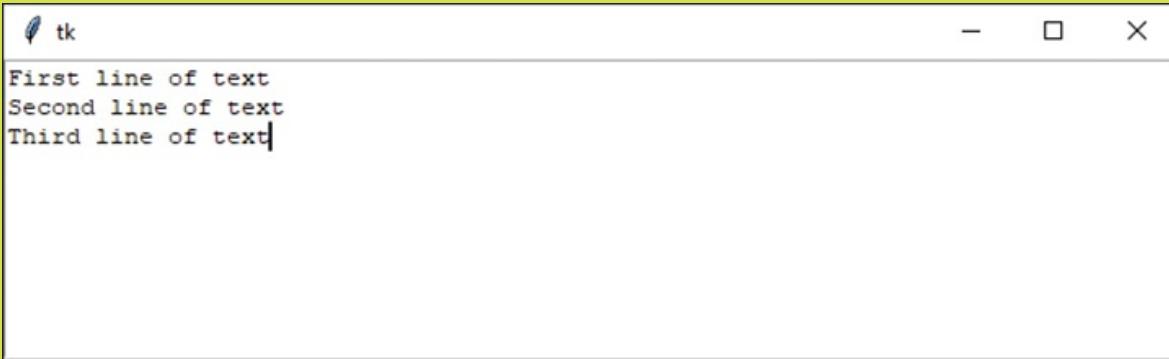
```
>>> root = Tk()
```

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)
```



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**.

[Click here to view code image](#)

```
>>> 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:

[Click here to view code image](#)

```
>>> 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:

[Click here to view code image](#)

```
>>> 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)          Create a new frame  
stock_ref_label = Label(frame, text='Stock ref:') Add a label to the frame  
stock_ref_label.grid(sticky=E, row=0, column=0, padx=5, pady=5) 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.

[Click here to view code image](#)

```
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):  
        Pass the constructor the root of the display for the frame  
        self.frame = Frame(root)           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 *           Import the Tkinter library

root = Tk()                     Create the root display

stock_frame = StockItemEditor(root)           Create the StockItemEditor
stock_frame.frame.grid(row=0, column=0)         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:

[Click here to view code image](#)

```
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:

[Click here to view code image](#)

```
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.

[Click here to view code image](#)

```
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):  
    item is a reference to the stock item being edited  
    clear_editor()  
    Clear the editor  
    self._stock_ref_entry.insert(0, item.stock_ref)  
    Insert the stock reference  
    from the stock item  
    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)  
    Display the list of tags as a text string  
    self._stock_level_label  
    Display the stock level as a label  
    self._price_entry  
    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)
```



## 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?

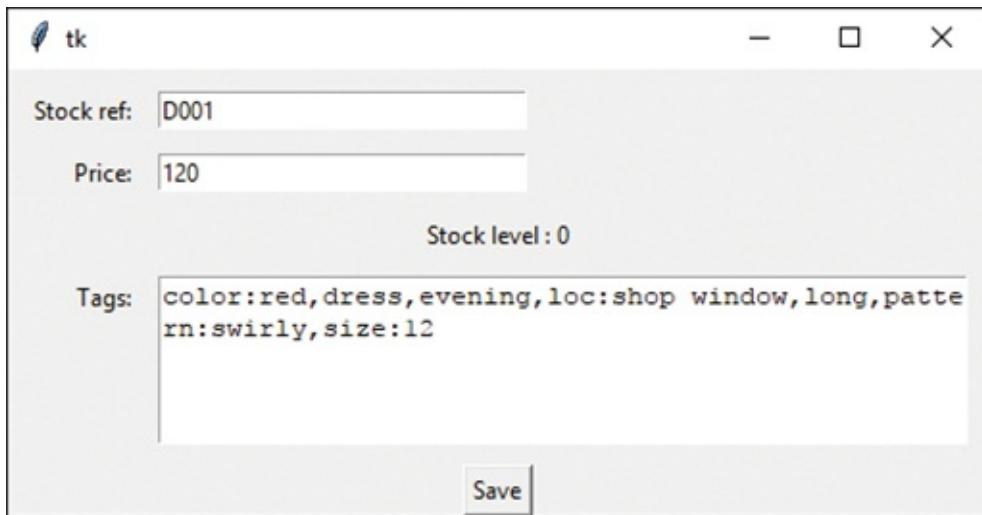
**Answer:** This is the method that takes the edited `StockItem` details and puts them back into a `StockItem`. You can think of this as the Fashion Shop equivalent of the code that takes your edited text and stores it when you press Save in a word processor.

**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**.



**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 Import the items we're using

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

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

[Click here to view code image](#)

```
>>> 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.

[Click here to view code image](#)

```
>>> 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](#).



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

[Click here to view code image](#)

```
>>> 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:



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.

[Click here to view code image](#)

```
>>> 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**.

[Click here to view code image](#)

```
>>> 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.

[Click here to view code image](#)

```
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
    Initialize the StockItemSelector
    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)
    self.listbox.grid(row=0, column=0)
    Create a Listbox in the frame
    Place the Listbox in the frame

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
    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:

[Click here to view code image](#)

```
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_ref` 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):
        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()
```

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 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

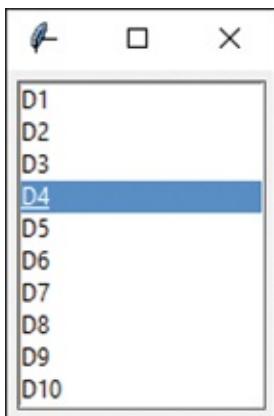
Create a StockItemSelector instance

Populate the StockItemSelector with the sample stock list

Add the StockItemSelector frame to the display

Start the display loop

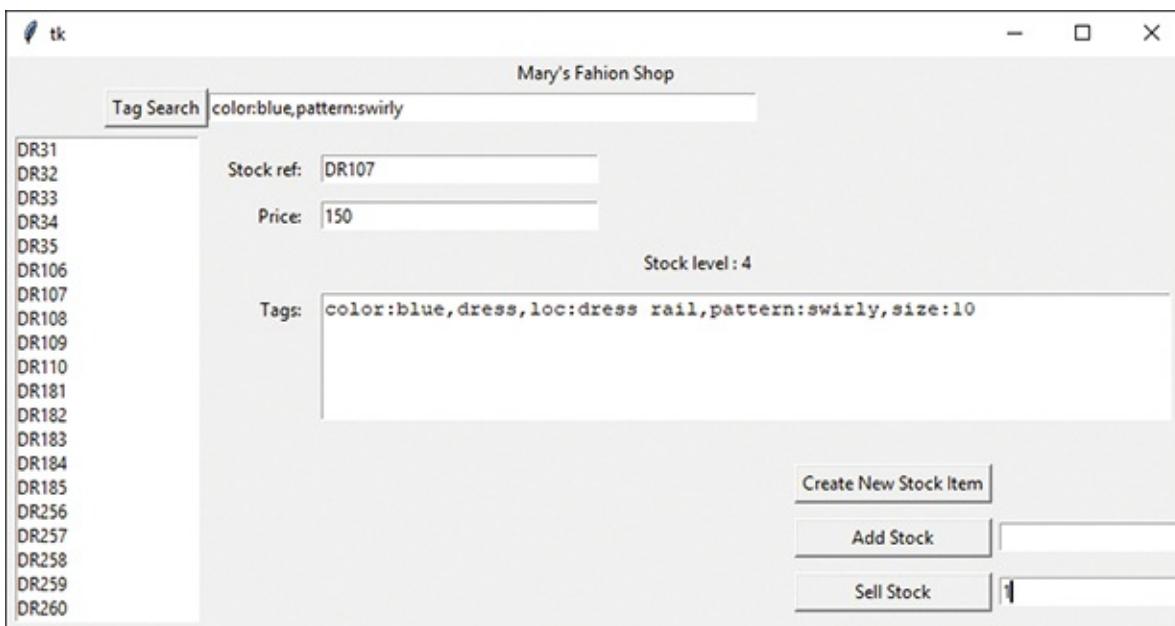
The program above is a demonstration of how the `StockItemSelector` is used. It creates 100 sample stock items and uses these to create a stock selector. When a stock item is selected, the stock reference of the selected item is printed. **Figure 13-22** below shows the output from the program. You can 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.



**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.



## 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](http://kivy.org/#home)) or PyQt ([wiki.python.org/moin/PyQt](http://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.

# 14 Python programs as network clients



## What you will learn

In this chapter, you'll discover how to create Python programs that interact with the Internet and the World Wide Web. You'll learn the fundamentals of computer networking and how to create a program that can grab information from a web server on the Internet. You'll also learn about the standards used to transfer data between programs.

[Computer networking](#)

[Consume the web from Python](#)

[What you have learned](#)

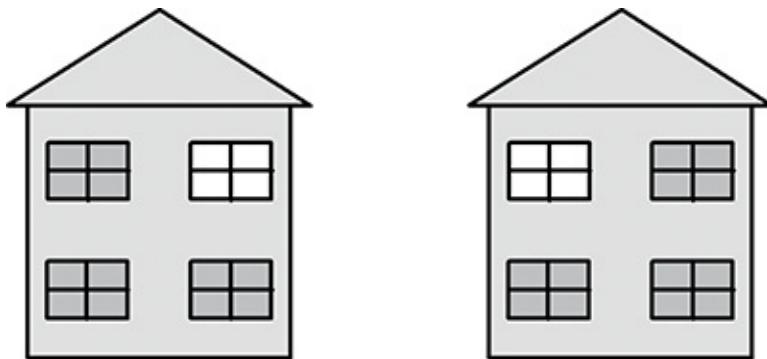
## Computer networking

Before we look at how Python programs use network connections, we need to

learn a little bit about networks. This is not a detailed description, but it should give you enough background to understand how our programs will work.

## Network communication

Networks can use wires, radio, or fiber optic links to send their data signals. Whatever the medium, the fundamental principle is that hardware puts data onto the medium in the form of digital bits and then gets it off again. A bit is either 0 or 1 (or you can think of a bit as either true or false) and can be signaled by the presence or absence of a voltage, light from a light-emitting diode (LED), or a radio signal. If you imagine signaling your friend in the house across the road by flashing your bedroom light on and off (**Figure 14-1**), you'll have an idea of the starting point of network communications.



**Figure 14-1** House-to-house networking

Once we have this raw ability to send a signal from one place to another, we can start transferring useful data. We could invent a protocol (an arrangement of messages and responses) and use it to pass messages. To communicate useful signals, you must agree on a message format. You could say, “If my light is off, and I flash it on twice, it means that it’s safe to come around because my sister is out. If I flash it once, it means don’t come. If I flash it three times, it means to come and bring pizza with you.” This is the basis of a protocol, which is an arrangement by communicating parties on the construction and meaning of messages.

The messages and the protocol are independent of each other. We could replace “flash the light” with “tap on the water pipes” or “make a puff of smoke,” and the protocol could be the same. Three flashes or three taps could each mean “bring pizza.” When we design networks, we can express this using layers, as

depicted in **Figure 14-2**.



**Figure 14-2** Layers in networks

The protocol sits on top of a physical layer that can deliver the network events. We can use light flashes, bangs on a pipe, or even puffs of smoke to deliver network messages. Each layer will set out standards. For example, the standard for the Lights physical layer in the network will state, “A flash must be no longer than one-half second, and all the flashes must occur within a five-second period.” The standard for the Pipes layer will describe how loud a tap on the pipe must be.

The transport protocol on top of the physical layer will be designed with no consideration for how the messages are sent; it only will be told what message events have been received. We can add new kinds of message delivery. For example, we could add a flag-waving delivery without having to change the entire network. The network protocols used by the Internet are based on this layered approach.

## Address messages

Your bedroom light communication system would be more complicated if you had two friends on your street who wanted to use their bedroom lights to communicate with you. You would have to add some form of addressing and give each person a unique address on the network. A message would now be made up of two components. The first component would be the address of the recipient, and the second would contain the message itself. Computer networks function in the same way. Every station on a physical network must have a unique address. Messages sent to that address are picked up by the network hardware in that station.

Networks also have a broadcast address, which allows a system to send a message that will be acted on by every system. In our “bedroom light network” a broadcast address could be used to warn everyone that your sister has come home and her new boyfriend is with her, so your house is to be avoided at all costs. In computer networks, a broadcast is how a new computer can learn the addresses for important systems on the network. A system can send out a broadcast saying, “Hi. I’m new around here!” Another system would respond with configuration information.

All the stations on a network can receive and act on a broadcast sent around it. In fact, if it wanted to, a station could listen to all the messages traveling down its part of the wire or Wi-Fi channel, which illustrates a problem with networks. Just as all your friends can see all the messages from your bedroom light, including those not meant for them, there is nothing to stop someone from eavesdropping on network traffic around your network. When you connect to a secure website, your computer is encoding all the messages it sends out so that someone listening other than the intended recipient would not be able to learn anything.

## Hosts and ports

If we want to use our bedroom light flashing protocol to talk to people at the same address, we need to improve our protocol. If we want to send messages to Chris and Imogen, who both live in the same house, we would need to improve our protocol so that a message contains data that identifies the recipient.

In the case of a computer system, we have the same problem. A given computer server can provide an immense variety of different services to the clients that connect to it. The server might be sending webpages to one user, sharing video with another, and hosting a multiplayer game for 20 people all at the same time. The different clients need a way of locating the service they want on the server.

The Internet achieves this by using “ports.” A port is a number that identifies a service provided by a computer. Some ports are “well-known.” For example, port number 80 is traditionally used for webpages. In other words, when your browser connects to a webpage, it’s using the Internet address of the server to find the actual computer, and then it’s connecting to port 80 to get the webpage from that server.

When a program starts a service, it tells the network software which port that service is sitting behind. When messages arrive for that port, the messages are passed to the program. If you think about it, the Internet is just a way that we can make one program talk to another on a distant computer. The program (perhaps a web server) you want to talk to sits behind a port on a computer connected to the Internet. You run another program (perhaps a web browser) that creates a connection to that port that lets you request webpages and display them on your computer.

Programmers can write programs that use any port number, but many network connections contain a component called a *firewall* that only allows certain packets addressed to particular well-known ports to be passed through, which reduces the chance of systems on the network coming under attack from malicious network programs.

## Send network messages with Python

Now that we know how the fundamentals of the network function, we can look at how a Python program can use a network to send and receive messages. We'll send a message using the *User Datagram Protocol (UDP)* element of the *internet protocol suite*.

A *datagram* is a single message sent from one system to another. The sender of a datagram has no idea that it has been received unless the recipient returns a message acknowledging receipt. The *internet protocol suite* is the set of standards that describes how the Internet and associated networks work. It is frequently referred to as the TCP/IP suite. This is because the standard originally described the *Transmission Control Protocol* that linked systems on a network and the *Internet Protocol* that allowed communications between networks. You can find a good description of how UDP works here:

[https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol).



MAKE SOMETHING HAPPEN

Send a network message

The best way to find out about networking is to use it to send a message from one program to another. We can do that 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 `socket` module. Give the following command and press **Enter**:

```
>>> import socket
```

The `socket` module contains the `socket` class that we'll use to create and manage network connections. Let's make an instance of the `socket` class to receive messages. Type in the statement below and press **Enter**:

[Click here to view code image](#)

```
>>> listen_socket =  
socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

The `socket` constructor accepts two arguments. The first is the *address family* that the socket will use to refer to hosts. In this case, we'll use the Internet address family, so we use the value `AF_INET` from the `socket` module. The second argument is the type of messages we will send. We will send *datagrams*. A datagram is a single, unacknowledged message that's sent from one system to another, in the same way that we could flash the lights in our inter-house network to deliver a message to someone who may or may not be watching.

Now that we've created our socket, we need to consider the address to which we'll connect it. A network address can be written as a tuple. Type in the statement below and press **Enter**:

[Click here to view code image](#)

```
>>> listen_address = ('localhost', 10001)
```

The `listen_address` tuple holds two values. The first of these is the address of the computer to which we will connect. Initially, we'll just send the messages to a process on our own computer so we can use the special address 'localhost' to represent the current machine. The second value in the tuple is the port to which the program will connect. Ports are identified by numbers. We'll use port 10001.

The next thing we need to do is *bind* the socket to the server address from which it will listen. Once we have done this, the socket can be made to listen for messages on the port given in the address. The `bind` method is given the address from which to listen, and it configures the socket to listen on the address given. Type in the following and press **Enter**.

[Click here to view code image](#)

```
>>> listen_socket.bind(listen_address)
```

Now we can ask our socket to receive some data. We can use the `recvfrom` method, which will fetch a single datagram. The method accepts an argument that gives the maximum size of the datagram that will be accepted. Type the following and press **Enter**.

[Click here to view code image](#)

```
>>> result = listen_socket.recvfrom(4096)
```

Notice that you don't get the `>>>` prompt back from this command because the `recvfrom` method has not yet returned; it is waiting for a datagram to arrive.

We now need to make a transmitter. We will need another copy of the IDLE Python Command Shell to do this, so start up another one. As with the listening program, the first thing we need to do is import the `socket` module:

```
>>> import socket
```

Now we can make a send socket. Type in the statement below and press **Enter**:

[Click here to view code image](#)

```
>>> send_socket =  
socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Note that `send_socket` is created in the same way as `listen_socket`. Next, we need to create an address to identify the recipient of the message. We'll send the message back to ourselves, so we use the same address. Type in the statement below and press **Enter**:

[Click here to view code image](#)

```
>>> listen_address = ('localhost', 10001)
```

And now, for the grand finale, we'll send a message over the network. Type in the following:

[Click here to view code image](#)

```
>>> send_socket.sendto(b'hello from me',  
send_address)
```

This sends a message from this IDLE Command Shell to anything listing on port 1001, which in our case is the listener program. Press **Enter** to send the message:

[Click here to view code image](#)

```
>>> send_socket.sendto(b'hello from me',
```

```
    send_address)
```

```
13
```

```
>>>
```

The `sendto` method returns the number of data bytes that the method has sent. In this case, it has sent 13 bytes (the number of characters in the string '`hello from me`'). You might be wondering why the string has the letter `b` in front of it. This is because Python 3 normally encodes string characters using a standard called Unicode (see "Working with Text" in [Chapter 4](#)). We can't send Unicode values over a socket, but we can send bytes. Putting a `b` in front of a string tells Python to make this string out of bytes rather than Unicode characters. So, now that the message has been sent, let's see if it has been received.

Go back to the IDLE Command Shell where the listener is running. You should see that the `>>>` prompt has returned because the `recvfrom` method has completed and returned a value into the variable `result`. We can use the `print` function to view the result:

```
>>> print(result)
```

When you press Enter to perform the `print`, you'll see that the contents of the `result` are a tuple that contains two items. The first item is a string containing the message sent from the sender. The second item is another tuple that contains the address of the system that sent the message. We'll talk about Internet addresses in the next exercise when we use these functions to send messages between computers.

[Click here to view code image](#)

```
>>> print(result)
(b'hello from me', ('127.0.0.1', 51883))
```

It might not seem like much, but these actions are the basic building blocks of every program that uses the Internet. Whenever you load a

webpage, stream a video, or send an email, the data is transferred by one process listening for packets of data and another sending packets of data.



## CODE ANALYSIS

# Sending network messages

You might have some questions about what we've just done.

**Question:** Can we send things other than text?

**Answer:** Yes. A datagram sends a block of byte values, but these can contain any kind of data. We are transferring strings, but we could just as easily transfer fashion shop stock items.

**Question:** What's the largest thing you can send?

**Answer:** We set the maximum size of the incoming message in the `recvfrom` method. A program can send a message of around 65,000 bytes. If we want to send larger items, we must send those as multiple messages. Fortunately, there are more network functions that can split and reassemble large items. We'll look at these later.

**Question:** What happens if we send a message and the listener is not listening?

**Answer:** Nothing. We're sending the simplest kind of message, a *datagram*. The sender has no way of knowing whether a datagram was received.

**Question:** Can the listener listen to messages from other computers?

**Answer:** Yes. As long as the messages are sent to the correct port

(in this case, port 10001), the listener will receive them.

**Question:** Can the sender send messages to other computers?

**Answer:** Yes. By using a different send address, a socket can send messages to other ports and machines.

**Question:** How long would the listener wait before it heard anything?

**Answer:** It would wait forever. However, the `Socket` module provides a method called `setdefaulttimeout` that can be used to set the number of seconds that a `recvfrom` method will wait for an incoming message. If nothing has arrived before the timeout has elapsed, the `recfrom` method will raise an exception.

**Question:** Can using sockets generate exceptions?

**Answer:** Yes. A program that uses network connections should take care to catch exceptions that might be raised when a network connection fails or a host disconnects unexpectedly.

## Send a message to another computer

The `sendto` and `recvfrom` methods can be used to send messages to another computer via a local network. You could use these methods to connect two machines you have at home. To do this, you need to obtain the IP (or Internet protocol) address of the machine to which you are sending the message. You can think of the IP address as the “phone number” of your computer on the network. If you don’t have the IP address of a computer, you can’t send messages to it. The Python socket module contains functions that can be used to find the IP address of the computer running the Python program. If you load the program below, it will print the address of the machine on which it is running. You can then use the address in the sender program.

```
# EG14.01 Receive packets on port 10001 from another machine
```

```
import socket
```

Import the socket library

```
host_name = socket.gethostname()
```

Get the host name for this computer

```
host_ip = socket.gethostbyname(host_name)
```

Use the host name to get the IP address

```
print('The IP address of this computer is:', host_ip)
```

Print the IP address

```
listen_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

Create the listen socket

```
listen_address = (host_ip, 10001)
```

Create the address to listen on this machine

```
listen_socket.bind(listen_address)
```

Bind the socket to the address

```
print('Listening:')
```

```
while(True):
```

Loop forever

```
    reply = listen_socket.recvfrom(4096)
```

Wait for an incoming message

```
    print(reply)
```

Print the message

When you run the receiver program above, it will print a message giving the IP address and then state that it is listening for inputs:

[Click here to view code image](#)

The IP address of this computer is: 192.168.1.55  
Listening:

Now you can load the send program on the machine that's doing the transmitting.

```

# EG14.02 Send packets on port 10001 to another machine

import socket
import time

# You will need to change this to the address
# of the machine to which you are sending
target_ip = '192.168.1.55'

send_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
destination_address = (target_ip, 10001)

while(True):
    print('Sending:')
    send_socket.sendto(b'hello from me', destination_address)
    time.sleep(2)

```

Import the socket module  
We will use the sleep function from the time module

Set the IP address of the machine to which we are sending

Create the socket

Set the destination address

Loop forever

Display a message

Send a message

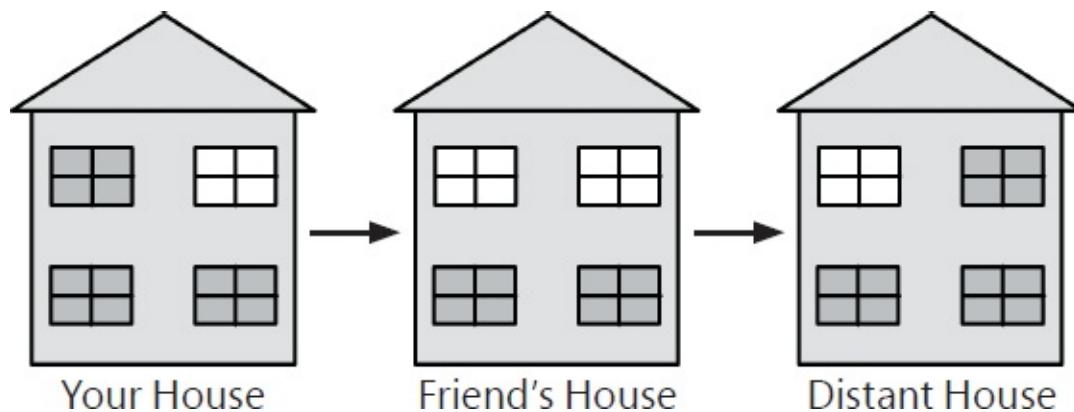
Sleep for two seconds

You will need to change the value of `target_ip` in the program to the address that was printed by the receiver program. When you run the sender program, you should see messages appearing on the screen of the receiver. You will have to interrupt them by pressing **Ctrl+C** or selecting **Shell, Interrupt Execution** from the IDLE menu.

## Route packets

The sample programs above worked for me because both computers were connected to my home network. However, not everything on the Internet is connected to the same network. My home network is different from the one operated by my next-door neighbor. The Internet is a very large number of separate “local” networks that are connected. To transmit messages from one network to another, we must introduce the idea of *routing*.

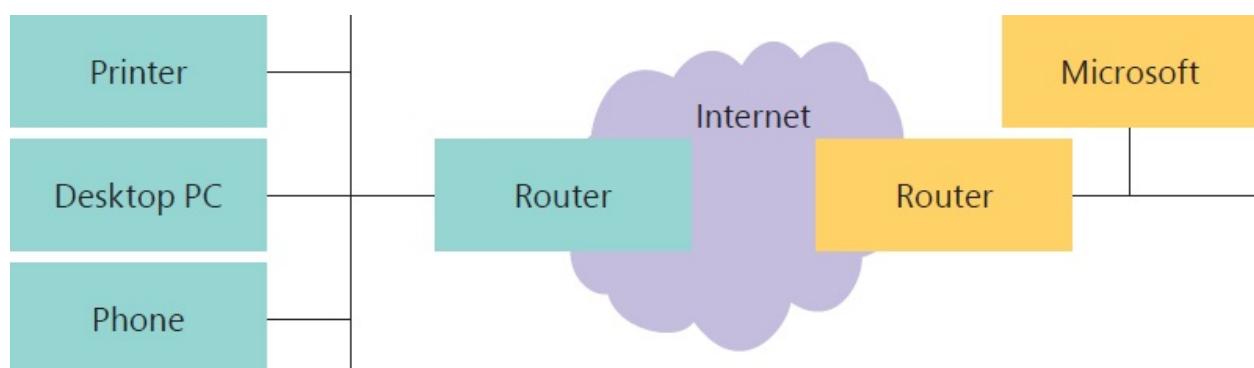
Going back to the bedroom light network we discussed earlier in this chapter, a friend who lives further down the street might not be able to see your bedroom light. However, she might be able to see the light from your friend’s house next door, so you could ask your friend next door to receive messages and then send them on for you. Your friend next door would read the address of the message coming in, and if it was for your friend on the next block, she would transmit it again. **Figure 14-3** shows how this works. Your friend uses the window on the left to talk to you and the window on the right to relay messages to your more distant friend.



**Figure 14-3** Routing from house to house to house

You can think of your friend in the middle as performing a routing role. She has a connection to both “networks”—the people you can see, and the people that your distant friend can see. She is therefore in a position to take messages from one network and resend them on the other one. Your connection to the Internet is managed by a *router*, which is a computer specially programmed to send and receive messages using the Internet protocols.

The diagram in **Figure 14-4** shows how this all fits together. The machines on the home network are directly connected. The Desktop PC can send pages straight to the printer. However, if the Desktop PC needs to load webpages from a web server at Microsoft, the requests for the pages must leave the local network and travel via the Internet. Messages that need to go off a local network are sent to the router, which forwards them to the Internet. The router is also responsible for receiving messages sent from the Internet to machines on the local network. The router will retransmit these messages onto the local network, addressed to the correct machine. This process is called network address translation, or NAT.



**Figure 14-4** Routing and the Internet



## WHAT COULD GO WRONG

### Network and firewall problems

I managed to use the sample programs **EG14-01 Receive packets on port 10001 from another machine** and **EG14-02 Send packets on port 10001 to another machine** to send messages from a Windows PC to an Apple Mac. I was asked by the Windows Firewall to allow Python programs to use the network, but once I did this, the programs worked fine.

A *firewall* is a component of the network management software in a computer connected to a network. It tries to make sure that programs are not using network connections improperly. If your computer becomes infected by a virus, it's the job of the firewall to stop the virus program from using your network connection to infect other computers. The firewall keeps a list of programs that are allowed to use the network. If the firewall detects network access from a program the firewall has not seen before, it will ask the user to confirm that the new program may use the network.



Once I selected Allow access in the above dialog, my network conversation worked fine. However, I had more difficulty sending messages from the Mac to the PC. If these programs don't work, your network might be restricting programs to a specific set of ports. These programs will also fail to work if the two machines are on separate networks.

## Connections and datagrams

The Internet provides two ways for systems to exchange information: connections and datagrams. A datagram is a single message sent from one system to another. The Python programs we created earlier use datagrams. However, you can also use the Internet to create connections between systems on the network. The *Transmission Control Protocol* (TCP) is used by the Internet to set up and manage connections between stations. You can find a good description of the protocol here:

[https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol).

When two systems are connected, they must perform extra work to manage the

connection itself. When one system sends a message that's part of a connection, the network either confirms that the message was successfully transferred (once the network has received an acknowledgment) or gives an error saying that it could not be delivered.

Connections are used when it's important that the entire message gets through. When your browser is loading a webpage, your computer and the web server share a connection across the network, which ensures that all parts of the webpage are delivered and that any failed pieces are retransmitted. The transmission, confirmation, and retransmission process means data is transported more slowly. Managing a connection places heavier demands on the systems communicating this way. You can regard a connection to another machine as much like the file object that we use to connect a program to a file. A program can call methods on a connection to send messages to the connection and check if anything has been received from the connection. The connection will remain open until it is closed by one of the systems using it.

## Networks and addresses

When we sent and received messages using the test programs above, we used addresses like 192.168.1.55. Earlier in this chapter, we said that these are called *Internet protocol*, or IP, addresses, and that you can think of them as the “telephone number” of a specific computer on a network.

However, nobody wants to have to remember an IP address like this. People would much rather use a name like [www.robmiles.com](http://www.robmiles.com) to find a site. To solve this problem, a computer on the Internet will connect to a name server, which will convert hostnames into IP addresses. The system behind this is called the *domain name system*, or DNS. A DNS is a collection of servers that pass naming requests around among themselves until they find a system with authority for a particular set of addresses that can match the name with the required address.

We can think of a name server as a kind of “directory inquiries” for computers. In days past, if I wanted to know the phone number of the local movie house, I would call for directory assistance. When a computer wants to know the IP address of a website, the DNS is queried.

## Consume the web from Python

The web is one of many services that use the Internet. When a browser wants to read a webpage, it sets up a connection to the server and requests the page content. The page content is expressed in Hypertext Markup Language, or HTML. The page content might contain references to images and sounds that are part of the webpage. The browser will set up connections to download these too and then draws the page for you on the screen.

## Read a webpage

If we wanted, we could write low-level, socket-based code to set up a TCP connection with a web server and then fetch the data back. However, this is such a common use for programs that the creators of Python have done this for us. The `urllib` module uses the Internet connection to talk to a web server and fetch webpages for our programs. The URL returns the webpage associated with it.

```
# EG14.03 Webpage reader

import urllib.request
url = 'https://www.robmiles.com'
req = urllib.request.urlopen(url)
for line in req:
    print(line)
```

The diagram illustrates the execution flow of the provided Python code. It consists of several horizontal arrows pointing from left to right, each connecting a specific line of code to a corresponding explanatory text box. The code is as follows:

```
# EG14.03 Webpage reader

import urllib.request
url = 'https://www.robmiles.com'
req = urllib.request.urlopen(url)
for line in req:
    print(line)
```

- An arrow points from the line `import urllib.request` to a box containing the text "Import the URL reader module".
- An arrow points from the line `url = 'https://www.robmiles.com'` to a box containing the text "This is the URL from which the program will read".
- An arrow points from the line `req = urllib.request.urlopen(url)` to a box containing the text "Create the web request object".
- An arrow points from the line `for line in req:` to a box containing the text "Work through the web request a line at a time".
- An arrow points from the line `print(line)` to a box containing the text "Print the line".

If you run this program, it will print the current contents of my blog page. There's a lot of it. The `urlopen` object uses HTTP to request the webpage and then returns an iterator that we can work through.

## Use web-based data

The ability to read from the web can be used for much more than just loading the text part of a webpage. We can also interact with many other data services. One such service is RSS (Really Simple Syndication, or Rich Site Summary, depending on which description you read), which is a format for describing web articles or blog posts. Lots of sites provide RSS feeds of their content, and programs can connect to and consume their content.

Point the webpage reader program above to <https://www.robmiles.com/journal/rss.xml> to download a document that contains my most recent blog posts. The document is formatted using a standard called XML (eXtensible Markup Language). The weather 563snaps that we used in [Chapter 5](#) also fetch the weather information from a web server. The program downloads the weather information from a server in the form of an XML document.

## The XML document standard

The XML standard allows us to create documents that can contain structured data. The documents are designed to be easy for computers and people to read. Programmers create an XML document to send data from one computer to another. An XML document contains a number of elements. Each element can have attributes, which are just like data attributes in a Python class. An element can also contain other elements. For a full description of XML, visit <https://en.wikipedia.org/wiki/XML>.

We can use the XML document returned by the RSS feed from my blog to investigate how XML works. Below, you can see a slightly abridged version of the RSS feed from my blog. I've removed some elements, but this shows the general format of the document (and the fact that I'm rather excitable in my blog posts).

```

<rss version="2.0">
  <channel>
    <title>
      robmiles.com
    </title>
    <item>
      <title>
        Water Meter Day
      </title>
      <category>Life</category>
      <description>
        <![CDATA[ We had a new water meter installed yay! ]]&gt;
      &lt;/description&gt;
    &lt;/item&gt;
    &lt;item&gt;
      &lt;title&gt;
        Python now in Visual Studio 2017
      &lt;/title&gt;
      &lt;category&gt;Python&lt;/category&gt;
      &lt;category&gt;Visual Studio&lt;/category&gt;
      &lt;description&gt;
        <![CDATA[ Python is now available in Visual Studio 2017 yay! ]]&gt;
      &lt;/description&gt;
    &lt;/item&gt;
  &lt;/channel&gt;
&lt;/rss&gt;
</pre>



The diagram illustrates the structure of the provided XML code. The root element is <rss>, which is labeled as the "RSS element in the document". Inside it is a <channel> element, labeled as the "Channel element in the RSS element". The <channel> element contains two <item> elements, each labeled as "Item in the blog". Each <item> element contains a <title> element, labeled as "Title of the item". The <item> elements also contain <category> and <description> elements. The <category> elements are labeled as "Category of the blog post" and the <description> elements are labeled as "Blog post content". The <description> elements also contain CDATA sections, which are highlighted in dark teal. The diagram uses horizontal lines to connect the XML code to its corresponding labels.


```

XML documents are organized into elements. An element has a name and can contain attributes (data about the element, just like a Python class attribute). The first element in the sample above is called `rss` and contains an attribute stating which version of RSS the element contains. This is used in the same way as the `version` attribute that we added to the `Contact` class in the Contacts manager we created in [Chapter 10](#). It tells programs the version of the RSS element; in the case of my blog, the version is number 2.0.

```
<rss version="2.0">
```

An element can contain other elements; they are enclosed between the `<name>` and `</name>` parts. Above, you can see that the `channel` element contains two `item` elements and that each `item` contains a `title` and a

`description` element.



## CODE ANALYSIS

### The XML document format

You might have some questions about the XML format.

**Question:** How do parent and child elements work in XML?

**Answer:** A given XML element can contain other elements. These are called *child* elements. Child elements can contain other child elements. In the RSS example above, the `channel` element contains two `item` elements as children. Each `item` has children, which are the `title` and `description` elements.

Don't get child elements confused with subclasses of superclasses. A subclass is used in a class hierarchy and picks up all the attributes of a superclass (sometimes confusingly called a "parent" class). We use subclasses to allow us to customize a superclass to better fit a particular situation. It is nothing to do with XML documents.

The best way to think of an XML child element is that it is an attribute of the element (such as a piece of data about the element), which is actually another XML element.

**Question:** What does CDATA mean?

**Answer:** When we put strings into a Python program, we can enclose them in triple quotes (''''). Text enclosed in triple quotes can span several lines of the program source and can contain any kind of quote characters. The `CDATA` element in an XML document works in the same way. Everything between the `<! [CDATA[` and the `]]>` items is treated as the text of that element. This behavior allows us to put entire blog posts inside an element in an XML

document.

**Question:** Why does the second item in the document contain two `category` elements?

**Answer:** XML doesn't necessarily enforce a standard on the content or organization of an XML document (although you can do this using a *schema* if you want to—but this is beyond the scope of this book). You can find out more about XML schema here:  
<http://www.xml.com/pub/a/2000/11/29/schemas/part1.html>.

The category elements of an item are used in the same way as we used the tags in the Fashion Shop application created in [Chapter 11](#). Readers can search for all my posts about Python, Visual Studio, or life. The RSS standard allows writers to tag an item with as many category elements as needed.

## The Python `ElementTree`

We could write a program that decodes the XML file, but it would be difficult work. Fortunately, Python provides the `ElementTree` class, which can be used to work with XML documents. A program can load an XML document in an instance of `ElementTree` and then call methods on the instance to navigate the document.

```
# EG14.04 Python ElementTree

import xml.etree.ElementTree as ElementTree      Import the module and give it a name

rss_text = '''                                The sample program holds all the text of the XML example
<rss version="2.0">
Sample RSS above goes here
</rss>
'''


doc = ElementTree.fromstring(rss_text)           Create an ElementTree instance from the RSS string

for item in doc.iter('item'):                    Iterate through all the item elements in the document
    title = item.find('title').text             Find the title element in the item and get the text out of it
    print(title.strip())                      Strip the title text of extra spaces and print it
    description = item.find('description').text  Find the description element in the
                                                item and get the text from it
    print('    ',description.strip())            Strip the title of extra spaces and print it
```

The **ElementTree** class provides a range of methods that you can use to find and work through elements in an XML document. The **iter** method is given the name of an element and will generate an iteration you can work through using a **for** loop. The **find** method will search a given element for any child elements with a particular name. The **text** attribute of an element is the actual text payload of the element. The output of the program is as follows:

[Click here to view code image](#)

Water Meter Day

We had a new water meter installed yay!  
Python now in Visual Studio 2017

Python is now available in Visual Studio 2017  
yay!

There are lots of other methods you can use to work with an XML document. You can even use the **ElementTree** class to allow you to edit the contents of elements, remove them, and even add new ones. However, you should be able to use the above methods to extract data items from XML feeds on the Internet. The sample program **EG14-05 RSS Feed reader** contains a few you can use to get started.



## MAKE SOMETHING HAPPEN

### Work with weather data

The weather snaps we used in [Chapter 5](#) decode an XML document from the U.S. Weather Service. The code to get the temperature for a given location is as follows:

```
# EG14.06 Weather Feed Reader

def get_weather_temp(latitude,longitude):
    address = 'http://forecast.weather.gov/MapClick.php'
    query = '?lat={0}&lon={1}&unit=0&lg=english&FcstType=dwml'.\
        format(latitude,longitude)
    req=urllib.request.urlopen(address+query)
    page=req.read()
    doc=xml.etree.ElementTree.fromstring(page)
    for d in doc.iter('temperature'):
        if d.get('type') == 'apparent':
            text_temp_value = d.find('value').text
    return int(text_temp_value)
```

The weather web server  
Web query containing the latitude and longitude  
Build a web request  
Read the text from the website  
Create an ElementTree from the text  
Work through all the temperature elements  
Is the type attribute of this element "apparent"?  
Get the content of the value element, which is a child element of this temperature  
Return an integer obtained from the text in the value element

You can find this function, along with a sample weather file that was returned by the server, in the folder **EG14-06 Weather Feed Reader** in the sample programs for this chapter. Try changing the methods so that you get the maximum and minimum temperatures and the forecast values.

## What you have learned

In this chapter, you discovered the fundamentals of network programming and how networks transfer data from one machine to another. You've seen that a protocol describes how systems can communicate and that the Internet uses

protocols that describe layers of different functionality, with hardware at the bottom and a software interface at the top. Information is sent between machines in messages called datagrams, and each machine has a unique IP address on a local network.

You saw that the Internet can be regarded as a large number of local networks that are connected. A device called a router will take datagrams addressed to remote sites (machines not connected to the local network) and send them to the Internet. Network connections can either be sent as individual, unacknowledged datagrams or as part of a connection. A given system can expose connections on one of a number of different *ports*. When a program wants to accept connections, it will bind a software *socket* to a port on the host machine and accept connections on that port.

Large amounts of data are transferred by the transmission of large numbers of datagrams. Python provides a socket class that can be used to control a network connection. You used a socket to perform simple communication between two Python programs. You also used the `urllib` Python module to connect to a web server and download the contents of webpages.

Finally, you've explored the eXtensible Markup Language (XML) and learned how to create `ElementTree` structures from XML documents and extract information from these documents.

Here are some points to ponder about networking.

### **Do wireless network devices use a different version of the Internet from wired ones?**

A wireless device uses a different medium from a wired device, but as far as the computer using the connection is concerned, the connections both work in the same way. The Internet protocols allow the *transport* method (the means by which data is moved between devices) to exist as a layer underneath other layers that set up and manage connections. We've done something similar with our software, when we had separate objects manage the storage of data in the Fashion Shop application in [Chapter 12](#). As long as the interface between the layers (the method by which one layer talks to another) is well defined, we can switch the component at one level of the layer with another component, and the rest of the system would still function.

## **How big can a datagram get?**

The *maximum transmission unit* (MTU) of a network is the largest message that can be sent in a single network transaction. The size of the MTU varies depending on the transmission medium used. You can find out the MTU values for various networks here:

[https://en.wikipedia.org/wiki/Maximum\\_transmission\\_unit](https://en.wikipedia.org/wiki/Maximum_transmission_unit)

## **Do all datagrams follow the same route from one computer to another?**

Not necessarily. The Internet is a huge collection of connected networks. A datagram may have to travel across several networks to get to its destination. Systems that route datagrams constantly decide on the best way to send them, based on how busy various parts of the network are and what connections are available. The Internet was originally designed to be used in a situation where parts of the network could suddenly stop working, so this rerouting behavior is built into how it works. It can lead to some strange effects. Sometimes a datagram sent after another can arrive before the first one. If we set up a connection using a socket, these effects are hiding from our program by the network.

## **Do all datagrams get to their destination?**

No. UDP packets are not guaranteed to arrive and are connectionless. TCP packets are part of a session and are guaranteed to arrive.

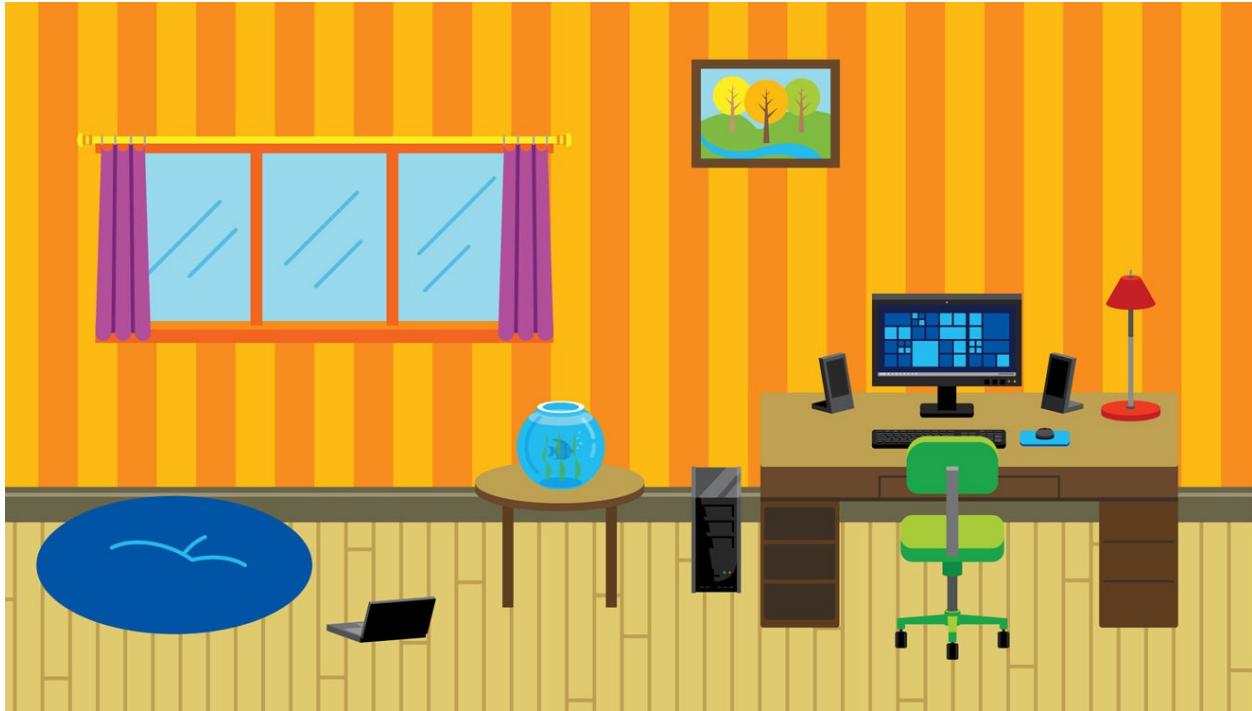
## **What is the difference between XML and HTML?**

XML and HTML are both *markup languages*. That's what the ML in both of their names means. HTML and XML look similar internally as they both use the same format for describing elements and attributes. XML is a standard that describes how to make any kind of document. I could design an XML document to hold football scores, or types of cheese, or anything else I want to manipulate and send to other computers. HTML is a markup language specifically for telling a web browser how to display a webpage. HTML contains elements that can describe the format of text, the position of images, and the color of the background, among other things. You can think of HTML as a kind of XML document specifically for webpages.

## **What is the difference between HTML and HTTP?**

HTML (Hypertext Markup Language) tells a browser what to display on the screen. HTTP (Hyper Text Transfer Protocol) is how the server and the browser move the page data (an HTML document) from the server into the browser. We'll see more of HTTP in the next chapter.

# 15 Python programs as network servers



## What you will learn

In this chapter, you'll learn how to create a Python program that will act as a server for network clients. You'll also discover how to make a Python program that responds to posts from users, and you'll create your first web application. This chapter will get you started creating solutions that use the web.

[Create a web server in Python](#)

[Host Python applications on the web](#)

[What you have learned](#)

## Create a web server in Python

The web works by using socket network connections, just like those we created in [Chapter 14](#). When we use a browser to connect to a web server, the basis of the communication is a socket. A server program listening to a socket connection will send back the page that your browser has requested.

In [Chapter 14](#), when we created a simple program to read webpages from a server, we noted that the appearance of webpages is expressed Hypertext Markup Language (HTML), and the conversation between a browser and a server is managed by a protocol called Hypertext Transfer Protocol (HTTP). In this section, we'll learn a bit more about the communication between a web server and a browser and create some web servers of our own.

## A tiny socket-based server

I've created a tiny Python program that provides a socket connection that you can connect to via a browser program on your computer. It serves out a tiny webpage that you can view. Let's look at the code:

```
# EG15-01 Tiny socket web server
```

```
import socket
```

Import the socket library

```
host_ip = 'localhost'
```

Use the localhost name for this server

```
host_socket = 8080
```

The server will listen on port 8080

```
full_address = 'http://'+host_ip+':'+str(host_socket)
```

Build a string that contains the server address

```
print('Open your browser and connect to: ', full_address)
```

Tell the user what to connect to

```
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Create the socket

```
listen_address = (host_ip, host_socket)
```

Create the address to listen on

```
listen_socket.bind(listen_address)
```

Bind the socket to the server address

```
listen_socket.listen()
```

```
connection, address = listen_socket.accept()
```

Wait for a request from a browser

```
print('Got connection from: ', address)
```

Indicate we have a connection

```
network_message = connection.recv(1024)
```

Get the network message

```
request_string = network_message.decode()
```

Decode the network message into the request string

```
print(request_string)
```

Print the request string

```
status_string = 'HTTP/1.1 200 OK'
```

HTTP status response

```
header_string = '''Content-Type: text/html; charset=UTF-8
```

HTTP response headers

```
Connection: close
```

```
'''
```

```
content_string = '''<html>
```

HTTP content

```
<body>
```

```
<p>hello from our tiny server</p>
```

```
</body>
```

```
</html>
```

```
'''
```

```
response_string = status_string + header_string + content_string
```

Build the complete response

```
response_bytes = response_string.encode()
```

Encode the response into bytes

```
connection.send(response_bytes)
```

Send the response bytes

```
connection.close()
```

Close the connection



MAKE SOMETHING HAPPEN

## Connect to a simple server

You can use the socket web server on your PC to explore how the web works. Use IDLE to open the example program **EG15-01 Socket web server** and get started.

When you run the program, it will display the address of the web server that has been created and is waiting for a web request. You should see a display like the one below.

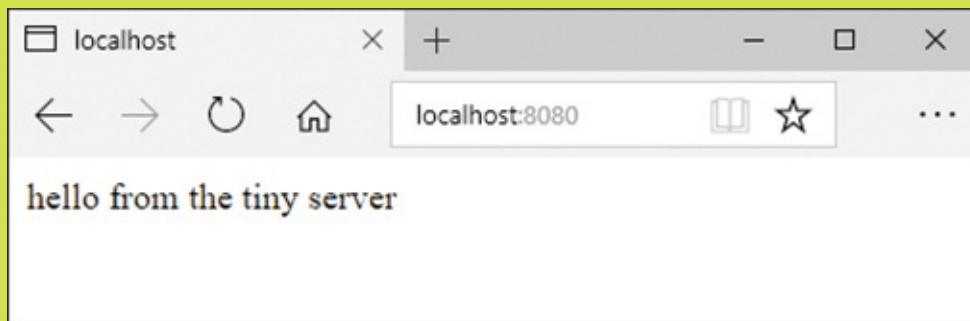
[Click here to view code image](#)

>>>

```
RESTART: C:/Users/Rob/EG14-03 Tiny socket  
web server.py
```

[Open your browser and connect to:  
http://localhost:8080](http://localhost:8080)

Now open your browser and connect to the address. The browser will connect to the socket from the server program and will display the webpage that it serves out:



If you now go back to IDLE, you should see the contents of the web

request made by the browser that's been printed.

[Click here to view code image](#)

```
>>>
RESTART: C:/Users/Rob/EG15-01 Tiny socket
web server.py
Got connection from: ('192.168.1.56', 51221)
GET / HTTP/1.1
Host: 192.168.1.56:8080
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/60.0.3112.113 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
>>>
```

The most important word on the page is the very first word of the message, **GET**, which is the beginning of the request for a webpage. The **GET** request is followed by information that the server uses to determine what kind of responses the browser can accept.



## CODE ANALYSIS

Web server program

**Question:** Previous sockets that we have created have used a socket type of `socket.SOCK_DGRAM`. Why is this program using a socket type of `socket.SOCK_STREAM`?

**Answer:** The programs we created in [Chapter 14](#) to send packets between computers sent individual datagrams using the User Datagram protocol (UDP). A datagram is very useful for sending quick messages to another computer. You can think of it as the network equivalent of a text message. When you send a text message, you have no way of knowing whether the message has been received. The browsers and servers on the web don't use datagrams to communicate; instead, they establish a network connection using the Transport Control Protocol (TCP) that allows them to exchange large amounts of data and ensure that the data has arrived. When a Python program creates a socket, it can identify that socket as using datagrams (`SOCK_DGRAM`) or a connection (`SOCK_STREAM`).

**Question:** What are the `status_string`, `header_string`, and `content_string` variables in the program used for?

**Answer:** The HTTP protocol defines how servers and browsers should interact. The browser will send a `GET` command to ask the server for a webpage. The server will send three items in its response. The first is a status response. If the page was found successfully, the status returned will be `200`, as in the contents of the variable `status_string` above. If the page is not found, the status returned will be `404`, which means "page not found."

The status information is followed directly by a header string that gives the browser information about the response. In the program above, the value assigned to `header_string` tells the browser that the content is text and that the network connection will be closed once the content has been delivered.

Finally, the server will send the HTML document that describes the webpage to be displayed. The content string is placed in the variable `content_string` in the program above. If you want to use this program to serve different content to the browser, just change the

text in `content_string`. These three strings are added together to create the complete response string.

**Question:** What are the `encode` and `decode` methods used for?

**Answer:** The `encode` method takes a string of text and encodes it as a block of bytes, ready for transmission over the network. The string type provides a method called `encode`, which will return the contents of a string encoded as a block of bytes. The program uses this method to encode the response string that the server sends to the browser:

[Click here to view code image](#)

```
response_bytes = response_string.encode()
```

The bytes type provides a method called `decode` that returns the contents of the bytes decoded as a string of text. The program uses this method to decode the command that the server receives from the browser.

```
request_string = network_message.decode()
```

The `network_message` contains the block of bytes received from the network, which is converted into the `request_string`. The tiny server always serves out the same message to the browser, but it could use the contents of the request to determine which page was being requested.

**Question:** Could browser clients connect to this server via the Internet?

**Answer:** This would only be possible if your computer was directly connected to the Internet, which is not usually the case. As we saw in [Chapter 14](#), a computer is normally connected to a local network, and the local network is connected via a router to the Internet. All the machines connected to a local network (whether it's a home, a school, or a hotel) could potentially connect to a server connected to

that network, but you would need to configure the router (which connects a local network to the Internet) to allow messages from the Internet to reach your computer if you want to serve out webpages to the Internet. This is not something that's normally permitted because it opens up a machine to attack from malicious systems on the Internet.

**Question:** How does the statement that gets the connection work?

**Answer:** The following statement gets the connection to the socket:

[Click here to view code image](#)

```
connection, address = listen_socket.accept()
```

This statement uses a form of method calling that we haven't used very often. It's explained at the end of [Chapter 8](#), in the descriptions of tuples. The `accept` method returns a *tuple* that holds the connection and address values of the system that has connected. We can assign these values directly to variables by using the statement above. The `connection` object is like the object we use when we open a file. We can call methods on the connection object to read messages sent by the program at the other end of the network connection. We can also call methods on the connection to send messages to the distant machine.

**Question:** How could I make the sample program above into a proper web server?

**Answer:** We would have to add a loop so that the web server would return to waiting for connections once it had finished dealing with a request. A "proper" web server would also be able to support multiple web requests at the same time. The socket mechanism can accept more than one connection at the same time, and Python allows the creation of threads that can run simultaneously on a computer. However, we wouldn't want to create our own web server, as the developers of Python have already done this for us. We'll use their server in the next section.

## Python web server

We know that a web server is just a program that uses the network to listen for requests from clients. We could create a complete web server by building on the tiny server we've just created, but it turns out that Python provides ready-built classes that we can use to do this. The `HTTPServer` class allows us to create objects that will accept connections on a network socket and dispatch them to a class that will decode and act on them.

The `BaseHTTPRequestHandler` class provides the basis of a handler for incoming web requests that our server receives. We can use the `HTTPServer` and `BaseHTTPRequest Handler` classes to create a web server as shown in the example code below. You can use a browser to connect to this server in the same way as the one we wrote above, but this server does not stop after the first request; it will continue to accept connections and serve out the website until the program is stopped.

```
# EG15-02 Python web server

import http.server
Get the server module

class WebServerHandler(http.server.BaseHTTPRequestHandler):
Create a subclass of the
BaseHTTPRequestHandler
    class

    def do_GET(self):
        ...
        This method is called when the server receives
        a GET request from the client
        It sends a fixed message back to the client
        ...
        self.send_response(200)
        Send a 200 response (OK)
        self.send_header('Content-type', 'text/html')
        Add the content type to the header
        self.end_headers()
        Send the header to the browser

        message_text = """<html>
<body>
<p>hello from the Python server</p>
</body>
</html>
"""
        Text of the webpage to be sent to the browser
        message_bytes = message_text.encode()
        Encode the HTML string into bytes

        self.wfile.write(message_bytes)
        Write the bytes back to the browser
    return

host_socket = 8080
Socket number for this server
host_ip = 'localhost'
Use localhost as the network address

host_address = (host_ip, host_socket)
Create the host address

my_server = http.server.HTTPServer(host_address, WebServerHandler)
Create a server
my_server.serve_forever()
Start the server
```



## CODE ANALYSIS

Python server program

**Question:** How does this work?

**Answer:** You can think of the `HTTPServer` class as the dispatcher for incoming requests, a bit like a receptionist at a large company. An employee of a company could tell the receptionist “If anyone asks for me, I’m in the board room.” When we create the `HTTPServer`, we tell it “If any web requests come in, create an instance of `WebServerHandler` to deal with them.”

When a request comes in, the `HTTPServer` creates a `WebServerHandler` and adds all the attributes that describe the incoming request. The server then looks through the incoming request and calls the method in the `WebServerHandler` that matches the request that’s been made. The handler we created above can only handle `GET` requests as it only contains a `do_GET` method.

**Question:** What does the `WebServerHandler` class do?

**Answer:** The `WebServerHandler` class is a subclass of a superclass called `BaseHTTPRequestHandler`. A subclass of a superclass inherits all the attributes of the superclass and can add attributes of its own. The `WebServerHandler` above contains one attribute, which is the method called `do_GET`. The `do_GET` method will run when a browser tries to get a webpage from our server; the `do_GET` method returns the webpage requested by the browser. We can create different server behavior by changing what the `do_GET` method does. We can also make a handler that responds to other HTTP messages by adding more methods to the handler class (covered later in this chapter).

**Question:** How does the server program send the page back to the host?

**Answer:** The connection to the host takes the form of a file connection. When the `WebServerHandler` instance is created, it is given an attribute called `wfile`, which is the write file for this web request. The `do_GET` method can use the `wfile` attribute to write back the message to the server.

[Click here to view code image](#)

```
self.wfile.write(message_bytes)
```

The variable `message_bytes` contains the message the server is returning. Using a file in this way makes it very easy for a server to send back any kind of information, including images.

**Question:** How is the `WebServerHandler` class connected to the server?

**Answer:** When we create the server, we pass the server a reference to the class that it will use to respond to incoming web requests.

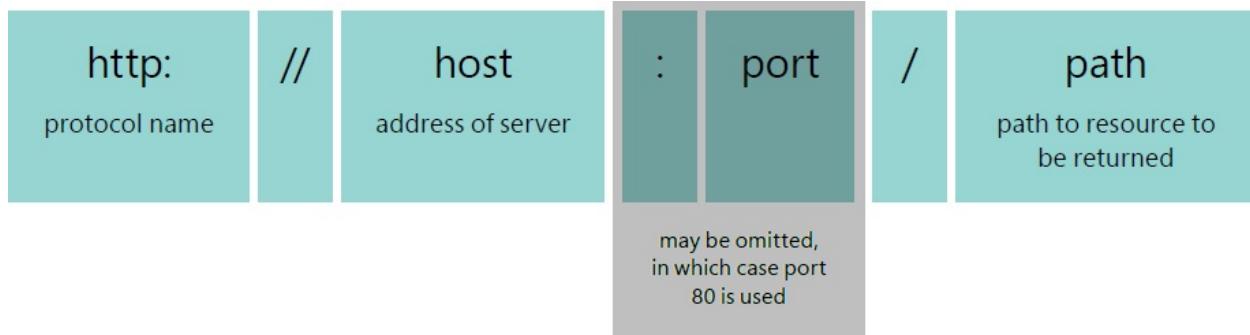
[Click here to view code image](#)

```
my_server =  
http.server.HTTPServer(host_address,  
WebServerHandler)
```

Above is the statement that constructs the server. Note that the second argument to the call is `WebServerHandler`. When the server receives a request from a browser, it creates an instance of the `WebServerHandler` class and then calls methods in that instance to deal with the request.

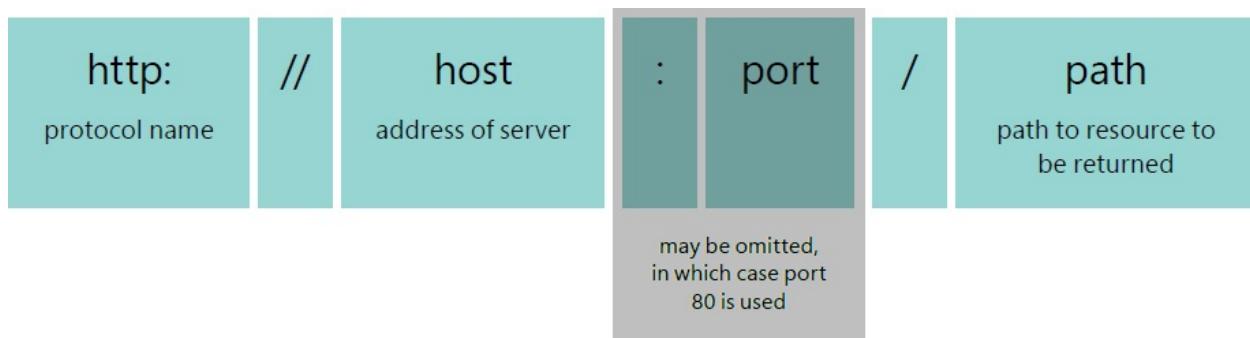
## Serve webpages from files

The web servers we've created so far are not very useful because they just serve out the same information. However, we know that a single web server can serve out many different pages. Browsers and servers on the World Wide Web use a *Uniform Resource Locator*, or *URL*, string to identify destinations, which includes a *path* to the resource that will be provided. **Figure 15-1** shows the anatomy of a URL.



**Figure 15-1** Anatomy of a Uniform Resource Locator (URL)

The URL of a host contains the protocol to be used, the network address of the server, the socket to be used for the connection to the server, and the path to the page on the server. The URL for the webpage that contains a description of how URLs are constructed is shown in **Figure 15-2**.



**Figure 15-2** URL example

This shows that the path to a resource can include folders. In the path shown, the requested page is in the folder `WD-htm140-970917`, which is held in the folder `TR`. This URL does not include a socket because the server is using port 80. If the port address is left out, the browser will use port number 80, which is the Internet port associated with the web. We've been using port 8080 for the web servers on our local machine.

A server can extract the path information from the `GET` request and send back the page that was requested. If the path is left out, the server will send back the “home” page for that location. A server can use the path to determine which file to return to the browser. The very first web servers were used to serve files of text that were stored on them. Below is a web request handler that serves out files.

```

# EG15-03 Python webpage server
class WebServerHandler(http.server.BaseHTTPRequestHandler):

    def do_GET(self):
        """
        This method is called when the server receives
        a GET request from the client
        It opens a file with the requested path
        and sends back the contents
        """

        self.send_response(200)           Send a 200 response (OK)
        self.send_header('Content-type', 'text/html') Tell the browser the content is text
        self.end_headers()               Finish sending the header

        # trim off the leading / character in the path
        file_path = self.path[1:]         Get the file name from the path
                                         supplied in the GET request

        with open(file_path, 'r') as input_file: Open the file
            message_text = input_file.read()   Read the file

        message_bytes = message_text.encode() Encode the file into a block of bytes

        self.wfile.write(message_bytes)     Write the file back to the browser

    return

```

## Extract slices from a collection

The code above uses *slicing*, which is something we haven't seen before. Python programs can extract slices from collections. **Figure 15-3** shows how we would express a slicing action.



**Figure 15-3** Anatomy of a slice

The start and end positions of the slice are given in square brackets, separated by a colon character. We can see how this works by slicing my name, which can be regarded as a collection of individual characters.

[Click here to view code image](#)

```
>>> 'Robert'[0:3]  
'Rob'
```

The statement above creates a slice from my full name. It starts at the character at the beginning of my name (with the index 0) and ends at the character “e” (with the index 3). Note that the “terminating” character is not included in the slice. Here’s another slice:

```
>>> 'Robert'[1:2]  
'o'
```

The statement above just extracts the “o” from my name. It starts at the character with the index of 1 and ends at the character with the index of 2 (but does not include the “b”). Here’s another example:

```
>>> 'Robert'[:4]  
'Robe'
```

If I leave out the start position, the slice starts at the start of the collection, as shown above. If I leave out the end position, as shown below, the slice continues to the end of the string.

```
>>> 'Robert'[2:]  
'bert'
```

I can also use negative numbers in my slices, in which case the number is used as an index from the end of the collection:

```
>>> 'Robert'[-2:-1]  
'r'
```

The above slice starts two positions in from the end of the string, and ends one position in from the end of the string, which means that it just slices off the letter “r.”

You can use slicing on any Python collection, including a tuple. Note that slicing doesn’t affect the item being sliced, it just returns a “slice” of that item.

The program above uses slicing to get rid of a leading `/` character on the `path` attribute in the `WebServerHandler` object. The statement below would convert “`\index.html`” to “`index.html`” by creating a slice that contains everything but the first character of the string. The web server can then use this as the name of the file to be opened and returned.

```
file_path = self.path[1:]
```



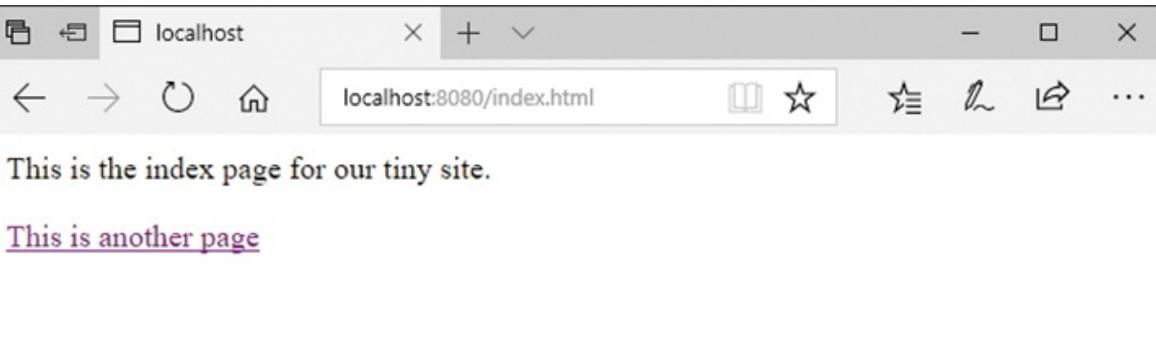
MAKE SOMETHING HAPPEN

## Connect to a file server

We can use the web server above to browse a tiny website. Use IDLE to open the example program **EG15-03 Python webpage server** in the folder **EG15-03 Python webpage server** in the sample programs folder for this chapter. The folder also contains two HTML pages that the server will return to the browser. They are called `index.html` and `page.html`.

Start the program and open the following address with your browser:

`http://localhost:8080/index.html`



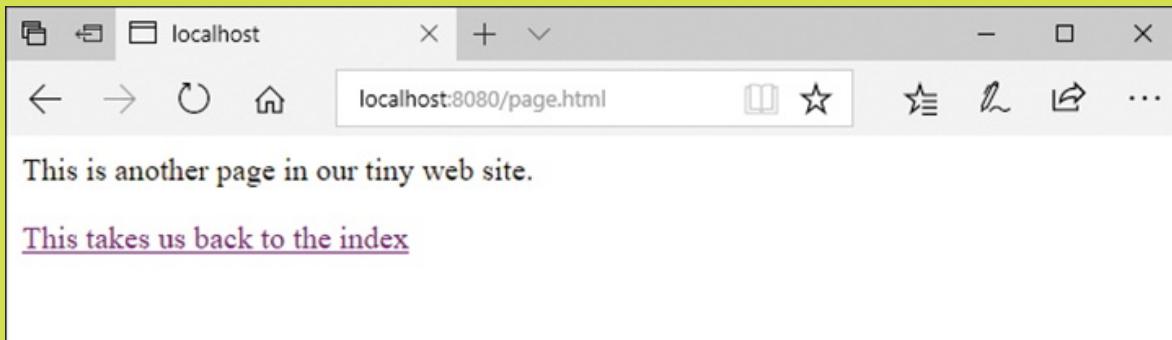
The browser will show the first page of our site.

[Click here to view code image](#)

```
<html>
<body>

```

This is the HTML file for the index page. It contains the text you can see on the page, along with a link to a second page. When you click the link, the browser will load the next page and display it.



You can click the link on this page to return to the index.

This is the HTML for the second page of our tiny website:

[Click here to view code image](#)

```
<html>
<body>
<p>This is another page in our tiny website.
</p>
<a href="index.html">This takes us back to
the index</a> </body>
</html>
```

By now you should have a good understanding of how a web server works and how we can use Python to create them. We could extend our web server above to serve out image files and handle the situation when a browser tries to load a file that doesn't exist, but the Python libraries provide a web server handler called **SimpleHTTPRequestHandler** that can be used to serve out files. Below is a program that uses this handler to create what must be one of the tiniest web servers you can build.

```
# EG15-04 Full Python webpage server

import http.server

host_socket = 8080                         Respond to web requests on port 8080
host_ip = 'localhost'                       Use the localhost address

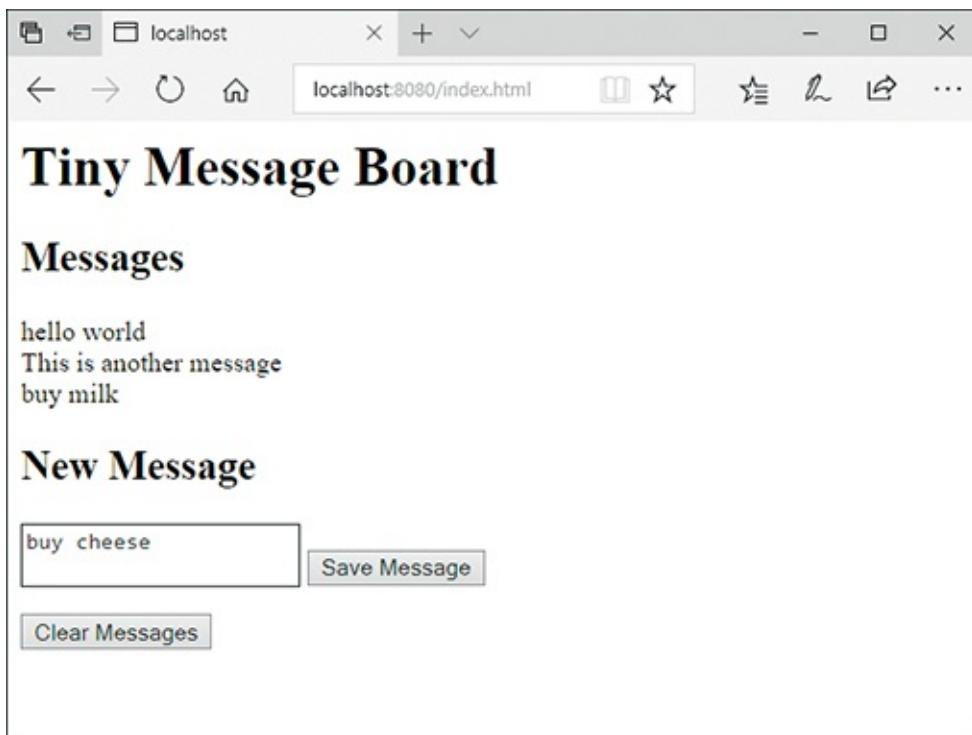
host_address = (host_ip, host_socket)        Create the host address

my_server = http.server.HTTPServer(host_address,      Address for the server
                                    http.server.SimpleHTTPRequestHandler)
my_server.serve_forever()                   Request
                                            handler
                                            class
```

## Get information from web users

We can use the Python servers we've created to provide information to users. Next, we'll see how our users can send information back to the Python program. To show how this works, we'll create a Tiny Message program. Anyone can write messages into the program for other readers to see via their browser.

**Figure 15-4** shows the user interface for this message board. The user can type in messages and click Save Message to add a message in the list. Also, the user can click Clear Messages to clear all the messages from the board.



**Figure 15-4** Tiny Message Board

**MAKE SOMETHING HAPPEN**

Use a message board

The best way to learn what this program will do is to try it. Use IDLE to open the example program **EG15-05 Web message board** in the sample programs for this chapter. Start the program and open the following address with your browser:

*<http://localhost:8080/index.html>*

You should see the message board display. Enter a message into the text

area underneath the New Message heading and click the **Save Message** button. The page will refresh, and the message will be displayed in the Messages part of the page. If you add a second message, you will see it appear below the first one. If you click **Clear Messages**, all the messages will be removed from the screen. Now that you know what the program does, we can investigate how the program does it.

## The HTTP POST request

Hypertext Transport Protocol, or HTTP, describes how the web browser and the web server communicate. It defines a series of browser requests. Until now, the only HTTP request that our server has responded to is the **GET** request, which is a request to get a webpage. There are several other browser requests, such as the **POST** request, which allows a browser to post information back to the server.

[Click here to view code image](#)

```
<form method="post">
    <textarea name="message"></textarea>
    <button id="save" type="submit">Save
        Message</button>
</form>
```

This is the Hypertext Markup Language (HTML) that describes the part of the webpage used to submit a new message. The browser will generate a text input area and a Save button that looks like **Figure 15-5**.



**Figure 15-5** Text entry

The HTML tells the browser to perform a **POST** request when the user clicks the Save Message button. The message sent with the **POST** request will include the contents of the text area.

We can create a `do_POST` method in our HTTP request handler class that will deal with a `POST` request.

```
def do_POST(self):
    length = int(self.headers['Content-Length'])           Get the length of the reply from the browser
    post_body_bytes = self.rfile.read(length)               Read the reply into a block of bytes
    post_body_text = post_body_bytes.decode()              Convert the block of bytes into a text string
    query_strings = urllib.parse.parse_qs(post_body_text,   Convert the text into a dictionary of query items
                                           keep_blank_values=True) Allow blank values
                                                       in the query string
    message = query_strings['message'][0]                  Extract the message from the query string
    messages.append(message)                             Add the message to the existing messages

    self.send_response(200)                             Send the OK response
    self.send_header('Content-type', 'text/html')        Tell the browser it is getting text back
    self.end_headers()                                  Send the headers

    message_text = self.make_page()                     Call a method to build the webpage to send back
    message_bytes = message_text.encode()               Encode the webpage into a block of bytes
    self.wfile.write(message_bytes)                    Send the bytes to the browser
```



## CODE ANALYSIS

### POST handler

The `POST` handler method is quite complicated, although it is not very long. You might have a few questions about how it works. When trying to work out what is happening, remember what the method has been written to do. The user has filled in a form on the webpage and pressed the Save Message button. The browser has assembled a response that includes the text the user entered and sent this back to the server as a `POST` request.

The **POST** request has arrived at the server, which has created an instance of the **webServerHandler** class to deal with this **POST** and then called the **do\_POST** method in this class to deal with the **POST**.

**Question:** How does **do\_POST** read the information sent by the browser?

**Answer:** The message being posted by the browser can be read via a file connection. The first statement of the **do\_POST** method determines the length of the file by reading the **Content-Length** item from the message header sent by the browser.

[Click here to view code image](#)

```
length = int(self.headers['Content-Length'])
```

The headers are provided in the **webServerHandler** as a dictionary (called **headers**), from which a program can load header items by name. The statement above gets the **Content-Length** header and then converts it into an integer, which is then used to read in the response:

[Click here to view code image](#)

```
post_body_bytes = self.rfile.read(length)
```

The variable **post\_body\_bytes** refers to a block of bytes that contain the response from the browser. Next, the method converts these bytes into a string using the **decode** method:

[Click here to view code image](#)

```
post_body_text = post_body_bytes.decode()
```

Now we have the text that the browser is sending back to the server. This text is presented by the browser in the form of a *query string*,

which is a way that HTTP encodes named items. Items in a query string are given in the form:

```
name=item
```

The name of the item will be the name of the `textarea` being sent back; in this case, the name is “message,” which you can see in the HTML for the page above. Python provides a method that converts query strings into a dictionary, which saves us from having to write our own code to process query strings.

[Click here to view code image](#)

```
query_strings =  
urllib.parse.parse_qs(post_body_text,  
keep_blank_values=True)
```

The `parse_qs` method creates a dictionary that contains a key for each named item in the query string. It has been given an extra argument to tell it to add blank query string values to the dictionary; we will use this when we add the `clear` command later.

Now that we have our query strings, we can extract the content of the `textarea` from the response:

[Click here to view code image](#)

```
message = query_strings['message'][0]
```

The `parse_qs` method creates a list of items for each key, so the statement above takes the item at the start of this list (which is the text we want) and sets the variable `message` to this. So, at this point, the variable `message` contains the text that the webpage user has entered. Now we just need to add the text to the messages that the

program is storing.

```
messages.append(message)
```

The variable `messages` is declared as a global variable, and it is a list that holds each of the entered messages. The `make_page` method uses the list of messages to create a webpage, which is returned to the browser.

**Question:** How does the `get_POST` method generate the webpage that contains the messages the user entered?

**Answer:** The `get_POST` method above extracts the message from the `POST` from the browser and adds it to a list of messages. It then calls the `make_page` method to create a webpage that includes these messages. Next, we'll investigate this method.

A server must send a webpage in response to a `POST` request from a browser. Sometimes this webpage contains the message, “Thank you for submitting the information,” but our message program will just redraw the webpage with the new message included. The `webPageHandler` class contains a method, `make_page`, that does this. The `make_page` method is called in the `do_GET` and `do_POST` methods.

```
def make_page(self):
    all_messages = '<br>'.join(messages) → Create a list of strings separated by the <br>
    page = '''<html>
<body>
<h1>Tiny Message Board</h1>
<h2>Messages</h2>
<p> {0} </p> → Placeholder for the list of messages
<h2>New Message</h2>
<form method="post">
    <textarea name="message"></textarea>
    <button id="save" type="submit">Save Message</button>
</form>
<form method="post">
    <button name="clear" type="submit">Clear Messages</button>
</form>
</body>
</html>'''
    return page.format(all_messages)
```



## CODE ANALYSIS

### Make a webpage from Python code

We've seen that a web server can send the contents of a file back to the browser client. It can also create HTML (HyperText Markup Language) text and send this back. The `make_page` method constructs a page of HTML that contains the input text area as well as the buttons. It also contains all the messages that have been entered. You might have some questions.

**Question:** How does this method create a list of messages?

**Answer:** The HTML format needs to be told when to end a line of text displayed on a webpage. The HTML command to do this is `<br>` (which is short for “line-break”). The `make_page` method uses `join` (which we first saw in [Chapter 10](#) when we used it to make a string containing a list of Time Tracker sessions) to create a

string containing a list of messages separated by the `<br>` command.

**Question:** How does this method insert the message list into the HTML that describes the page?

**Answer:** The method uses Python string formatting. It contains the placeholder `{0}` for a value to be inserted into the page. The string containing the messages, which was created using `join`, is entered as the value.

The final element of the application that we need to implement is the Clear button, which can be used to clear all the elements in the message list. We can add a clear behavior to the `do_POST` method by checking for certain elements in the query string returned by the browser.

```
if 'clear' in query_strings:  
    messages.clear()  
  
elif 'message' in query_strings:  
    message = query_strings['message'][0]  
    messages.append(message)
```

Has the user clicked on Clear?  
If Clear clicked, clear the messages  
Has the user clicked on Save Message?  
If Save Message clicked, save the message

The `in` operator returns `True` if a given dictionary contains a particular key. The code above checks to see if the clear entry is in the dictionary. If you look in the HTML returned by the `make_page` method above, you'll see that the "Clear Messages" button has been given the name `clear`.

## Host Python applications on the web

The web applications we've created in this chapter have been hosted on our own computers, and we've used the special port number 8080. In theory, we could host these programs on a machine connected to the Internet and make them available for anyone to use. However, while writing our own client and server applications has given us a good understanding of how the web works, it turns out that there are much better ways to create web applications using Python than by writing them from scratch as we've been doing. Some existing Python frameworks give you a head start in creating web applications. I strongly

recommend that you look at Flask ([flask.pocoo.org](http://flask.pocoo.org)) and Django ([djangoproject.com](http://djangoproject.com)) These frameworks hide a lot of the low-level network access and provide access to databases and components that make it very easy to produce good-looking websites underpinned with Python code.

Once you've created your Python web application, you will need to find a place on the Internet to host it so that it's available to your users. Find out more about how to use Azure to host your applications at <https://azure.microsoft.com/en-us/develop/python/>.

## What you have learned

In this chapter, you discovered how to create Python programs that serve out webpages in response to requests from web browsers. You looked at the HTTP protocol used to manage web requests and saw that there are numerous web requests, including the **GET** request, to load a page. You saw that the **POST** request is used to post data back to a server. You saw that the server response contains a status line, a header element, and a content element. You discovered that Python provides a helper class called **HTTPServer** that can manage a web server and also a class **BaseHTTPRequestHandler** that can be used as the starting point for making programs that respond to web requests.

You created a simple message board application that responds to **GET** and **POST** requests and learned that the basis of web applications is creating programs that respond to these and other requests from the browser.

Here are some points to ponder about Python and web servers.

### **Is this how webpages work?**

The original world wide web worked in the same manner as the programs we created in this chapter. A web server delivered pages of data (which were loaded from files of text) in response to requests from a browser. However, the web today is slightly more complicated. Modern webpages contain program code, usually written in a language called JavaScript. The program code in the webpages interacts with the user and sends requests to programs running on the server. The actual layout and appearance of webpages that the user sees are expressed using “style sheets” that are acted on by the browser when a page is

displayed. However, a solid understanding of the concepts described in this chapter and [Chapter 14](#) will serve as a very good starting point for web development.

## **Can a web server determine what kind of client program is reading the webpage?**

Yes. The header sent by the browser contains details of the browser type and even the kind of computer and operating system being used.

## **Can a web server have a conversation with a user?**

You can think of a request from a web browser as a question. The server then provides the answer; however, this is not a conversation. Each question and answer is an individual transaction. When two people are talking, they will establish a context for their conversation. If you and I were talking about a particular type of computer and you asked me, “How fast is it?” I’d remember that we were talking about computers and give the appropriate answer. HTTP does not work on the basis of a conversation like this.

However, websites can use “cookies” to establish a conversation with a user. A cookie is a tiny piece of data that the web server gives the browser. The cookie is stored on the client computer, and at a later time the server can request the cookie so that it can retrieve context information. Cookies are used to implement things like shopping carts, and to allow a website to discover the identity of a user. However, they are also somewhat contentious in that they allow websites to track users in ways that the user might not be aware of.

## **How can I make my website secure?**

The webpages we’ve created so far have been insecure. The messages exchanged between the browser and the server are sent as plain text. The free program Wireshark, which you can download from [www.wireshark.org](http://www.wireshark.org), can be used to capture and view network messages.

To counter against network eavesdroppers, modern browsers and servers *encrypt* the data they’re transferring. Encryption is the process of converting the plain text messages into data that only makes sense when it has been decrypted by the receiver.

Encrypted websites use the protocol name https (rather than http) and they also connect via port 443 rather than port 80. If you want to create a secure, web-based application, you should look at the two previously suggested frameworks, Flask and Django, as they provide support for these kinds of sites. These also provide support for user authentication.

# 16 Create games with pygame



## What you will learn

Writing games is great fun. Unlike “proper” programs, games are not always tied to a formal specification and don’t need to do anything useful. They just must be fun to play. Games are a great place to experiment with your software. You can write code just to see what happens when it runs, and see whether the result is interesting. Everyone should write at least one computer game in their lives. In this chapter, you’ll start creating games. You’ll learn how to make a complete game and finish with a framework you can use to create more games of your own design.

[Getting started with pygame](#)

[Draw images with pygame](#)

[Get user input from pygame](#)

[Create game sprites](#)

[Complete the game](#)

[What you have learned](#)

## Getting started with pygame

In this section, we'll get started with pygame, and we'll create some shapes and display them on the screen. The free pygame library contains lots of Python classes you can use to create games. The snaps functions we used in the early chapters of this book were written using pygame, so you should have already loaded pygame onto your computer (see [Chapter 3](#) for instructions).

Note that the pygame library makes use of tuples to create single-data items that contain colors and coordinates that describe items in the games. If you're not sure what a tuple is, read the description of tuples in [Chapter 8](#) before you work through the following "Make Something Happen."



### MAKE SOMETHING HAPPEN

#### Start pygame and draw some lines

The best way to understand how pygame works is to start it up and draw something. Open the Python Command Shell in IDLE to get started. Before we can use pygame in a program, we need to import it; enter the statement below and press **Enter**:

```
>>> import pygame
```

Once we've imported the pygame module, we can start using the functions and classes it contains. The pygame framework needs to be set up before you can use it to display the items in your game. A game program does this by calling the `init` function in the pygame module,

as shown here:

```
>>> pygame.init()
```

When you press Enter, the `init` function sets up the different pygame elements, each of which performs a specific task when the game is running. Elements read user input, make sounds, and so on. The `init` function returns a tuple that tells you how many elements have been successfully initialized, and how many have failed to initialize. If an element fails to initialize, pygame might not have been installed correctly. However, most games ignore this value and assume that all is well.

```
>>> pygame.init()  
(6, 0)
```

The display above shows that six modules have been set up correctly and that none have failed to initialize. If you see any failures—in other words, if the second value in the tuple is any value other than zero—you should make sure that pygame has been properly installed.

Next, we need to create a drawing surface. A drawing surface has a specific size, which is set when we create it. The size is given in pixels (a pixel is the size of a dot on the display). The more pixels you have, the better quality the display. You also find pixel dimensions when talking about camera and video screen resolution. We'll use a screen size of 800 pixels wide and 600 pixels high. We can use a tuple to create a surface as follows:

```
>>> size = (800, 600)
```

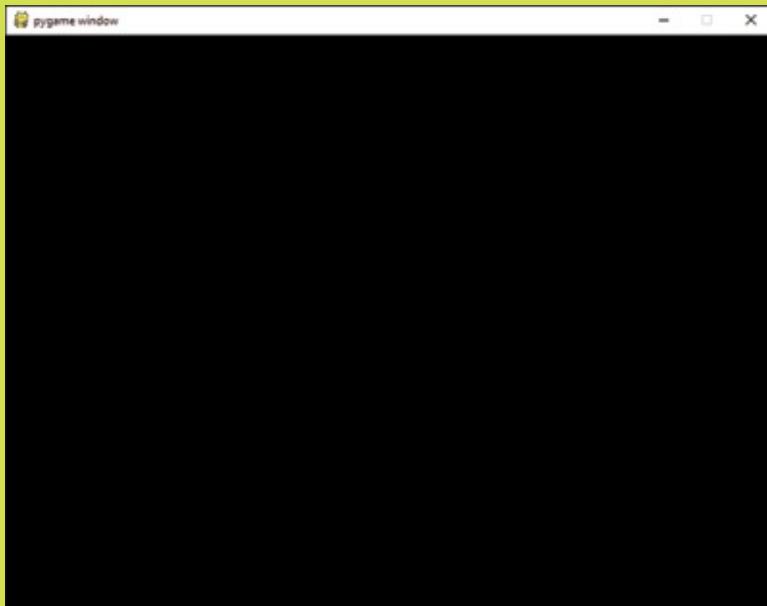
Remember that a tuple is a way of grouping a number of items. You can find out more about them in [Chapter 8](#). Once we have the tuple that describes the size of the game screen, we can use this value as an

argument to the function that creates a pygame drawing surface.

[Click here to view code image](#)

```
>>> surface = pygame.display.set_mode(size)
```

This statement creates the drawing surface, sets the variable `surface` to refer to it, and then displays the surface on the screen. You should see the window below appear on your screen.



You can change the title of the drawing window using the following function:

[Click here to view code image](#)

```
>>> pygame.display.set_caption('An awesome game by Rob')
```

This function changes the title of the window as shown below.



Now we can draw things on the surface, so we'll start by drawing some lines. The line drawing function in pygame accepts four parameters:

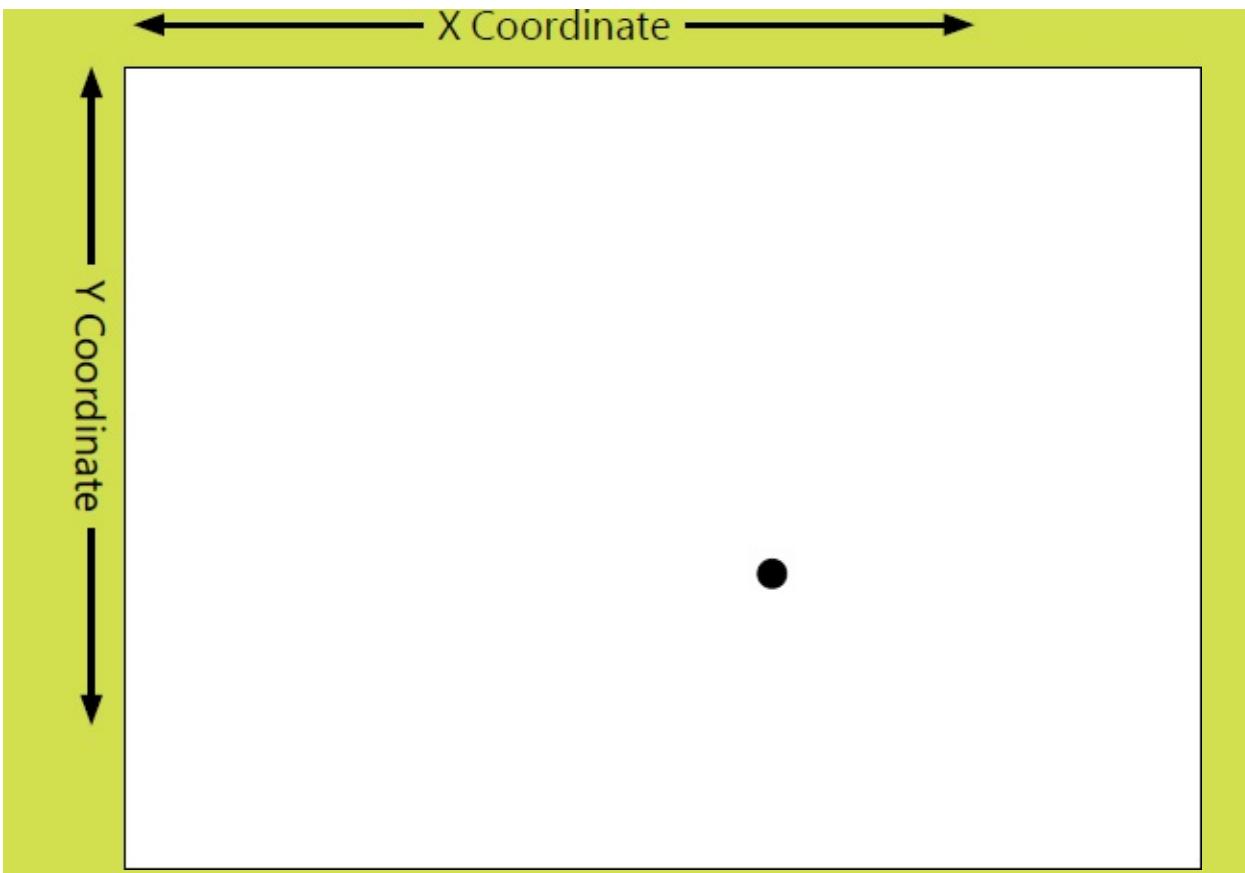
- The surface on which to draw
- The drawing color
- The start position of the line
- The end position of the line

Let's assemble these items. We've already created the surface, so we can just use that. The color of an item in pygame is expressed as a tuple containing three values. We first saw this mechanism for expressing color in [Chapter 3](#) when we used the snaps framework to draw text. Each value in the tuple represents the amount of red, green, and blue, respectively. The lowest level is 0; the highest level is 255. If we want to draw a red line, we can create a tuple that contains all the red and none of the other two primary colors. Enter the following tuple:

```
>>> red = (255, 0, 0)
```

Now we can set the start position of the line. For a given position on the screen, the value of x specifies how far the position is from the left edge, and the value of y specifies how far down the screen from the top edge. A specific location is expressed as a tuple containing the values (x, y). The figure below shows how pygame coordinates work. The important thing to remember is that the *origin*, which is the point with the coordinate (0,0) is the top left corner of the display. Increasing the value of x moves you toward the right of the screen, and increasing the value of y will move you down the screen.

This might not be how you expect graphics to work. Most graphs that you draw have their origins in the bottom left, and increasing y moves up. However, placing the origin in the top left corner is standard practice when drawing graphics on a computer.



Bearing this in mind, let's draw a line from the origin on the screen to the position (500,300). We can create some tuples that hold these values. Type in these two statements to set the start and end position of the line.

[Click here to view code image](#)

```
>>> start = (0,0)
>>> end = (500, 300)
```

Now we can issue our drawing instruction. Type in the following call to the `line` function in the pygame `draw` module:

[Click here to view code image](#)

```
>>> pygame.draw.line(surface, red, start,
end)
```

When you press **Enter**, the line is drawn, and the line function returns a rectangle object that encloses this line:

[Click here to view code image](#)

```
>>> pygame.draw.line(surface, red, start,
end)
<rect(0, 0, 501, 301)>
```

We'll ignore the values returned from the drawing methods. Unfortunately, if you look at the game window, you won't see any lines on the screen. Draw operations take place on the *back buffer* managed by pygame. We don't draw directly on the screen because we don't want the player to see each individual draw action. Instead, we perform all our drawing operations on a piece of memory in the computer (called the back buffer). When the drawing is finished, we copy this piece of memory onto the display memory. The memory that used to be displayed becomes the new back buffer, and the process starts again.

In pygame, the `flip` function swaps the display memory and the back-buffer memory. We need to call `flip` to make a line appear on the screen, so type the call below and press **Enter**.

[Click here to view code image](#)

```
>>> pygame.display.flip()
```

This call will cause a red line to appear on the game display, as shown on the next page.



If you don't want a black background, you can use the `fill` function to fill the screen with a chosen color. These three statements create a tuple that describes the color white, fills the back buffer with white, and then flips the back buffer to display the white screen.

[Click here to view code image](#)

```
>>> white = (255, 255, 255)
>>> surface.fill(white)
>>> pygame.display.flip()
```

If you do this, you'll notice that the red line we created has been erased.

We can use these functions to create some nice-looking images. The program below draws 100 colored lines and 100 colored dots. The program uses functions

that create random colors and positions on the display area.

```
#EG 16.01 pygame drawing functions

import random          The demo uses random numbers
import pygame           The demo uses pygame

class DrawDemo:         Class to contain our demo program

    @staticmethod      Make the method static since we should need to create a demo class
    def do_draw_demo(): Method to demonstrate pygame drawing
        init_result = pygame.init() Initialize pygame
        if init_result[1] != 0: If the number of failures is not zero, we have a problem
            print('pygame not installed properly') Display a message
            return Abandon the demonstration

        width = 800          Set the width of the screen
        height = 600          Set the height of the screen
        size = (width, height) Set the size of the game display

        def get_random_coordinate(): Function to get a random coordinate
            X = random.randint(0, width-1) Get a random X value
            Y = random.randint(0, height-1) Get a random Y value
            return (X, Y) Return a tuple made from X and Y

        def get_random_color(): Function to get a random color
            red = random.randint(0, 255) Get a random red value
            green = random.randint(0, 255) Get a random green value
            blue = random.randint(0, 255) Get a random blue value
            return (red, green, blue) Return a tuple made from red, green, and blue

        surface = pygame.display.set_mode(size) Create the game surface
        pygame.display.set_caption('Drawing example') Set the window caption
```



```
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
black = (0, 0, 0)
yellow = (255, 255, 0)
magenta = (255, 0, 255)
cyan = (0, 255, 255)
white = (255, 255, 255)
gray = (128, 128, 128)
```

Create some color tuples

```
# Fill the screen with white
surface.fill(white)
```

```
# Draw 100 random lines
for count in range(100):
```

```
    start = get_random_coordinate()
    end = get_random_coordinate()
    color = get_random_color()
    pygame.draw.line(surface, color, start, end)
```

```
# Draw 100 dots
dot_radius = 10
for count in range(100):
    pos = get_random_coordinate()
    color = get_random_color()
    radius = random.randint(5, 50)
    pygame.draw.circle(surface, color, pos, radius)
```

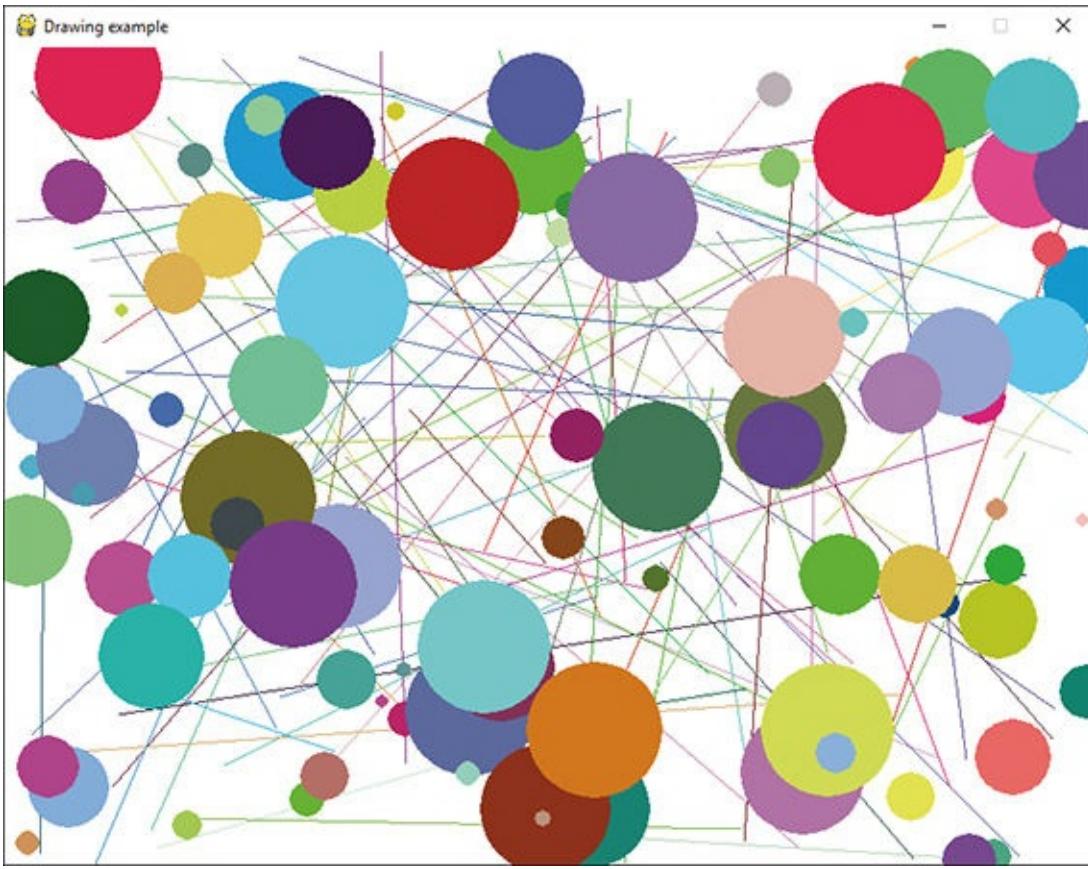
```
pygame.display.flip()
```

Flip the drawn elements to the display memory

```
DrawDemo.do_draw_demo()
```

Call the do\_draw\_demo method in the DrawDemo object

When I ran the above program, the display appeared as shown in **Figure 16-1**:



**Figure 16-1** Drawing dots and lines

When you run the program, you'll get an image that looks similar but will have a completely different arrangement of lines and circles because your program will get a different sequence of random numbers from the ones produced when I ran the program.



## MAKE SOMETHING HAPPEN

### Making art

You could create a program that displays a different pattern every now and then. You could use the time of day and the current weather conditions to determine what colors to use in the pattern and create a display that changes throughout the day (perhaps with bright primary

colors in the morning and more mellow and darker colors in the evening). If the weather is warm, the colors could have a red tinge, and if it's colder, you could create colors with more blues. Remember that you can create any color you like for your graphics by choosing the amount of red, green, and blue it should contain.

## Draw images with pygame

Pygame can also draw images on the screen. The images are loaded from files stored on your computer. You've already used the `display_image` function from the snaps library to draw images; now you'll discover how to use pygame to load and display images.

### Image file types

There are a number of different formats for storing pictures on computers. When working with Pygame, your pictures should be in one of these two formats:

- PNG—The PNG format is *lossless*, meaning it always stores an exact version of the image. PNG files can also have transparent regions, which is important when you want to draw one image on top of another.
- JPEG—The JPEG format is *lossy*, meaning the image is compressed in a way that makes it much smaller, but at the expense of precise detail.

The games you create should use JPEG images for the large backgrounds and PNG images for smaller objects drawn on top of them.

If you have no usable pictures of your own, you can use the ones I've provided with the sample files for this chapter, but the games will work best if you use your own pictures.

**Figure 16-2** shows my picture of the cheese we'll be using in the game that we will create. In the game, the player will control the cheese and use it to catch crackers around the screen. You can use another picture if you wish. In fact, I strongly advise that you do. I've saved the image in the PNG file format with a width of 50 pixels, which will work with the size of the screen we're using.



**Figure 16-2** The cheese

If you need to convert images into the PNG format, you can load an image using the Microsoft Paint program and then save it in this format. With Paint, you can also scale and crop images if you want to reduce the number of pixels in the image. For more advanced image manipulation, I recommend the program Paint.Net, which is free here: [www.getpaint.net](http://www.getpaint.net). Another great image manipulation program is Gimp, which is available for most machines. You can download Gimp from [www.gimp.org](http://www.gimp.org).

## Load an image into a game

The pygame library contains a function called `load` that loads an image. The image to be loaded is identified by its file name. The `load` function searches the local folder for the file. In other words, it looks in the folder from which the program is running. We saw this behavior in [Chapter 8](#) when we wrote programs to store and load data using files. The statement below loads an image from a file. The variable `cheeseImage` is set to refer to the image that's been loaded.

[Click here to view code image](#)

```
cheeseImage = pygame.image.load('cheese.png')
```

Now that we have an image loaded, we can draw it on the display. When an

image is drawn, the data that describes the image is copied into the memory used for the display. Game developers call this *blitting* the graphics data onto the screen. The pygame library contains a method called **blit** that's used to copy an image into display memory. The **blit** method requires two pieces of information to work:

- The image to be drawn
- The coordinates on the screen where the image is to be blitted

Let's put our cheese image at the top left corner of the display. The statement below creates a tuple that describes this position. The values of the x and y coordinates are both zero.

```
cheesePos = (0, 0)
```

We can now call the **blit** method to actually draw the cheese. The **blit** method is provided by the display surface that we created when our game program started.

[Click here to view code image](#)

```
surface.blit(cheeseImage, cheesePos)
```

The complete program that draws the cheese on the screen can be found below:

```
# EG16-02 Image Drawing

import pygame

class ImageDemo:

    @staticmethod
    def do_image_demo():
        init_result = pygame.init()           Initialize pygame
        if init_result[1] != 0:
            print('pygame not installed properly') End the method if pygame fails
            return                                to start

        width = 800
        height = 600
        size = (width, height)               Set the size of the display

        surface = pygame.display.set_mode(size) Get the pygame drawing surface

        pygame.display.set_caption('Image example') Sets up the pygame display

        white = (255, 255, 255)
        surface.fill(white)                  Clear the screen to white

        cheeseImage = pygame.image.load('cheese.png') Load the cheese image
        cheesePos = (0,0)                   Set the cheese position to the top left corner of the screen
        surface.blit(cheeseImage, cheesePos)   Draw the cheese
        pygame.display.flip()               Flip the display memory so that the cheese is displayed

ImageDemo.do_image_demo()
```

When we run this program, it draws some cheese on the screen as shown in **Figure 16-3**. Note that the drawing position for an image when we blit it onto the screen is the top left corner of that image.



**Figure 16-3** Cheese on the screen

## Make an image move

The **blit** function is given the draw position for an image. We can make an image appear to move by repeatedly drawing the image at different positions.

```
# EG16-03 Moving cheese

cheeseX = 40
cheeseY = 60                                         Set the start position for the cheese

clock = pygame.time.Clock()                           Create a pygame clock instance

for i in range(1,100):
    clock.tick(30)                                    Move the cheese 100 times
    surface.fill((255,255,255))                      Pause the game so that we have 30 frames per second
    cheeseX = cheeseX + 1                            Fill the screen with white
    cheeseY = cheeseY + 1                            Increase the x position of the cheese
    cheesePos = (cheeseX,cheeseY)                     Increase the y position of the cheese
    surface.blit(cheeseImage, cheesePos)              Create a cheese position tuple
    pygame.display.flip()                            Blit the cheese onto the screen
                                                Flip to the back buffer to update the display
```



### MAKE SOMETHING HAPPEN

#### Move an image

We can investigate the way that games make objects appear to move by using the **EG16-03 Moving cheese** program. When you use IDLE to run it, you should find that the cheese moves majestically down the screen for a while and then stops. The speed of the movement is controlled by the *frame rate* of the game. The frame rate is the rate at which the screen is redrawn, expressed as the number of frames per second (fps). The pygame **Clock** class provides a tick method that is given the number of frames per second required by the game. The program creates a new clock before it starts moving the cheese around.

```
clock = pygame.time.Clock()
```

The `Clock` class provides a set of time management methods that games can use. We'll use the `tick` method that allows us to make the game run at a constant speed. Without the clock, our game would run as fast as Python can execute the program, which would be impossible to play.

```
clock().tick(30)
```

The `tick` method will pause the game until the start of the next frame "slot." Find the above statement in the program and change the value from 30 to 60. The program will now update the screen 60 times per second. Run the program, and you'll find that the cheese moves twice as fast as it did before because the tick method is now allowing 60 frames per second.

If you change the frame rate to 5 (5 frames per second), you'll find that the cheese moves slowly and you'll be able to see each movement.

A player will get a good game experience if the game updates at 60 frames per second. Games on smaller devices—for example, mobile phones and tablets—might use lower frame rates to save battery power.

## Get user input from pygame

Now that we can move items around the screen under program control, the next thing we need is a way that a player can interact with the game. A game receives input from the user by means of pygame *events*. An event is a user action—for example, pressing a keyboard key or moving the mouse. We first saw these kinds of events when we created a graphical user interface using Tkinter in [Chapter 13](#). When we wanted to receive events in Tkinter, we bound a method to an event. When the event occurred, the method was called.

In pygame, events are managed differently. While a pygame program is running, the pygame system captures input events and places them in a queue. The game program must check the event queue regularly to see if there are any actions to which the program must respond. The events we're interested in are keyboard events generated when a key is pressed or released.



## MAKE SOMETHING HAPPEN

### Investigate events in pygame

We can look at how events work in pygame by creating some events and seeing the results. Open the Python IDLE Command Shell and type in the following statements to create a pygame window:

[Click here to view code image](#)

```
>>> import pygame  
>>> pygame.init()  
(6, 0)  
>>> size = (800, 600)  
>>> surface = pygame.display.set_mode(size)
```

Now use your mouse to click in the window that pygame has opened and press a few keys. Each key press will generate an event that will be captured by pygame. Now we can create a loop to look at the events that have been stored. Go back to IDLE and enter the following:

[Click here to view code image](#)

```
>>> for e in pygame.event.get():  
    print(e)
```

The `get` method returns a collection of events. This loop will print all

the events in the pygame event queue. When you enter an empty line after the `print` statement, you'll see all the event information:

[Click here to view code image](#)

```
>>> for e in pygame.event.get():
    print(e)

<Event(17-VideoExpose {})>
<Event(16-VideoResize {'size': (800, 600),
'w': 800, 'h': 600})>
<Event(1-ActiveEvent {'gain': 0, 'state':
1})>
<Event(2-KeyDown {'unicode': 'r', 'key': 114,
'mod': 0, 'scancode': 19})>
<Event(3-KeyUp {'key': 114, 'mod': 0,
'scancode': 19})>
<Event(2-KeyDown {'unicode': 'o', 'key': 111,
'mod': 0, 'scancode': 24})>
<Event(3-KeyUp {'key': 111, 'mod': 0,
'scancode': 24})>
<Event(2-KeyDown {'unicode': 'b', 'key': 98,
'mod': 0, 'scancode': 48})>
<Event(3-KeyUp {'key': 98, 'mod': 0,
'scancode': 48})>
<Event(1-ActiveEvent {'gain': 1, 'state':
1})>
>>>
```

Each event is described by a dictionary that holds information about the event. If you look through the events above, you'll see that the R, O, and B keys have been pressed and released in turn.

As the game runs, the event queue must be checked to see if any commands have been entered that should cause objects on the screen to move. We want the cheese to move while an arrow key is held down and then stop moving when the

key is released. The code below does this. Also, this code contains a test that causes the game to end when the player presses the Escape (Esc) key.

```
# EG16-04 Steerable cheese

cheeseX = 40
cheeseY = 60
cheeseYSpeed = 2
cheeseMovingUp = False
cheeseMovingDown = False
clock = pygame.time.Clock()
while True:
    clock.tick(60)
    for e in pygame.event.get():
        if e.type == pygame.KEYDOWN:
            if e.key == pygame.K_ESCAPE:
                pygame.quit()
                return
            elif e.key == pygame.K_UP:
                cheeseMovingUp = True
            elif e.key == pygame.K_DOWN:
                cheeseMovingDown = True
            elif e.type == pygame.KEYUP:
                if e.key == pygame.K_UP:
                    cheeseMovingUp = False
                elif e.key == pygame.K_DOWN:
                    cheeseMovingDown = False
        if cheeseMovingDown:
            cheeseY = cheeseY+cheeseYSpeed
        if cheeseMovingUp:
            cheeseY = cheeseY-cheeseYSpeed
    Clear the flag that indicates the cheese is moving down
```

Set the cheese's initial position  
Set the speed of the cheese movement  
Cheese is not moving up  
Cheese is not moving down  
Create a clock  
Repeatedly perform the game loop  
Wait for the next frame start  
Work through the events  
Does the event describe a key-down event?  
Is the key the Escape key?  
Shut down pygame  
If Escape has been pressed, exit the game loop  
Is the key the Up arrow?  
Set the flag that indicates the cheese is moving up  
Is the key the Down arrow?  
Set the flag that indicates the cheese is moving down  
Does the event describe a key up event?  
Is the key the Up arrow?  
Clear the flag that indicates the cheese is moving up  
Is the key the Down arrow?  
Move the cheese down the screen  
Is the cheese moving up?  
Move the cheese up



## CODE ANALYSIS

### Game loops

The code above is an example of a “game loop.” You may have some questions about it.

**Question:** What is the variable `e` used for in the program?

**Answer:** The variable `e` contains each event that the game loop is checking. The game is interested only in events generated when a key is pressed or released. When a key press is detected, the program checks to see which key was pressed. If the key is the Up Arrow, the code sets the flag to indicate that the cheese should move up; if the key is the Down Arrow, the code sets the flag to indicate that the cheese should move down. The game loop also contains tests that will clear the flag if a key is released.

**Question:** Why does the cheese move when I hold a key down?

**Answer:** Remember that the statements in the game loop are being repeated 60 times a second. So, every sixtieth of a second, the program is updating the position of the cheese. If a key is down, the cheese will be moved each time around the game loop. Currently, the `cheeseYspeed` is 2, which means that in a second the cheese will move 120 pixels.

**Question:** How do we change the speed of the cheese?

**Answer:** The variable `cheeseYspeed` gives the speed of the cheese in the y direction (up and down the screen). If we want to make the cheese move faster, we can increase the value of this variable.

**Question:** Why do we increase the value of y to move the cheese down the screen?

**Answer:** This is because the coordinate system used by pygame places the origin (the point where the values of x and Y are zero) at the top of the screen. Increasing the value of y will move the cheese down the screen.

**Question:** What would happen if the player pressed both the Up and the Down Arrow keys at the same time?

**Answer:** The cheese would be moved both up and then down again when it was updated. The result of this would be that the cheese would not appear to move, which is what we want the game to do.

**Question:** What would happen if the player moved the cheese right off the screen?

**Answer:** You can run the sample program to find out what happens. Drawing an image off the screen will not cause the game program to fail, but the object will not be visible. If we want to stop the cheese from moving off the screen, we will need to add code to make sure that the cheese is never positioned off the screen.

**Question:** What does the `pygame.quit()` method do?

**Answer:** The `pygame.quit()` method is called when the user presses the Escape key to finish a game; it closes pygame and causes the game window to be closed.

## Create game sprites

The game we'll create will display three different object types on the screen:

- **Cheese**—The player will steer the cheese around the screen.
- **Crackers**—The player will try to capture the cheese on the cracker.
- **Killer tomato**—The tomato will chase the cheese.

Each of these screen objects is called a *sprite*. You can think of a sprite as an image that is part of the game display. We will create a `Sprite` class that has an image drawn on the screen, a position on the screen, and a set of behaviors. Each sprite will do the following things:

- Draw itself on the screen.
- Update itself. If the sprite is the cheese, it will move in response to player input; if the sprite is the killer tomato, it will chase the cheese.

- Reset itself. When we start a new game, we must put the sprite in its starting position.

Sprites might have other behaviors, too, but these are the fundamental things that a sprite must do. We can put these behaviors into a class:

```

class Sprite:
    This will be the superclass for all sprites in the game
    ...
    A sprite in the game. Can be subclassed
    to create sprites with particular behaviors
    ...
    def __init__(self, image, game): Called to set up the values in a sprite
        ...
        Initialize a sprite
        image is the image to use to draw the sprite
        default position is origin (0,0)
        game is the game that contains this sprite
        ...
        self.image = image Store the image in the sprite
        self.position = [0, 0] Set the position in the sprite to the top left corner
        self.game = game Store the game reference in the sprite
        self.reset() Reset the sprite

    def update(self): Called when a sprite is to be updated
        ...
        Called in the game loop to update
        the status of the sprite.
        Does nothing in the superclass
        ...
        pass

    def draw(self): Called to ask a sprite to draw itself
        ...
        Draws the sprite on the screen at its
        current position
        ...
        self.game.surface.blit(self.image, self.position)

    def reset(self): Called to ask a sprite to reset itself
        ...
        Called at the start of a new game to
        reset the sprite
        ...
        pass

```



## CODE ANALYSIS

## Sprite superclass

The code above defines the superclass for all the sprites in the game. You may have some questions about it.

**Question:** What is the `game` parameter used for in the initializer?

**Answer:** When the game creates a new sprite, it must tell the sprite which game it is part of because some sprites will need to use information stored in the game object. For example, if the cheese manages to capture a cracker, the score value will need to be updated.

Programmers say that the sprite class and the game class will be tightly *coupled*. Changes to the code in the `CrackerChaseGame` class might affect the behavior of sprites in the game. If the programmer of the `CrackerChaseGame` class changes the name of the variable that keeps the score from `score` to `game_score`, the `Update` method in the Cheese class will fail when the player captures a cracker. A lot of coupling between classes in a large system is a bad idea, but in the case of our game it makes the development much easier, so I think it's reasonable to make the program work in this way.

**Question:** Why are the `update` and `reset` methods empty?

**Answer:** You can think of the `Sprite` class as a template for subclasses. Some of the game elements will need methods to implement `update` and `reset` behaviors. The cheese will need a `reset` method that places it in the middle of the screen at the start of the game. The cheese will need an `update` method that moves it around the screen. The `cheese` class will be a subclass of `Sprite`, and adds its own version of these methods.

**Question:** How does the `draw` method work?

**Answer:** The `draw` method is called to ask the sprite to draw itself on the screen.

[Click here to view code image](#)

```
def draw(self):
    ...
    Draws the sprite on the screen at its
    current position
    ...
    self.game.surface.blit(self.image,
self.position)
```

The game that the sprite is part of contains an attribute called `surface`, which is the pygame drawing surface for this game. The above method finds the game attribute from the sprite that's drawing itself. The game attribute was set when the sprite was created; the game attribute uses the game's `surface` property to blit the sprite image onto the screen.

The `Sprite` class doesn't do much, but it can be used to manage the background image for this game. The game will take place on a "tablecloth" background. We can think of this as a very large sprite that fills the screen. We can now make our first version of the game that contains a game loop that just displays the background sprite.

```

class CrackerChase:
    ...
    Plays the amazing cracker chase game
    ...

def play_game(self):
    ...
    Starts the game playing
    Will return when the player exits
    the game.
    ...
    init_result = pygame.init()           Initialize pygame
    if init_result[1] != 0:
        print('pygame not installed properly')
        return                                Quit if pygame is not installed on this machine

    self.width = 800
    self.height = 600                     Set the width and height of the game display
    self.size = (self.width, self.height)   Create a tuple that defines the screen size

    self.surface = pygame.display.set_mode(self.size)      Create the drawing surface
    pygame.display.set_caption('Cracker Chase')            Set the caption for the game screen
    background_image = pygame.image.load('background.png')
    self.background_sprite = Sprite(image=background_image,
                                    game=self)          Tell the sprite the game it is part of
    clock = pygame.time.Clock()                    Create the game for the clock
    while True:                                     Game loop that runs forever
        clock.tick(60)                            Ensure the game updates 60 times per second
        for e in pygame.event.get():
            if e.type == pygame.KEYDOWN:
                if e.key == pygame.K_ESCAPE:
                    pygame.quit()                  Close the game screen
                    return                        Return from the game method
            self.background_sprite.draw()         Ask the background to draw itself
            pygame.display.flip()                Flip the back buffer to the front
    
```

If the key pressed is Escape, return from the game method

Create the background sprite

Load the background image



## CODE ANALYSIS

## Game class

The code above defines the class that will implement our game. You might have some questions about it.

**Question:** How does the game pass a reference to itself to the sprite constructor?

**Answer:** We know that when a method in a class is called, the `self` parameter is called to reference the object within which the method is running. We can pass `self` into other parts of the game that need it:

[Click here to view code image](#)

```
self.background_sprite =  
Sprite(image=background_image, game=self)
```

The code above makes a new `Sprite` instance and sets the value of the game argument to `self` so that the sprite now knows which game it is part of.

**Question:** Why does the game call the `draw` method on the sprite to draw it? Can't the game just draw the image held inside the sprite?

**Answer:** This is a very important question, and it comes down to responsibility. Should the sprite be responsible for drawing on the screen, or should the game do the drawing? I think drawing should be the sprite's job because it gives the developer a lot more flexibility.

For instance, adding smoke trails to some of the sprites in this game by drawing "smoke" images behind the sprite would be much easier to do if I could just add the code into the "smoky" sprites rather than the game having to work out which sprites needed smoke trails and draw them differently.

**Question:** Does this mean that when the game runs the entire screen

will be redrawn each time, even if nothing on the screen has changed?

**Answer:** Yes. You might think that this is wasteful of computer power, but this is how most games work. It is much easier to draw everything from scratch than it is to keep track of changes to the display and only redraw parts that have changed.

The code below shows how we would start a game running:

```
# EG16-05 background sprite  
  
game = CrackerChase()  
game.play_game()
```

Create a game instance  
Start the game running

## Add a player sprite

The player sprite will be a piece of cheese that is steered around the screen. We've seen how a game can respond to keyboard events; now we'll create a player sprite and get the game to control it. The [Cheese](#) class below implements the player object in our game.

```

class Cheese(Sprite):
    """
    Player-controlled cheese object that can be steered
    around the screen by the player
    """

    def reset(self): Override the reset method in the sprite superclass
        """
        Reset the cheese position and stop any movement
        """
        self.movingUp = False Stop the cheese moving up
        self.movingDown = False Stop the cheese moving down
        self.position[0] = (self.game.width - self.image.get_width()) / 2
        self.position[1] = (self.game.height - self.image.get_height()) / 2
        self.movement_speed=[5,5] Set the initial move speed for the cheese
        self.position[1] = (self.game.height - self.image.get_height()) / 2 Center the cheese down the screen
        self.position[0] = (self.game.width - self.image.get_width()) / 2 Center the cheese across the screen

    def update(self):
        """
        Update the cheese position and then stop it moving off
        the screen.
        """

        if self.movingUp: If we are moving up, move the cheese up
            self.position[1] = self.position[1] - (self.movement_speed[1])
        if self.movingDown: If we are moving down, move the cheese down
            self.position[1] = self.position[1] + (self.movement_speed[1])

        if self.position[0] < 0: Stop movement off the left edge of the screen
            self.position[0]=0
        if self.position[1] < 0: Stop movement off the top of the screen
            self.position[1]=0
        if self.position[0] + self.image.get_width() > self.game.width: Stop movement off the right of the screen
            self.position[0] = self.game.width - self.image.get_width()
        if self.position[1] + self.image.get_height() > self.game.height: Stop movement off the bottom of the screen
            self.position[1] = self.game.height - self.image.get_height()

    def StartMoveUp(self): Called to start the cheese moving up the screen
        'Start the cheese moving up'
        self.movingUp = True Set the up movement flag to True

    def StopMoveUp(self): Called to stop the cheese moving up the screen
        'Stop the cheese moving up'
        self.movingUp = False Set the up movement flag to False

    'Other cheese movement methods go here...'

```



## CODE ANALYSIS

### Player sprite

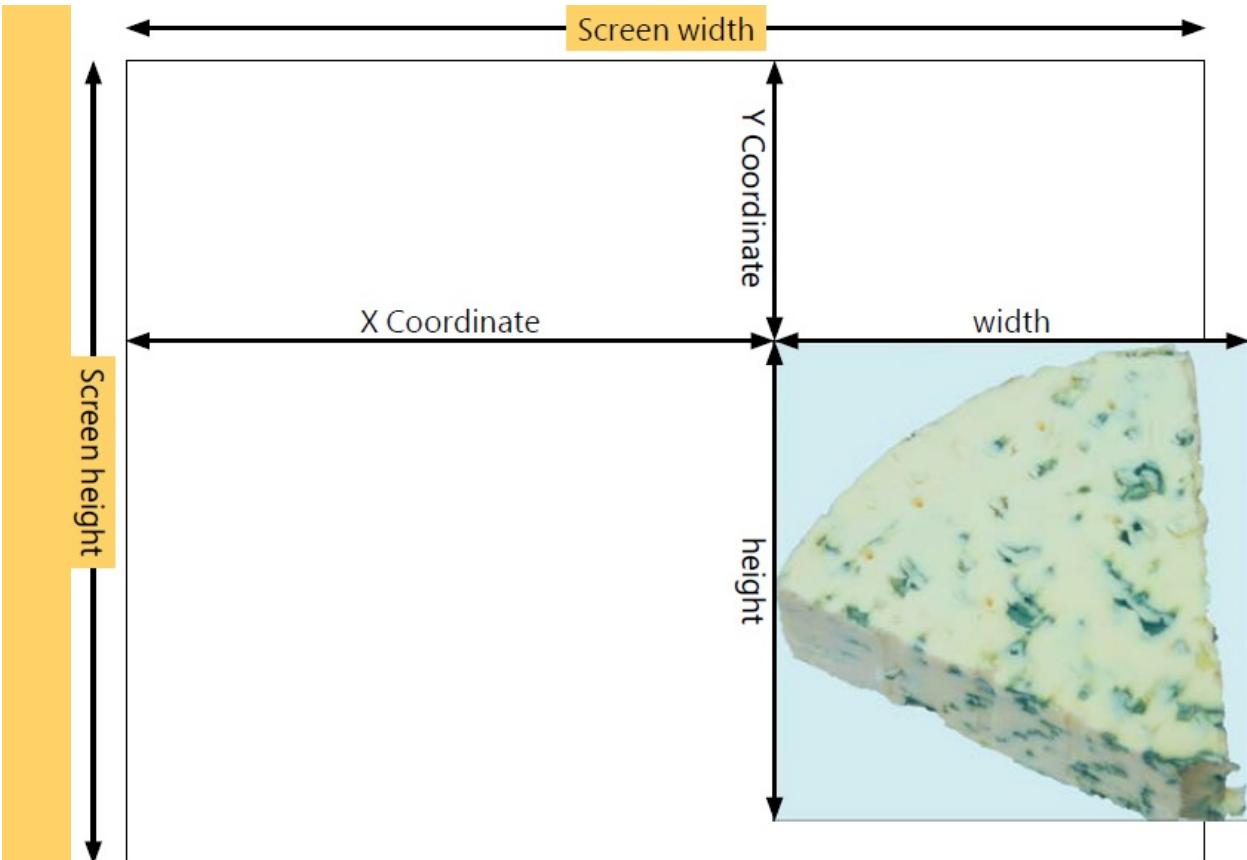
The code above defines the `Cheese` sprite. I've left off some of the movement methods to save space in the book, but you can find them all in the example program **EG16-06 Cheese Player** in the sample code for this chapter. You might have some questions about it.

**Question:** Why does the `Cheese` class not have an `__init__` or `draw` method?

**Answer:** The `Cheese` class is a subclass of the `Sprite` class we created earlier, which means the `Cheese` class inherits those two methods from the `Sprite` class.

**Question:** What do the `get_width` and `get_height` methods do?

**Answer:** These methods are provided by the pygame image class to allow a game to determine the dimensions of an image. We use them to make sure that the player cannot move the cheese off the screen.



The image above shows how this works. The program knows the position of the cheese and the width and height of the screen. If the x position plus the width of the cheese is greater than the width of the screen (as it is in the image above), the `update` method for the cheese will put the cheese back on the right edge:

[Click here to view code image](#)

```
if self.position[0] + self.image.get_width()
> self.game.width:
    self.position[0] = self.game.width -
self.image.get_width()
```

The position of a sprite is held in a list, with the element at location 0 holding the x position of the sprite. The sprite can use its reference to the game to get the width of the screen and the `get_width` method to obtain the width of the sprite image. Note

that in the above image, the cheese is not moving off the bottom of the screen. Forcing a sprite to stay on the screen in this way is called *clamping* the sprite.

The **Cheese** class also uses the width and the height of the sprite image to position the cheese in the center of the screen when the cheese is reset.

[Click here to view code image](#)

```
self.position[0] = (self.game.width -  
self.image.get_width())/2  
self.position[1] = (self.game.height -  
self.image.get_height())/2
```

## Control the player sprite

The **game** class creates an instance of the cheese sprite and uses keyboard events to trigger message to the sprite to control its movement. Below is the **game** class code that does this. If you run the example program **EG16-06 Cheese Player**, you can see this in action. The player can move the cheese around the screen, but the cheese will not move off the edge of the screen.

```

cheese_image = pygame.image.load('cheese.png')           Load the cheese image
self.cheese_sprite = Cheese(image=cheese_image, game=self) Create a cheese sprite

clock = pygame.time.Clock()                            Create a clock to control the game

while True:                                              Start of the game loop
    clock.tick(60)                                         Ensure that the game runs at 60 frames per second
    for e in pygame.event.get():                           Process game events
        if e.type == pygame.KEYDOWN:                      Is this a key pressed event?
            if e.key == pygame.K_ESCAPE:                  Has the Escape key been pressed?
                pygame.quit()                            Shut down the game
                return
            elif e.key == pygame.K_UP:                   Has the Up key been pressed?
                self.cheese_sprite.StartMoveUp()          Start the cheese moving up
            elif e.key == pygame.K_DOWN:                 Has the Down key been pressed?
                self.cheese_sprite.StartMoveDown()         Start the cheese moving down
            'Other cheese movement key handlers go here...'

    self.background_sprite.draw()                         Draw the background sprite
    self.background_sprite.update()                      Update the background sprite
    self.cheese_sprite.draw()                           Draw the cheese sprite
    self.cheese_sprite.update()                         Update the cheese sprite
    pygame.display.flip()                            Flip the display buffer to make the draw actions visible

```

## Add a Cracker sprite

Moving the cheese around the screen is fun for a while, but we need to add some targets for the player. The targets are crackers the player must use to capture the cheese. When a cracker is captured, the game score is increased, and the cracker moves to another random position on the screen. The **Cracker** sprite is a subclass of the **Sprite** class:

[Click here to view code image](#)

```
class Cracker(Sprite):
    '''


```

The cracker provides a target for the cheese

When reset, it moves to a new random place

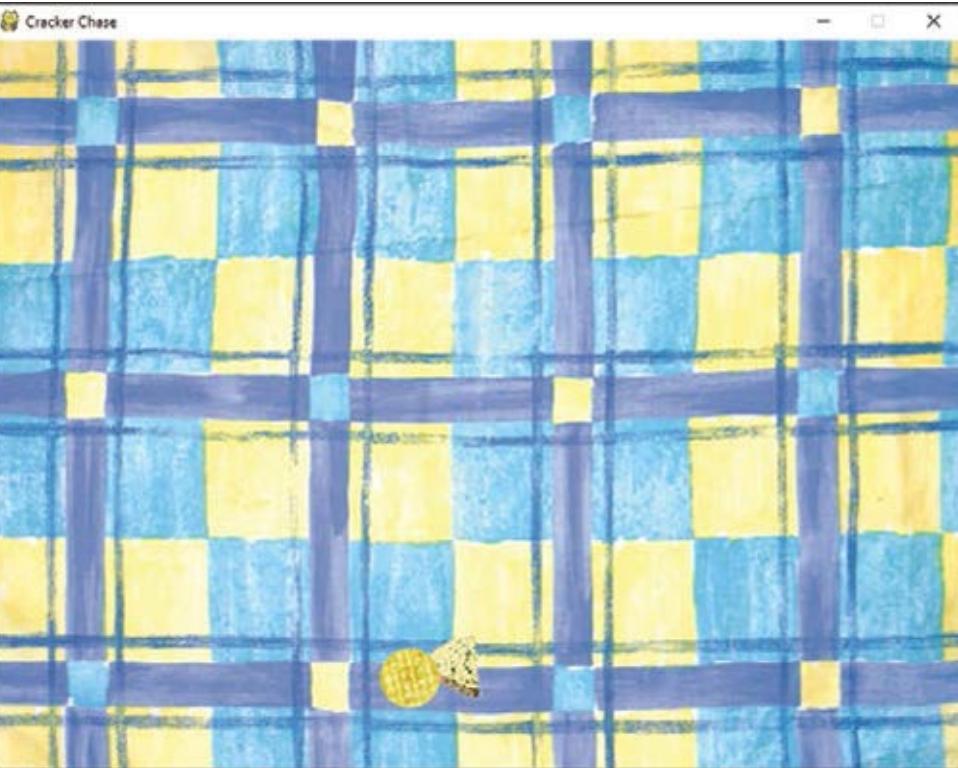
on the screen

```
'''
```

```
def reset(self):
    self.position[0] = random.randint(0,
                                      self.game.width-
                                      self.image.get_width())
    self.position[1] = random.randint(0,
                                      self.game.height-
                                      self.image.get_height())
```

The **Cracker** class is very small because it gets most of its behavior from its superclass, the **Sprite** class. It just contains one method, **reset**, which uses the Python random number generator to pick a random position for the cracker. We can add it to our game by creating it and then drawing it in the game loop. The sample program **EG16-07 Cheese and cracker** shows how this works.

**Figure 16-4** shows the game in action. The figure shows that there are at least two problems with this game. First, the cracker seems to be on top of the cheese. If the cheese is going to “capture” the cracker, it would look better if the cheese appeared to be “on top” of the cracker. We can fix this by changing the order in which the game elements are drawn. The pygame framework places images on the screen in the order they are drawn. The second problem with this game is that it looks a bit boring. I think we need more crackers to serve as additional targets.



**Figure 16-4** Cheese and cracker

## Add lots of sprite instances

We could increase the number of crackers by creating more individual cracker instances:

[Click here to view code image](#)

```
cracker_image = pygame.image.load('cracker.png')
self.cracker1 = Cracker(image=cracker_image,
game=self)
self.cracker2 = Cracker(image=cracker_image,
game=self)
self.cracker3 = Cracker(image=cracker_image,
game=self)
```

The code above would create three crackers called `cracker1`, `cracker2`, and

**cracker3**. This would work, but it would be hard to manage because the game would have to update and draw each of these sprites individually. It would turn into a real problem when game players request 50 crackers on the screen. Whenever we've had this problem in the past, we have used a collection of some kind (usually a list) to solve it. We can do this here, too.

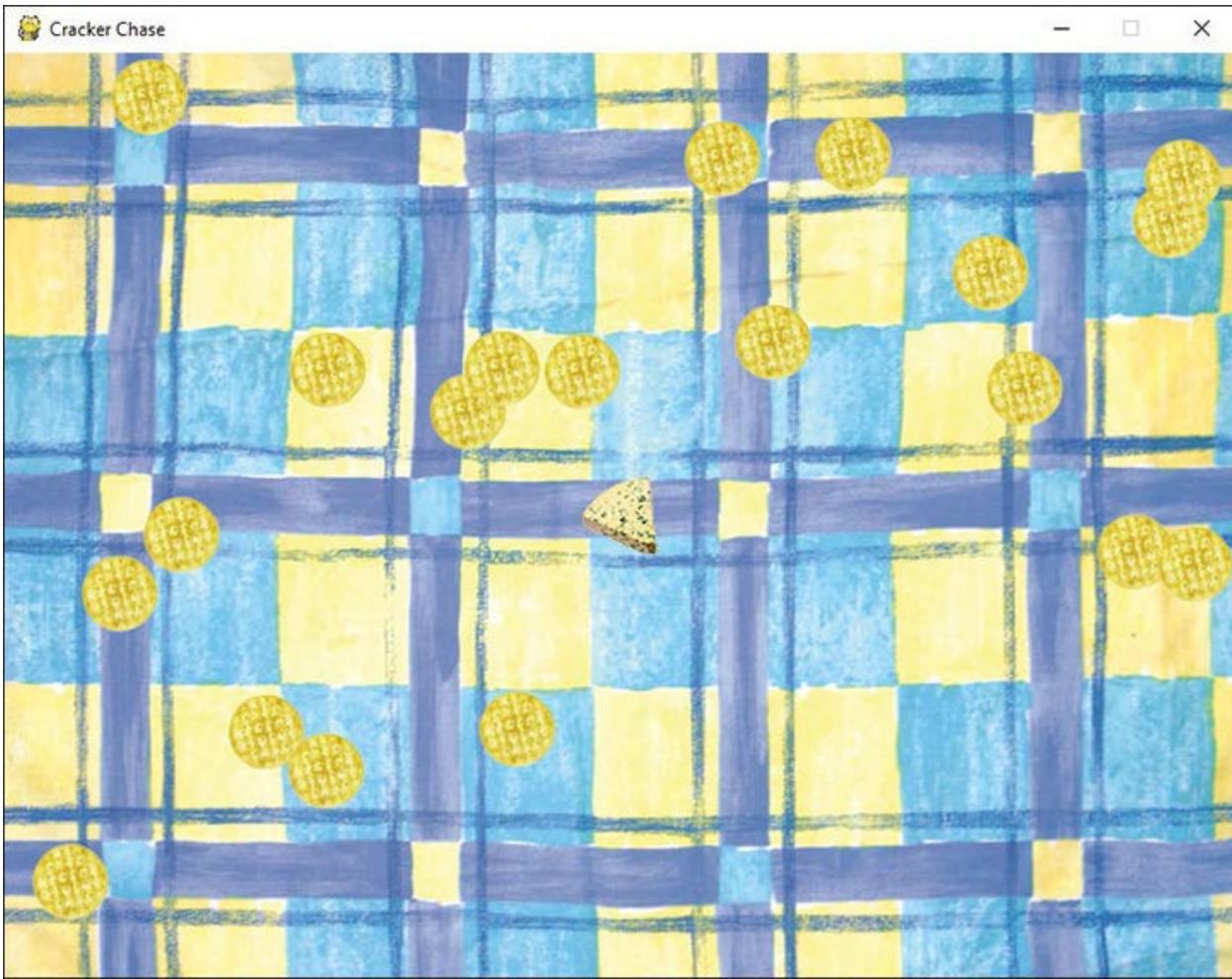
```
self.sprites = []          Create a list to hold all the sprites in the game  
  
cracker_image = pygame.image.load('cracker.png')      Load the cracker image  
  
for i in range(20):        Create a for loop that goes around 20 times  
    cracker_sprite = Cracker(image=cracker_image,game=self)  Create a Cracker sprite  
    self.sprites.append(cracker_sprite)                      Add the sprite to the list of sprites
```

The statements above create 20 cracker sprites. The game now contains a list, called **sprites**, which holds all the sprites in the game.

[Click here to view code image](#)

```
for sprite in self.sprites:  
    sprite.update()  
  
for sprite in self.sprites:  
    sprite.draw()
```

Above are the statements that we can use in the game loop to update and draw the cracker sprites. In the sample game **EG16-08 Cheese and crackers**, you can see how this works. This version of the game also adds the background and the cheese objects to the **sprites** list so that everything in the game is drawn and updated by the above two loops. **Figure 16-5** shows the game now. If we want to have even more crackers, we just need to change the limit of the range in the **for** loop that creates them.

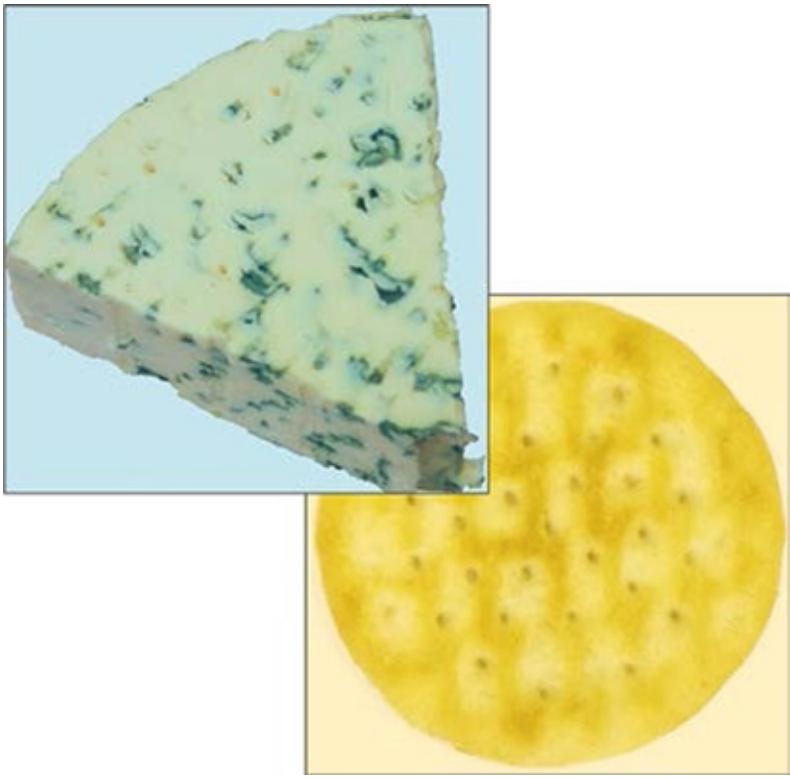


**Figure 16.5** Cheese and multiple crackers

## Catch the crackers

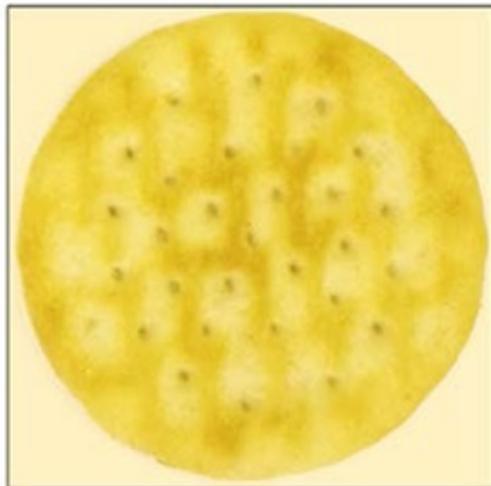
The game now has lots of crackers and a piece of cheese that can chase them. But nothing happens when the cheese “catches” a cracker. We need to add a behavior to the [Cracker](#) that detects when the cracker has been “caught” by the cheese. A cracker is caught by the cheese when the cheese moves “on top” of it. The game can detect when this happens by testing that rectangles enclosing the two sprites *intersect*.

[Figure 16-6](#) shows the cheese in the process of catching a cracker. The rectangles around the cheese and cracker images are called *bounding boxes*. When one bounding box moves “inside” another, we say that the two are intersecting. When the cracker updates, it will test to see whether it intersects with the cheese.



**Figure 16-6** Intersecting sprites

**Figure 16-7** shows how the test will work. In this figure, the two sprites are not intersecting because the right edge of the cheese is to the left of the left edge of the cracker. In other words, the cheese is too far to the left to intersect with the cracker. This would also be true if the cheese were above, below, or to the right of the cracker. We can create a method that tests for these four situations. If any of them are true, the rectangles do not intersect.



**Figure 16-7** Non-intersecting sprites

```

def intersects_with(self, target):
    """
    Returns True if this sprite intersects with
    the target supplied as a parameter
    """

    max_x = self.position[0]+self.image.get_width()      Get the right edge of this sprite
    max_y = self.position[1]+self.image.get_height()     Get the bottom edge of this sprite
    target_max_x = target.position[0]+target.image.get_width() Get the right edge of
                                                               the target
    target_max_y = target.position[1]+target.image.get_height() Get the bottom edge
                                                               of the target

    if max_x < target.position[0]:                      Is this sprite to the left?
        return False

    if max_y < target.position[1]:                      Is this sprite underneath?
        return False

    if self.position[0] > target_max_x:                 Is this sprite to the right?
        return False

    if self.position[1] > target_max_y:                 Is this sprite above?
        return False

    # if we get here, the sprites intersect
    return True                                         Return True because the sprites intersect

```

The method is an attribute of a `Sprite` object, which returns `True` if the sprite intersects with a particular target. We add this method to the `Sprite` class so that all sprites can use it. Now we can add an `update` method to the `Cracker` class that checks to see whether the cracker intersects with the cheese:

```

def update(self):
    if self.intersects_with(game.cheese_sprite):
        self.captured_sound.play()                    Have we been captured?
        self.reset()                                Play our capture sound effect
                                                    Reset the position of the cracker

```

## Add sound

The preceding `update` method plays a sound effect when a cracker is “captured” by the cheese. The pygame framework provides a `Sound` class to manage sound playback. When an instance of `Sound` is created, it is given the

name of the file that contains the sound data.

[Click here to view code image](#)

```
cracker_eat_sound = pygame.mixer.Sound('burp.wav')
```

The statement above creates a **Sound** instance called **cracker\_eat\_sound** from the sound file **burp.wav**. We pass this sound into a **Cracker** when we create a new instance:

```
cracker_sprite = Cracker(image=cracker_image, game=self,  
                           captured_sound=cracker_eat_sound) — Store the capture sound in the cracker
```

For this to work, we must modify the **\_\_init\_\_** method in the **Cracker** to store the sound in the cracker:

```
def __init__(self, image, game, captured_sound):  
    super().__init__(image, game) — Call the constructor in the superclass  
    self.captured_sound = captured_sound — Set the sound attribute of the cracker
```

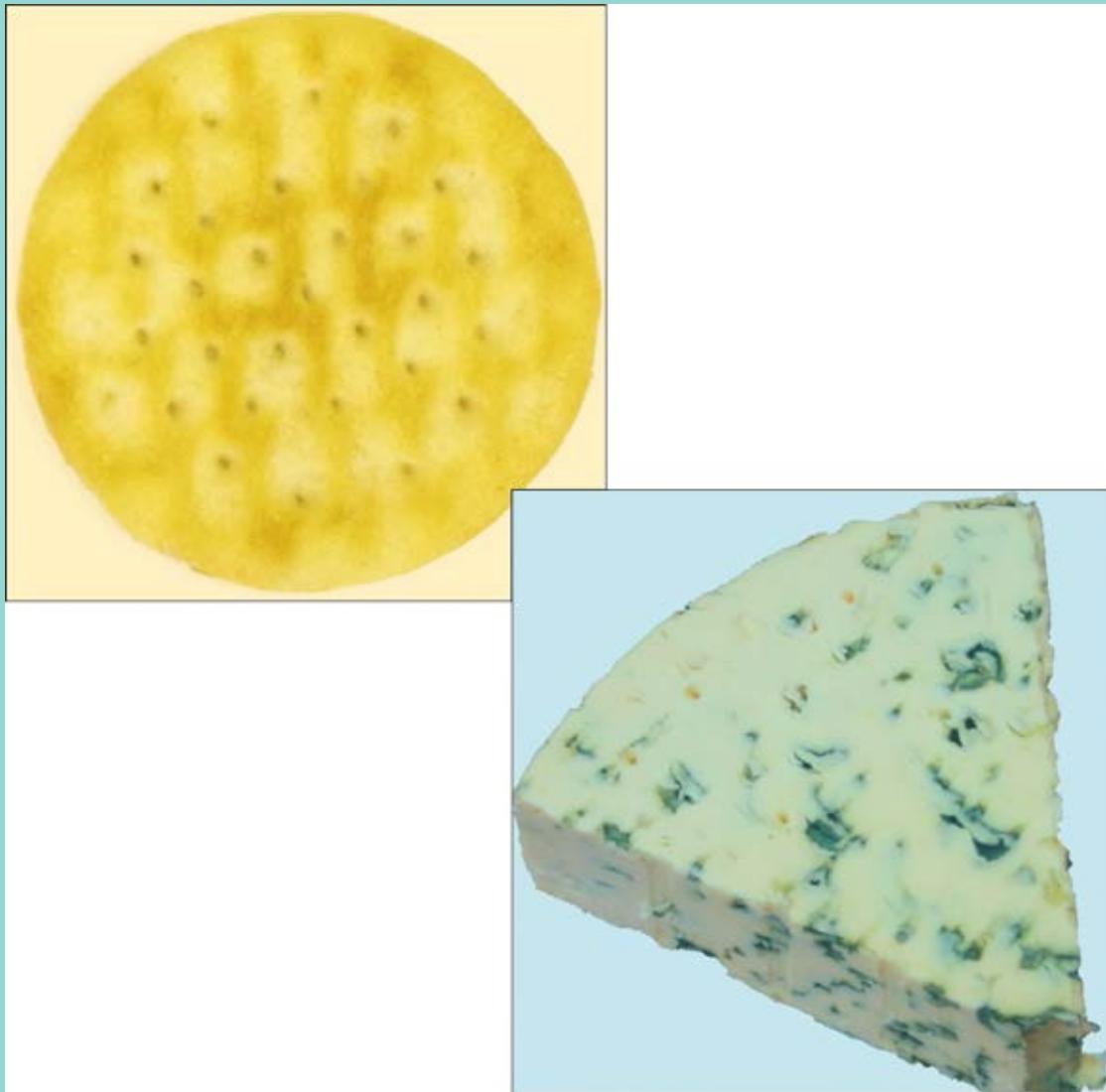
The attribute **captured\_sound** in the **Cracker** object can be used to play the sound effect when that cracker is eaten. In the present version of the game, all the crackers make the same sound when they are eaten, but we could use different sound effects for each cracker if we wished. The example program **EG16-09 Capturing crackers** lets the player capture crackers. When a cracker is captured, the game plays a sound effect and the cracker moves to a different location.

If you want to create your own sound effects, you can use the program Audacity to capture and edit sounds. It is a free download from [www.audacityteam.org](http://www.audacityteam.org) and is available for most operating systems.



## WHAT COULD GO WRONG

## Bad collision detection



There are some problems with using bounding boxes to detect collisions. The image above shows that the cheese and the cracker are not colliding, but the game will think that they are. This should not be too much of a problem for our game. It makes it easier for the player, as they don't always have to move the cheese right over the cracker to score a point. However, the player might have grounds for complaint if the game decides they have been caught by a killer tomato because of this issue. There are three ways to solve this problem:

- When the bounding boxes intersect (as they do above), we could check the intersecting rectangle (the part where the two bounding

boxes overlap) to see if they have any pixels in common. Doing so provides very precise collision detection, but it will slow down the game.

- Alternatively, we could detect collisions using distance rather than intersection, which works well if the sprites are mostly round.
- The final solution is the one I like best. I could make all the game images rectangular, so the sprites fill their bounding boxes and the player always sees when they have collided with something.



## PROGRAMMER'S POINT

### When you write a game, you control the universe

One of the reasons I like writing games so much is that I have complete control of what I'm making. If I'm solving a problem for a customer, I must deliver certain outcomes. But in a game, I can change what it does if I find a problem. I can also redefine the gameplay if I make a mistake in the program. Sometimes, this produces a more interesting behavior than the one I was trying to create. This has happened on a number of occasions.

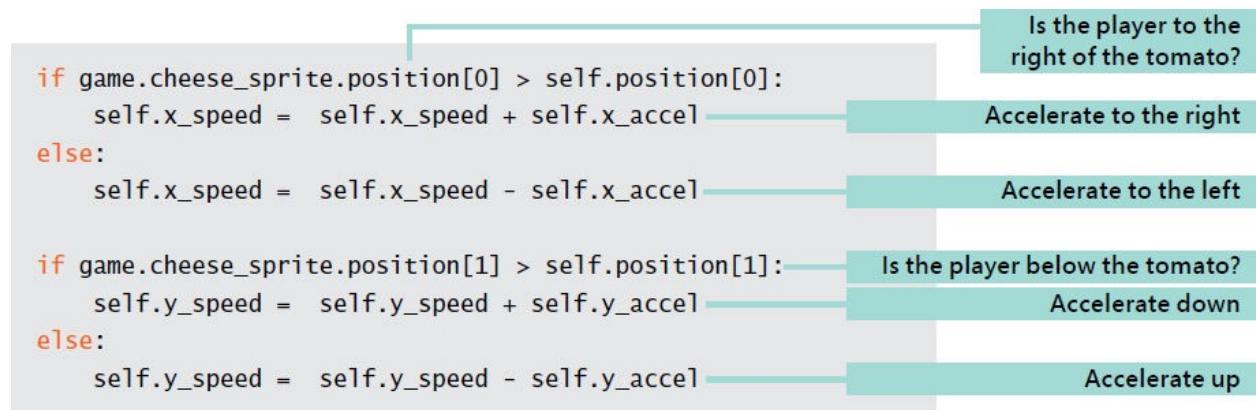
## Add a killer tomato

Currently, the game is not much of a game. There is no jeopardy for the player. When you make a game, you set up something that the player is trying to achieve. Then you add some elements that will make this difficult for them. In the case of the game "Cracker Chase," I want to add "killer tomatoes" that will relentlessly hunt down the player. As the game progresses, I want the player to be chased by increasingly more tomatoes until the game becomes all about survival. The tomatoes will be interesting because I'll give them *artificial intelligence* and *physics*.

## Add "artificial intelligence" to a sprite

Artificial intelligence sounds very difficult to achieve, but in the case of this game, it is actually very simple. At its heart, artificial intelligence in a game simply means making a program that would behave like a person in that

situation. If you were chasing me, you'd do this by moving toward me. The direction you would move would depend on my position relative to you. If I were to your left, you'd move left, and so on. We can put the same behavior into our killer tomato sprite:



This condition shows how we can make an intelligent killer tomato. It compares the x positions of the `cheese_sprite` and the tomato. If the cheese is to the right of the tomato, the x speed of the tomato is increased to make it move to the right. If the cheese is to the left of the tomato, it will accelerate in the other direction. The code above then repeats the process for the vertical positions of the two sprites. The result is a tomato that will move intelligently toward the cheese. Note that this means we could make a “cowardly” tomato that runs away from the player by making the acceleration negative so that the tomato accelerates in the opposite direction of the cheese.

## PROGRAMMER'S POINT

### Using “artificial intelligence” makes games much more interesting

There is a lot of debate as to whether “game artificial intelligence” is actually “proper” artificial intelligence. You can find a very good discussion of the issue here: <https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1>. I personally think that you can call this kind of programming “artificial intelligence” because players of a game really do react as if they are interacting with something intelligent when faced with something like our killer tomato. You can make a game much more compelling by giving game objects the kind of intelligence described above.

## Add physics to a sprite

Each time the game updates, it can update the position of the objects on the screen. The amount that each object moves each time the game updates is the *speed* of the object. When the player is moving, the cheese’s position is updated by the value 5. In other words, when the player is holding down a movement key, the position of the cheese in that direction is being changed by 5. The updates occur 60 times per second because this is the rate at which the game loop runs. In other words, the cheese would move 300 pixels ( $60 \times 5$ ) in a single second. We can increase the speed of the cheese by adding a larger value to the position each time it is updated. If we used a speed value of 10, we’d find that the cheese would move twice as fast.

*Acceleration* is the amount that the speed value is changing. The statements below update the `x_speed` of the tomato by the acceleration and then apply this speed to the position of the tomato.

```
self.x_speed = self.x_speed + self.x_accel                  Add the acceleration to the speed  
self.position[0] = self.position[0] + self.x_speed        Update the position of the sprite
```

The initial speed of the tomato is set to zero, so each time the tomato is updated, the speed (and hence the distance it moves) will increase. If we do this in conjunction with “artificial intelligence,” we get a tomato that will move rapidly toward the player.

If we just allowed the tomato to accelerate continuously, we’d find that the tomato would just get faster and faster, and the game would become unplayable.

The statement below adds some “friction” to slow down the tomato. The friction value is less than 1, so each time we multiply the speed by the friction, it will be reduced, which will cause the tomato to slow down over time.

```
self.x_speed = self.x_speed * self.friction_value      Multiply the speed by the friction
```

The friction and acceleration values are set in the `reset` method for the [Tomato](#) sprite:

[Click here to view code image](#)

```
def reset(self):
    self.entry_count = 0
    self.friction_value = 0.99
    self.x_accel = 0.2
    self.y_accel = 0.2
    self.x_speed = 0
    self.y_speed = 0
    self.position = [-100, -100]
```

After some experimentation, I came up with the acceleration value of 0.2 and a friction value of 0.99. If I want a sprite that chases me more quickly, I can increase the acceleration. If I want the sprite to slow down more quickly, I can increase the friction. You can have a lot of fun playing with these values. You can create sprites that drift slowly toward the player and, by making the acceleration negative, you can make them run away from the player.



## PROGRAMMER'S POINT

### When you write a game, you can always cheat

When you're writing a game, you should always start with the simplest, fastest way of getting an effect to work, and then improve it if necessary.

The “physics” that I’m using are not really an accurate simulation of physical objects. The way that I’ve implemented friction is not very realistic, but it works and gives the player a good experience. I find it interesting that six or seven lines of Python can make something that behaves in such a believable way. The Cracker Chase game uses very simple collision detection, artificial intelligence, and physics, but it is still fun to play. It really feels as if the tomatoes are chasing you. Making the physics model completely accurate would take a lot of extra work and would add very little to the gameplay.

## Create timed sprites

It's important that a game be progressive. If the game started with lots of killer tomatoes, the player would not last very long and would not enjoy the experience. I'd like each tomato to appear every 5 seconds. We can do this by giving each tomato an “entry delay” value when we construct it:

```
tomato_image = pygame.image.load('tomato.png')

for entry_delay in range(300,3000,300):
    tomato_sprite = Tomato(image=tomato_image,
                           game=self,
                           entry_delay=entry_delay)
    self.sprites.append(tomato_sprite)
```

Loop to generate the entry delay values  
Create a new tomato  
Give the tomato the entry delay value  
Add the tomato to the list of sprites

This code uses a version of the `range` function that we haven't seen before. The first argument to the `range` is the start value, which in this case is `300`. The second argument is the upper limit, and the third argument is the "step" between values. This will give us values of `entry_delay` that start at `300` and then go up in steps to `2700` (note that the value `3000` is the limit).

The `__init__` method in the `Tomato` class stores the value of `entry_delay` and is used to delay the entry of the sprite:

```
def update(self):

    self.entry_count = self.entry_count + 1
    if self.entry_count < self.entry_delay:
        return
```

Increase the entry counter by 1  
If the entry counter is less than the delay, return

The `update` method is called 60 times per second. The first tomato has an entry delay of `300`, which means that it will arrive at  $300/60$  seconds, which is 5 seconds after the game starts. The next tomato will appear 5 seconds after that, and so on, up until the last one. The example program **EG16-10 Killer tomato** shows how this works. It can get rather frantic after a few tomatoes have turned up and are chasing you.

## Complete the game

We now have a program that provides some gameplay. Now we need to turn this into a proper game. To do so, we need to add a start screen, provide a way that the player can start the game, detect and manage the end of the game, and then, because it adds a lot to the gameplay, add a high score.

### Add a start screen

A start screen is where the player will—you guessed it—start the game. Then, when the game is complete, the game returns to the start screen. We can add a start screen to the Cracker Chase game by using a flag value to indicate the mode of the game:

```
def start_game(self):
    for sprite in self.sprites:
        sprite.reset()
    self.score=0
    self.game_running = True
```

Reset all the sprites  
Clear the game score  
Set the flag to indicate that the game is running

Above is the method that starts a game playing. It resets all the sprites, sets the score to zero, and sets the `game_running` flag to `True`. The `game_running` flag controls the behavior of the game loop:

```
while True:
    clock.tick(60)
    if self.game_running:
        self.update_game()
        self.draw_game()
    else:
        self.update_start()
        self.draw_start()
    pygame.display.flip()
```

Repeat game forever  
Keep the frame rate to 60 frames per second  
Is the game active?  
Update the game  
Draw the game  
Update the start screen  
Draw the start screen  
Display the back buffer

This is the game loop for the game. The code that updates the game and draws it is now in methods that are called if the game is running. If the game is not running, methods are called to update and draw the start screen.

```
def update_start(self):
    for e in pygame.event.get():
        if e.type == pygame.KEYDOWN:
            if e.key == pygame.K_ESCAPE:
                pygame.quit()
                sys.exit()
            elif e.key == pygame.K_g:
                self.start_game()
```

Work through all the pygame events  
Is the event a key down?  
Is the key the Escape key?  
Quit pygame  
Exit the program

The start screen `update` behavior checks for two keys:

- If the G key is pressed, the `start_game` method is called to start the game.

If the Escape key is pressed, the method shuts down pygame by calling `quit` and then using the `exit` method from the `sys` module to end the program.

## Use `exit` to shut down Python

The `exit` method is in the `sys` module, which means that the game must import the module:

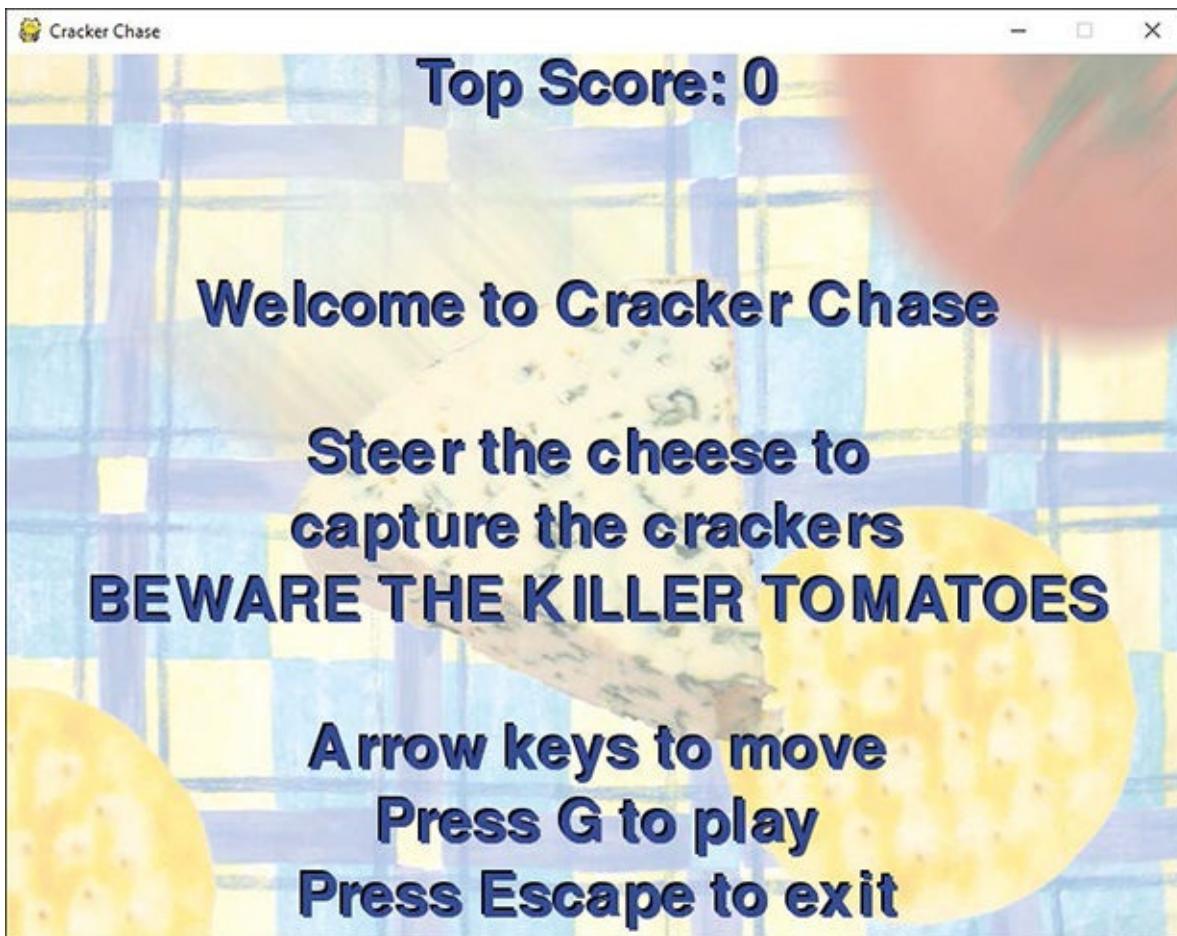
```
import sys
```

Once we have imported `sys`, we can call the `exit` function from the module to exit a Python program instantly.

```
sys.exit()
```

## Draw text in pygame

The start screen will display information for the player, as shown in **Figure 16-8**. The pygame framework can draw text on the screen. It uses a `Font` object that is created when the game starts.



**Figure 16-8** Start screen

[Click here to view code image](#)

```
self.font = pygame.font.Font(None, 60)
```

The initializer for the font accepts two parameters—the font design to use and the size of the font. The statement above specifies `None` for the font design, which will select the default pygame font. The size of 60 gives a text size that works well for the game. To place a message on the screen, the game first renders the text using the font.

[Click here to view code image](#)

```
text = self.font.render('hello world', True,  
(255, 0, 0))
```

---

The `render` method accepts three arguments:

- The first is a string that contains the text to be rendered.
- The second argument selects *aliasing*. This technique smooths the edges of the characters, and you should use it to make your text look nice.
- The third argument specifies the color of the text. It contains the amount of red, blue, and green that the text color should contain. The maximum color intensity is 255.

The code above will render “hello world” in bright red.

Once the text has been rendered, the next step is to blit it onto the display. We do this the same way we blit images.

[Click here to view code image](#)

```
self.surface.blit(text, (0,0))
```

The first argument to the `blit` method is for the text to be drawn; the second argument is the location on the screen. The statement above would render “hello world” in the top left corner of the screen. A program can get the width and the height of rendered text, which can be used to center text on the screen. The `CrackerChase` class contains a little method that draws text on the screen:

```

def display_message(self, message, y_pos):
    """
    Displays a message on the screen
    The first argument is the message text
    The second argument is the vertical position
    of the text
    The text is drawn centered on the screen
    It is drawn with a black shadow
    ...
    shadow = self.font.render(message, True, (0,0,0))      Render the text in black
    text = self.font.render(message, True, (0,0,255))      Render the text in blue
    text_position = [self.width/2 - text.get_width()/2, y_pos]  Calculate the position of the text
    self.surface.blit(shadow, text_position)                Draw the shadow
    text_position[0] += 2                                    Move the draw position across
    text_position[1] += 2                                    Move the draw position down
    self.surface.blit(text, text_position)                  Draw the text

```

This method actually draws the text twice. The first time, the text is drawn in black, and then the text is drawn again in blue. The second time the text is drawn, it is moved slightly to make it appear that the black text is a shadow.

This method uses the `+=` operator, which can be used to increase the value of a variable. Rather than writing:

[Click here to view code image](#)

```
text_position[0] = text_position[0]+2
```

You can write:

```
text_position[0] += 2
```

There are similar operators for subtract (`-=`), multiply (`*=`) and divide (`/=`).

If you look closely at [Figure 16-8](#), you can see that the result of this extra drawing is that text looks three-dimensional, which makes text stand out on the screen.



## PROGRAMMER'S POINT

### Don't worry about making the graphics hardware work for you

You might think it's rather extravagant to draw all the text on the screen twice just to get a shadow effect. However, modern graphics hardware is perfectly capable of many thousands of drawing operations per second. I've been known to draw text twenty times just to get a nice blurred shadow effect behind it. If you think something might look good, my advice is to try it and only worry about performance if the game seems to run very slowly after you've done it.

[Click here to view code image](#)

```
def draw_start(self):
    self.start_background_sprite.draw()
    self.display_message(message='Top Score: ' +
str(self.top_score), y_pos=0)
    self.display_message(message='Welcome to
Cracker Chase', y_pos=150)
    self.display_message(message='Steer the cheese
to', y_pos=250)
    self.display_message(message='capture the
crackers', y_pos=300)
    self.display_message(message='BEWARE THE KILLER
TOMATOES', y_pos=350)
    self.display_message(message='Arrow keys to
move', y_pos=450)
    self.display_message(message='Press G to play',
y_pos=500)
    self.display_message(message='Press Escape to
exit', y_pos=550)
```

Above is the `draw_start` method for the game, which draws the sprite that contains the background image and then displays the help messages on the display.



## PROGRAMMER'S POINT

### Make sure you tell people how to play your game

In my long and distinguished career in computing, I've judged quite a few game development competitions. I've lost count of the number of games that I've tried to play and failed because the game doesn't tell me what to do. The problem is usually that everyone focuses on making the game, and not on telling people how to play it. Failing at a game while you work out which keys you are supposed to press doesn't make for a very good introduction to it, so make sure that you make the instructions clear and present them right at the start.

### End the game

The start screen allows the player to play the game. We've seen that the game has two states, which are managed by the `game_running` attribute. This attribute is set to `True` when the game is running and `False` when the start screen is displayed. Now we need to create the code that manages the `game_running` value. At the start of this section, we saw that the game contained a method that started the game. The game also contains a method to end it.

[Click here to view code image](#)

```
def end_game(self):
    self.game_running = False
    if self.score > self.top_score:
        self.top_score = self.score
```

The `end_game` method sets `game_running` to `False`. It also updates the `top_score` value. If the current score is greater than the highest score so far, it is updated to the new top score.



## PROGRAMMER'S POINT

### Adding a high score makes a game much more interesting

Adding a high score to a game makes the game much more compelling. Players will spend a lot of time trying to beat their previous scores. A good improvement to this game would be to make it save the high score in a file and load the high score when the game starts.

## Detect the game end

The game ends when the player collides with a killer tomato, which is detected in the `update` method for the tomato sprite:

[Click here to view code image](#)

```
def update(self):
    ' position update code for the tomato here'
    if self.intersects_with(game.cheese_sprite):
        self.game.end_game()
```

We can add more logic to make the game more interesting. We could give the player a health value that reduces each time he or she collides with a tomato. We could make the health slowly recover over time. We could even add the traditional “three lives” that are standard for games like this.



### PROGRAMMER'S POINT

## Always make a playable game

Something else I noticed while judging game development competitions was that some teams would produce a brilliant piece of gameplay but not attach it to a game. You'd start playing the game and find that it never actually ended. You should make sure that your game is a complete game from the very start. The game should have a beginning, middle, and end. As you have seen in this section, it's easy to do this, but when people start making a game, they seem to leave it to the last minute to create the game start screen and the game ending code, so that what they produce is not a game, but more of a technical demo, which is not quite the same thing. Making your game into a proper game right from the start also makes it much easier for people to try it and then give you feedback.

## Score the game

Each time the cheese collides with a cracker, the game score is increased. The

score is updated in the `update` method for the cracker sprite:

```
def update(self):
    if self.intersects_with(game.cheese_sprite):
        self.captured_sound.play()
        self.reset()
        self.game.score += 10
```

Update the game score

The score is displayed on the screen each time the game display is drawn by the `draw_game` method.

```
def draw_game(self):
    for sprite in self.sprites:
        sprite.draw()
    status = 'Score: ' + str(game.score)
    self.display_message(status, 0)
```

Draw all the game sprites

Assemble the score message

Display the score at the top of the screen

You can find the completed game in the folder **EG16-11 Complete Game**. It's fun to play for short bursts, particularly if there are a few of you trying to beat the high score. My highest score so far is 380, but I never was any good at playing video games.



## MAKE SOMETHING HAPPEN: DEVELOPMENT CHALLENGE

### Make a game of your own

The Cracker Chase game can be used as the basis of any sprite-based game you might like to create. You can change the artwork, create new types of enemies, make the game two-player, or add extra sound effects. When I said at the start of this book that programming is the most creative thing you can learn to do, this is the kind of thing I was talking about. You can create a game called "Closet frenzy" where you are chased around by coat hangers while you search for a matching sock. You could create "Walrus Space Rescue," where you must steer an

interplanetary walrus through an asteroid minefield. Anything you can think up, you can build. However, one word of caution. Don't have too many ideas. I've seen lots of game development teams get upset because they can't get all their ideas to work at once. It is much more sensible to get something simple working and then add things to it later.

## What you have learned

In this chapter, you created a playable game and discovered how the pygame framework lets you work with graphics and sound. You found that a class hierarchy, with a sprite superclass and different game objects as subclasses of this is a great way to create game objects. You also discovered that games work by having a “game loop” that repeatedly updates and draws items on the screen. You used the event mechanism of pygame to capture keyboard input, and you used events to control an object on the screen. You've seen that “artificial intelligence” can be created with a couple of if conditions, and physics can be implemented using a few calculations. You also implemented a start screen and a game screen to make a complete game experience.

Hopefully, you've also taken a few ideas of your own and used them to create some more games.

Here are some points to ponder about game development.

### **Do all games work using a game loop?**

Most games use a game loop. A text-based adventure will work by reading in what you type and replying, but most modern games work with a loop.

### **Why are draw and update separate methods?**

You might wonder why I separated the draw and update behaviors in the game. Although they are separate methods, they always seem to be called together. Why not have just one method (perhaps called `do_game`) which does both?

The answer has to do with performance. For simple games like Cracker Chase, it's perfectly fine for the drawing and updating to take place at the same rate.

However, if you’re running on a low-performance platform, you may want to update the game at a different rate from the rate you draw it. The reason for this is that people are much more tolerant of the game display “flickering” than they are for changes in the speed of a game update. If the game update slows down, it can cause problems with collisions not being detected (for example, bullets might pass right through things without the game noticing they had collided). For this reason, a game should separate drawing and updating so that the two processes can be made to run at different speeds if required.

### **How would I create an attract mode for my game?**

Currently, our game just has two states, the start screen and the game screen. Many games have an “attract mode” screen as well, which displays some gameplay. Creating an attract mode screen is quite easy. We could make an “AI player” who moved the cheese around the screen in a random way, and then just run the game with the random player at the controls. We could add an “attract mode” behavior to the tomatoes so that they were aiming for a point some distance from the player, to make the game last longer in demo mode.

### **How could I make the gameplay the same each time the game is played?**

The game uses the Python random number generator to produce the position of the crackers, which means each time the game runs, the crackers are in a different position. We can use the `seed` function from the Random module to give the Python random number generator the same seed before each game. This would mean that the crackers would be drawn and would respawn in the same sequence each time the game was played. A determined player could learn the pattern and use this to get a high score.

### **Is the author of the game always the best person at playing it?**

Most definitely not. I’m often surprised how other people can be much better than me at playing games I’ve created. Sometimes they even try to help me with hints and tips about things to do in the game to get a higher score.

# Index



Index entries listed in gray are only found in the PDF files for **Part 3** available at <https://aka.ms/BeginCodePython/downloads>.

## Numbers

0s and 1s, [31](#)

## Symbols

\\" escape sequence, [82](#)  
\' escape sequence, [82](#)  
\\" escape sequence, [82](#)  
\' escape sequence, [82](#)  
\* character, inclusion in arguments, [460](#)  
+ (addition), [29](#)  
== (equality) operator, [112, 125–126](#)  
> (greater than) operator, [112](#)  
>= (greater than or equals) operator, [112](#)  
< (less than) operator, [112](#)  
<= (less than or equals) operator, [112](#)  
\* (multiplication), [29](#)  
!= (not equals) operator, [112](#)  
( ( )) (parentheses), [30, 33](#)  
' (single quote), entering, [81](#)

# A

\a escape sequence, 82  
abstraction, 381, 437  
acceleration, 626  
accuracy versus precision, 92  
Add Session menu, modifying, 337–340  
`add_session` method, 316, 321–322  
adding machine, 506  
addition (+), 29  
address book. *See* [Contacts app](#)  
age, holding for contacts, 318  
alarm clock, making, 128. *See also* clock; digital clock  
algorithms, 227  
`and` operator, 115, 117  
`append` method, 215–224, 259  
`append` mode, 242  
application behaviors, implementing, 405–409  
applications. *See also* [data-processing applications](#)  
    data design, 376–377  
    versus programs, 14  
arguments  
    \* character, 460  
    arbitrary number, 457–460  
    defaults in initializers, 299  
    naming, 182  
    and parameters, 176–179  
    positional and keyword, 180–181  
arithmetic operator, 111  
artificial intelligence, adding to sprites, 625–626  
artificial intelligence (AI), 14  
ASCII symbols, escape sequences, 82–83  
assert statement, 471–472  
assignment statements, using with variables, 74  
“attract mode” screen, using with games, 637

attributes. *See also* [data attributes](#); [method attributes](#); [static class attributes](#); [version attributes](#) adding, [306](#)  
[\\_\\_billing\\_amount](#), [343](#)  
confusion, [275](#)  
contacts app, [274](#)  
missing, [289](#)  
and values, [109](#)

## B

backslash (\) character, escape sequence, [82](#), [84](#)  
[BaseHTTPRequestHandler](#) class, [577](#), [590](#)  
bell, escape sequence, [82](#)  
billing amount, managing, [337–339](#)  
[\\_\\_billing\\_amount](#) attribute, [343](#)  
[bin](#) function, [39–41](#). *See also* [numbers](#)  
binary files, storing, [307](#)  
binary representation, [32](#)  
bits, [31](#)  
blitting, [602](#)  
block copy action, warning, [379](#)  
[bool](#) function, [107–108](#)  
Boolean expressions, [109–110](#). *See also* [expressions](#)  
Boolean operations, [114–118](#)  
Boolean values, [106–108](#), [138](#)  
Boolean variables. *See also* [variables](#)  
    creating, [106](#)  
    [doneSwap](#), [234](#)  
bounding boxes, [620](#)  
[break](#) statement, [157–159](#), [163](#), [168](#)  
breaking programs, [15](#)  
breakpoint, adding, [202–203](#)  
broadcast address, [551](#)  
[BTCInput](#) module, function references, [441–444](#)  
[BTCInput.py](#) source file, [201–202](#), [267](#)

bubble sort algorithm. *See also* [sorting using bubble sort](#)

[Button](#) instance, 509

bytes, 39

## C

calculations, performing, 96–98

canvas, drawing on, 518–522, 525–526

carriage return, escape sequence, 82

cars, microcomputers, 23

catching exceptions, 326, 512

centigrade and Fahrenheit, converting between, 100–102, 515–517

[check\\_version](#) method, 342, 345

Cheese class, 614–615

cheese image, 601–602

cheese object, 609

[chr](#) function, 38

class hierarchies

explained, 381

protecting data in, 395–396

versus sets, 431–433

using, 435

class instances, setting up, 294–299

class properties, 332–336, 369–370

class references, 467–468. *See also* [references and lists](#)

class variables, validation and methods, 317–318

class versions, managing, 340–345

classes. *See also* [superclasses and subclasses](#)

advantages, 433

[BaseHTTPRequestHandler](#), 577

[BaseHTTPRequestHandler](#), 590

[Cracker](#), 618

creating, 273–274

data attributes, 311–312

data storage, 384–386

designing with, 421–422  
**Dress**, 378, 380, 383, 388  
**ElementTree**, 565–567  
Fashion Shop application, 434  
game, 613, 617  
**HTTPServer**, 577, 590  
instances, 284  
method attributes, 314–316  
method overriding, 388–392  
methods in, 368  
**Note**, 365–366  
**Pants**, 378, 380, 383  
**Secret**, 329–331  
**Sound** in pygame, 623  
**Sprite**, 612, 618  
static items, 437  
**StockItem**, 380–381, 383–384, 387  
storing contact details, 272–273  
upgrading, 343  
using as values, 466–470  
**webPageHandler**, 588  
classes hierarchy, storing data in, 384–386  
clock, making, 110–111. *See also* alarm clock; digital alarm clock  
**close** method, 240  
closing quote, missing, 33  
code, documenting, 197  
code security, 331  
cohesion, 312–313  
cohesive object, 368, 370, 435–436. *See also* components  
collision detection, pygame, pygame, 624  
command shell. *See* IDLE Command Shell  
commands  
    **import**, 58  
    **randint**, 58–60

comments  
length, 71  
using, 61–62  
using with functions, 195–197

communication, 21

comparison operators, 111–113

components. *See also* cohesive object  
versus objects, 435  
self-contained, 410  
user interface, 417–421  
using, 435

computers  
data-processing applications, 23  
in devices, 23  
and programs, 22

conditional statement layout, 123–124

conditions, 119

connections and datagrams, 561

**Contact** class. *See also* Time Tracker  
creating, 298  
`hours_worked` attribute, 315  
initializer, 297  
properties, 333  
using, 273–276

**Contact** instance  
attributes, 305  
creating, 294, 299

**Contact** object, time tracker, 311

contacts. *See also* test contacts  
Edit Contact item, 278  
editing, 287–289  
holding ages, 318  
lists and references, 282–284  
loading from files, 292  
objects and references, 281–282

- saving and loading, 293–294
- saving in files, 289–291
- storing in dictionaries, 303–304

Contacts app

- class instances, 294–299
- classes for details, 272–275
- duplicate names, 277–278
- prototype, 267–268
- refactoring, 279–280
- save and load, 293–294
- starting, 266–267
- storing details, 269–270

`contacts.pickle` file, opening, 290

`continue` keyword, using with loops, 158–159, 163, 168

`count` variable

- using, 231
- using with loops, 218
- using with while loop, 160

countdown program, looping, 147

`Cracker` class, 618

- cracker sprite, 618
- crackers, catching, 620–622
- crackers sprite, 609

Ctrl key. *See* `keyboard shortcuts`

customers

- communicating with, 21
- writing software for, 433

## D

data

- and information, 31–35, 41–42
- loading using pickle, 292
- storing in classes hierarchy, 384–386
- storing in files, 238–239

data attributes. *See also* attributes  
    adding to classes, 311–312  
    protecting, 328–331, 368–369  
    using, 274

data design, Fashion Shop application, 396–401

data processors  
    programs as, 24–25  
    Python as, 25–30, 32, 51

data storage  
    **bin** function, 39  
    version management, 345

data storage app, creating, 304

data types, text and numbers, 35

datagrams  
    and connections, 561  
    defined, 553  
    fetching, 554  
    using, 568

data-processing applications, 23, 547. *See also* applications

days, counting through, 255–256

debugger, 202–208

decimal library, 92

decisions, using to make applications, 129–133, 138–139

decorators, 319, 321

**def**, using with functions, 175

“defensive programming,” 148. *See also* programs

dependency injection, 470

design decisions, 434

designing with classes, 421–422

desktop, running Python, 63–64

devices, computers in, 23

dictionaries  
    **access\_control**, 302  
    creating, 300–302  
    deleting entries, 302

“key:item,” 302  
keys for items, 300–301, 307  
managing, 302–303  
returning from functions, 303  
storing contacts, 303–304  
digital alarm clock, making, 167. *See also* `clock`  
dimensions, using with lists, 255  
display elements, grouping in frames, 528–529  
`display_contact` method, 346–347  
`display_image` function, 167  
Django framework, 590–591  
DNS (domain name system), 561  
`do_add` function, 512–513  
documentation  
    viewing, 478–481  
    writing, 485  
documenting code, 197  
`doneSwap` Boolean variable, 234  
dots, drawing in pygame, 600  
double precision value, 92  
double quote (“) character, escape sequence, 82  
downloading Python, 7  
`draw_text` function, 167  
drawing on canvas, 518–522  
drawing program, creating, 523–524  
`Dress` class, 378, 380, 383, 388  
duplicate names, 277–278

## E

Easter Egg, 13  
`edit_contact` function, 279, 289  
editing contacts, 278, 287–289  
editors  
    `get_from_editor` method, 534–535

`load_into_editor` method, 533–534  
using, 499

egg timer, 61–62

`ElementTree` class, 565–567. *See also XML (eXtensible Markup Language)*

`elif` keyword, 226–227

else part, adding to if construction, 125, 131

encrypted websites, 591

end of program, delaying, 64

`end_game` method, 634

EOL “End of Line,” 33

equality (`==`) operator, 112–114, 125–126

error checking, 55–56

error handling, GUI (Graphical User Interface), 512–513

error messages

- appearance, 77–78
- expressions, 28

errors. *See also invalid user entry*

- assigning faults, 43
- floating-point variables, 91

escape sequences, 82–83

`eval` function, 86

event handler function, GUI, 510

events

- creating drawing programs, 523–524
- and drawing, 518–522
- pygame, 606–607
- Tkinter, 522–523

except handlers, 156–157

exception error message, extracting, 325–326

exceptions

- catching, 199, 326, 512
- construction, 154
- file handling, 248–249
- handling, 156

loops, 155  
and number reading, 154  
raising and dealing with, 327–328  
raising to indicate errors, 323–324  
sockets, 556  
testing for, 475–476  
`ValueError`, 153  
`exit` method, 630  
expressions. *See also* Boolean expressions; lambda expressions  
    addition and multiplication, 29  
    anatomy, 27  
    comparison operator, 111–112  
    error messages, 28  
    in programs, 48  
    working out, 28  
Extension, installing for Visual Studio Code, 491–492

## F

Fahrenheit and centigrade, converting between, 100–102, 515–517  
failure, planning for, 157  
failure behaviors, testing, 151  
`False` and `True` values, 106–109, 115–119  
Fashion Shop application  
    application behaviors, 405–409  
    class diagram, 380  
    classes, 434  
    data design, 376–377, 396–401  
    design decisions, 434  
    GUI (Graphical User Interface), 544–545  
    instrumented stock items, 402–405  
    item name, 387–388  
    object-oriented design, 376–379  
    objects as components, 409–410  
    overview, 374–376

`__str__` method in classes, 388–391  
superclasses and subclasses, 379–381  
version management, 392–393

**FashionShop** component

- class, 411–412
- features, 410

**FashionShop** object. *See also* `sets`

- classes, 421–422
- stock data, 416–417
- stock items, 414–416
- user interface component, 417–421

faults

- fixing, 249
- testing for, 477

FDS (functional design specification), 20

file access, tidying up, 249–250

file associations, changing, 63–64

file errors, 257–258

file extensions, 63

file handling exceptions, 248–249

file server, connecting to, 582–583

files

- `close` method, 240
- `open` function, 239
- overwriting, 240
- reading from, 244–246
- saving, 49–50
- storing data, 238–239
- `write` method, 240–242
- writing into, 239–241, 260

filter on tags, 429–431

`finally`, using with exceptions, 248–249

`find_contact` function, 271–272, 277, 279, 289

firewall problems, 560

`firstProg`, naming, 50  
Flask framework, 590–591  
`float` and `int`, converting between, 98–99  
`float` function, 95–96  
floating-point numbers, 90–92  
floating-point values  
    converting into integers, 99  
    converting strings to, 95–96  
    and equality, 113–114  
floating-point variables, 92–94, 103  
folders, programs in, 63  
`Font` object, using with pygame, 630–633  
`for` loop construction, 162–163  
    `break` statement, 163–166  
    `continue` statement, 163–166  
    displaying lists, 219–221  
    teletype printer, 184–185  
    tuples, 257  
    using with lists, 253–254  
    versus `while` loop, 230–231  
`format()` method, using with strings, 348  
fortune teller, 137  
fraction library, 92  
frames, using with display elements, 528–529  
`func` function, 191  
function calls, return values, 185–186  
function references, 440–446  
functions. *See also* iterator functions; methods; reusable functions; snaps  
    framework  
    arguments, 457–460  
    `bin`, 39–41  
    `bool`, 107–108  
    calling functions, 174–175  
    `chr`, 38

converting into modules, 201–202  
defining, 175  
designing with, 188–189  
`do_add`, 512–513  
`edit_contact`, 279, 289  
elements, 172  
`eval`, 86  
`find_contact`, 271–272, 277, 279, 289  
float, 95–96  
`func`, 191  
`get_treasure_location`, 258–259  
`get_value`, 186–188  
`get_weather_temp`, 101  
`greeter`, 172–173  
help information, 195–197  
input, 84–87, 107  
`int`, 87–88, 96, 247  
interactive help, 182  
investigating, 172–173  
`isinstance`, 178–179  
`join`, 369  
lambda expressions, 446–450  
`load_contacts`, 292  
`load_sales`, 246–247  
local variables, 189–190  
`localtime`, 109–111  
`make_test_data`, 253  
`map`, 369, 440  
and methods, 126–127  
`new_contact`, 269–270, 276  
number input, 197–198  
`open`, 239, 260  
`ord`, 36–37  
parameters, 176, 179–180

placeholders, 224–225  
`play_sound`, 363  
`print`, 51–57, 184  
`print_sales`, 223–224  
`range`, 163, 451  
`raw_input`, 86  
`read_float`, 197–198  
`read_float_ranged`, 198–200  
`read_sales`, 223–224  
`read_text`, 194–195, 198, 268  
`readme`, 461  
`real_number`, 443  
references, 440–446  
return values, 186–187  
returning dictionaries, 302  
sales analysis program, 223–224  
`save`, 251  
`save_contacts`, 291  
`save_sales`, 242–243  
`seed`, 637  
`sleep`, 60–61, 95, 167, 184–185  
`sort_high_to_low`, 228  
`startswith`, 272  
`str`, 243  
strings in, 209  
“stub” versions, 239  
`sum`, 460  
`super`, 387  
`type`, 285  
using, 209  
`what_would_I_do`, 183

# G

`game` class, 613, 617  
game consoles, 23  
game loops, 636  
games, benefits, 625. *See also* `pygame`  
`GET` request, 585, 590  
`get_from_editor`, method, 534–535  
`get_hours_worked` method, 315  
`get_string` function, 134–135  
`get_treasure_location` function, 258–259  
`get_value` function, 186–188  
`get_weather_temp` function, 101  
global variables, 190–193, 208. *See also* `variables`  
graphical application, creating, 506–507  
greater than (`>`) operator, 112  
greater than or equals (`>=`) operator, 112  
`greeter` function, 172–173  
“greeter” program, making, 85  
grid, laying out, 507–510  
grid cells, spanning, 509–510  
grid layout for GUI, 507–510  
GUI (Graphical User Interface). *See also* `Tkinter`; `user interface`  
    application, 544–545  
    versus Command Shell, 547  
    creating, 499–506  
    display elements in frames, 528–529  
    drawing on canvas, 525–526  
    drawing program, 523–524  
    editable `StockItem`, 529–536  
    error handling, 512–514  
    event handler, 510–511  
    Fahrenheit and centigrade, 515–517  
    grid layout, 507–510  
    Kivy, 547  
    `Listbox` selector, 537–543

`mainloop`, 511–512  
message box, 514–515  
multi-line text, 526  
padding, 509  
PyQT, 547  
spanning grid cells, 509–510  
sticky formatting, 508  
Text object, 527–528

## H

### `hello`

attempt, 12  
saying, 32–33  
`Hello`, printing, 173  
“`hello world`,” displaying in snaps, 66  
help information, adding to functions, 195–197  
help message, 59. *See also* interactive help  
“High-Low” party game, 69  
hosts and ports, 552  
HTML (Hypertext Markup Language), 562, 568  
HTTP (Hyper Text Transfer Protocol), 568  
`HTTP POST` request, 585–589  
`HTTPServer` class, 577

## I

`i` variable name, 189–190  
Ice-Cream Sales program, 213  
`IDLE` command, 26  
IDLE Command Shell. *See also* Python Command Shell; Visual Studio Code  
arbitrary arguments, 457–460  
classes, 273–274  
classes as values, 466–467  
dictionaries, 300–302  
eliminating save requests, 54

events and drawing, 518–522  
exceptions, 324–325  
file-server connection, 582–583  
functions, 172–173  
“greeter” program, 85  
versus GUI (Graphical User Interface), 547  
“immutable,” 284–287  
initializer, 295–297  
`join` function, 362–363  
lambda expression, 447–448  
`Listbox` object, 537–539  
lists, 215–217  
`map` function and iteration, 356–361  
message board, 585  
method overriding, 388–390  
network messages, 553–555  
one-handed clock, 110–111  
`ord` function, 37  
`print` function, 52–53  
properties, 334–335  
protecting data attributes in classes, 329–331  
`pygame`, 594–598  
`random` library, 58–60  
running programs, 46–50  
server connection, 573–574  
sets, 423–425  
string formatting, 348–349  
text and numeric variables, 80  
`Text` object, 527–528  
text representation, 37  
user interface, 500–504  
variables, 75  
version management, 344–345  
`yield` statement, 452–453

IDLE debugger, 202–208  
IDLE editor, debugger, 203–208  
IDLE environment  
    alternatives, 15  
    configuring, 54  
    creating programs, 46–50  
    opening, 10–13  
`if` conditions, nesting, 127–128  
`if` construction, 125  
    combining statements, 119–123  
    comparing strings, 125–126  
    conditions, 119  
    definition, 122  
    else part, 125, 131  
    fortune teller, 137  
    limit, 139  
    using, 118–119  
images, displaying in snaps, 67–68  
images in pygame  
    file types, 601–602  
    loading into games, 602–604  
    moving, 604–605  
“immutable,” discovering, 284–287  
immutable behavior, 257, 305–307  
`import` command, 58  
`import` statement, 201–202  
indenting text, 121–122  
index values, inadequacy, 252–253  
`index.html` page, 582–583  
information and data, 31–35, 41–42  
inheritance, 379, 381–384  
    `__init__` initializer method, 295–299, 311  
initializer method, 295–297, 306  
`InitName` class, 296–297

`InitPrint` class, 295

input

    and output, 24–25, 51–54

    validating, 149

`input` function

    “greeter” program, 85

    and Python versions, 86

    return values, 185–186

    using, 107

    using to read in text, 84–85

installer program, 8–10

installing Visual Studio Code, 490–492

instances, of classes, 284

instrumented stock items, 402–405

`int` and float, converting between, 98–99

`int` data type, immutability, 286

`int` function, 87–88, 96, 152–153, 247

integer division, 94–95

integer values, converting strings to, 87

interactive help, 182. *See also* help message

Internet ports and hosts, 552

Internet protocols, 552, 567

interpreting programs, 30

Interrupt Execution, 144

invalid number entry, detecting, 152–154

`invalid syntax` error, 55

invalid user entry, handling, 147–149. *See also* errors

IP addresses, 557–558, 561

`isinstance` function, 178–179

iteration, 356–362, 371

iterator, 369

iterator functions, yield statement, 451–456. *See also* functions; methods

J

`join` function, 369  
`join` method, 361–363  
JPEG format, 601

## K

keyboard shortcuts  
interrupting messages, 558  
New Window, 46  
stopping programs, 144, 153, 156, 193  
keyword arguments, 180–181  
killer tomato, adding with pygame, 609, 625–628  
Kivy GUI (Graphical User Interface), 547

## L

lambda expressions, 446–450, 484. *See also* expressions  
`len` function, 231, 259  
less than (`<`) operator, 112  
less than or equals (`<=`) operator, 112  
libraries  
    decimal, 92  
    fraction, 92  
    function names, 70  
    os, 240  
    path, 240  
    putting functions in, 209  
    pydoc, 196  
    Pygame, 65  
    random, 57–60  
    snaps, 66–69  
    time, 60–61, 109  
line feed/new line, escape sequence, 82  
lines, drawing in pygame, 594–598  
`Listbox` object, 537–539  
`listen_address` tuple, 554

list-reading loop, 218–219

lists

containing lists, 252

creating, 215–217

dimensions, 255

displaying using for loop, 219–221

function references, 445–446

initializing with test data, 228

lookup tables, 255–256

and loops, 230–231

reading in, 218

recording with save function, 251

and references, 282–284

sorting high to low, 228–233

storing contact data, 269–270

`sum` function, 460

versus tables of data, 251

and tuples, 256–257

using, 260–261

`week_sales`, 252

literal values, 84

`load` and `save` behaviors, 289, 293–294

`load` method, 413–414

`load_contacts` function, 292

`load_into_editor` method, 533–534

`load_sales` function, 246–247

local variables, 189–190. *See also* variables

`localtime` function, 109–111

logic, working with, 128

logic errors, preventing, 77

lookup tables, lists as, 255–256

loop counting, 254

loops

best practices, 168

breaking out of, 157–159  
continue keyword, 158–159  
continuous, 144–145  
countdown program, 147  
exceptions, 155  
faults, 150  
list-reading, 218–219  
and lists, 230–231  
messages, 145  
nesting, 237–238  
print statements, 145–146  
range function, 168  
repeating, 159–160  
selection program, 147  
using with tables, 253–254  
validating input, 149  
`lower()` method, 126–127

## M

`mainloop`, creating for GUI, 511–512  
`make_page` method, 589  
`make_test_data` function, 253  
`map` function, 355–361, 369, 440  
markup languages, 568  
Mary’s Fashion Shop. *See* `Fashion Shop` application  
matching names, 272  
memory, adding via variables, 74  
menu items, adding, 238–239. *See also* `user menu`  
message board, 584–585  
message box, displaying in GUI, 514  
messages  
    interrupting, 558  
    printing from escape sequences, 84  
    printing from programs, 52

sending to computers, 557–558  
method attributes, creating for classes, 314–316. *See also* attributes  
method overriding, 392, 437  
methods. *See also* functions; protected methods; static methods; validation and methods  
`add_session`, 316, 321–322  
`blit`, 602–603  
`check_version`, 342, 345  
in classes, 368  
`close`, 240  
`display_contact`, 346–347  
`end_game`, 634  
`exit`, 630  
`format()`, 348  
and functions, 126–127  
`get_from_editor`, 534–535  
`get_hours_worked`, 315  
initializer, 295–297  
`load`, 413–414  
`load_into_editor`, 533–534  
`lower()`, 126–127  
`make_page`, 589  
overriding in classes, 388–392  
`play_note`, 364  
`recvfrom`, 554–555, 557–558  
`render`, 631  
`reset`, 618  
`sendto`, 555, 557–558  
`session_report`, 355, 361  
`setter`, 333  
`__str__`, 369  
`tick`, 605  
`upper()`, 126–127  
`valid_session_length`, 320–321

`write`, 240–242  
`min` and `max` values, 200  
mobile phones, microcomputers, 23  
mode strings, 242  
modules  
    converting functions to, 201–202  
    detecting execution as programs, 463–464  
    features, 460–461  
    importing from packages, 466–470  
    making, 465  
    putting functions in, 209  
    running as programs, 462, 483  
    `socket`, 553–555  
    `unittest`, 472–477  
MTU (maximum transmission unit), 568  
multi-line text, entering in GUI, 526–528. *See also* `text`  
multiplication (\*), 29  
music player, creating, 363–367

## N

`\n` (new line) character, 82, 241, 260  
name attributes, 274  
names, matching, 272  
naming  
    arguments, 182  
    programs, 50  
    variables, 76–77  
“Nerves of Steel” party game, 69  
nesting  
    `if` conditions, 127–128  
    loops, 237–238  
network communication, 550–551  
network layers, 551  
network messages, sending, 552–555

network problems, 551–552, 560  
networking  
    address messages, 550–551  
    connections and datagrams, 561  
    hosts and ports, 552  
    routing packets, 558–559  
    sending messages, 557–558  
networks, broadcast address, 551  
networks and addresses, 561  
new line (`\n`) character, 82, 241, 260  
`new_contact` function, 269–270, 276  
`None` value, 187, 208  
not equals (`!=`) operator, 112  
`not` operator, 115  
`Note` class, 365–366  
notes, playing, 363  
number input function, 197–198, 202  
numbers. *See also* `bin` function  
    adding to strings, 34  
    converting to text, 38  
    and exceptions, 154  
    floating-point, 90–92  
    reading, 88  
    storing, 90  
    strings and integer values, 87  
    and text, 35  
    whole and real, 89  
numeric values and text, 80

## O

object-oriented design, 376–379, 436  
objects  
    cohesion, 312–313  
    as components, 409–410

versus components, 435  
features, 484  
initializer methods, 306  
references, 306  
self-contained, 368  
storage in memory, 306  
using, 284  
one-handed clock, making, 110–111  
`open` function, 239–240, 260  
operands and operators, 27  
`or` operator, 115  
`ord` function, 36–37. *See also* text  
os library, 240  
output and input, 24–25, 51–54  
ovals, drawing, 526  
overriding in classes. *See* method overriding  
overwriting files, 240

## P

packages  
    creating, 464–465  
    importing modules from, 466–470  
    moving, 484  
packets, routing, 558–559  
padding, 509  
`page.html` page, 582–583  
`Pants` class, 378, 380, 383  
parameters  
    and arguments, 176–179  
    default values, 181–182  
    in functions, 179–180  
    using, 208  
        as values, 183  
parentheses (`( )`), 30, 33

party games, making, 69, 78–79  
party guests, reading names, 221  
party planning, 18  
`pass` keyword, 225  
`path` library, 240  
performance, improving, 233–234  
Petzold, Charles, 31–32  
piano keys, mapping notes, 364  
.pickle extension, 290  
pickling, 289–292  
pip program, 65  
pizza order, calculating, 99–100  
placeholder functions, 224–225  
`play_note` method, 364  
`play_sound` function, 363  
player sprite, 614–617. *See also* `sprites`  
PNG format, 601  
polymorphism, 394–395  
ports and hosts, 552  
positional arguments, 180–181  
`POST` request, 585–588, 590  
PowerShell command prompt, 478–482  
precision versus accuracy, 92  
Preferences, selecting, 54  
prices dictionary, 300–301  
`print` function  
    default behavior, 184  
    Python versions, 56–57  
    using, 51–57  
`print` statements  
    conditions, 123–124  
    using, 81–82  
    while construction, 143  
`print_sales` function, 223–224

`print_times_table` function, 177–181  
printing messages, 84  
problems, solving, 20–21, 42  
program context, checking, 463  
program testing  
    assert statement, 471–472  
    elements, 470–471  
    importance, 253, 545  
    `unittest` module, 472–476  
programming  
    concepts, 19  
    languages, 4  
    and party planning, 18–19  
    and problems, 19–21  
programming languages, 41–42, 287  
programs. *See also* data-processing applications; “defensive programming”  
    versus applications, 14  
    breaking, 15  
    and computers, 22  
    as data processors, 24–25  
    delaying end of, 64  
    errors, 43  
    expressions in, 48  
    interpreting, 30  
    naming, 50  
    and recipes, 19, 25  
    refactoring into programs, 221–222  
    running, 46–50  
    saving, 49–50  
    stopping, 144, 153, 156, 193  
    testing, 253  
    understanding, 100  
properties, using with classes, 332–336, 369–370  
property code, failures, 336  
protected methods, 331. *See also* methods

protecting

    data attributes, [328–331, 368–369](#)

    data in class hierarchy, [395–396](#)

prototyping, [21, 267–269](#)

Pycharm, [499](#)

pydoc library, [196](#)

pydoc program, [478–483](#)

pygame

    acceleration, [626](#)

    “attract mode” screen, [637](#)

**blit** method, [602–603](#)

    bounding boxes, [620](#)

    catching crackers, [620–623](#)

    collision detection, [624](#)

**cracker** sprite, [618](#)

    draw and update behaviors, [636–637](#)

    drawing lines, [594–598](#)

    drawing text, [630–633](#)

    ending games, [634](#)

    events, [606–607](#)

    frame rates, [605](#)

    game loops, [608–609](#)

    image file types, [601–602](#)

    killer tomato, [625–628](#)

    loading images, [602–604](#)

    making images move, [604–605](#)

**player** sprite, [614–617](#)

**render** method, [631–632](#)

    sameness of gameplay, [637](#)

    scoring games, [635](#)

**Sound** class, [623](#)

**sprite** instances, [619–620](#)

    sprites, [609–614](#)

    start screen, [629–633](#)

starting, 594–598  
`tick` method, 605  
user input, 606–608

Pygame library, adding, 65

PyQT GUI (Graphical User Interface), 547

Python

- conversation, 26–27
- as data processor, 51
- downloading, 7
- origins, 4
- overview, 4
- as programming language, 71
- as scripting language, 30
- shutting down, 630
- starting, 10–13
- tools, 6–7
- versions, 4–5

Python Command Shell, 26, 47, 70. *See also* IDLE Command Shell

Python libraries

- `decimal`, 92
- `fraction`, 92
- function names, 70
- `os`, 240
- `path`, 240
- putting functions in, 209
- `pydoc`, 196

Pygame, 65

- `random`, 57–60
- `snaps`, 66–69
- `time`, 60–61, 109

Python versions

- and input function, 86
- integer division, 94–95
- `print` function, 56–57

Python web server, 577–579

## Q

quotes and strings, 81

## R

\r escape sequence, 82

`randint` command, 58–60, 137

random library, 57–60

random number generation, 618, 637

`range` function

    iterator function, 451

    using with for loop, 163, 168

`raw_input` function, 86

`read_float` function, 197–198

`read_float_ranged` function, 198–200

`read_sales` function, 223–224

`read_text` function, 194–195, 198, 268

reading

    from files, 244–246

    sales figures, 246–247

`readme` function, adding to `BTCInput`, 461

real numbers, 89–92

`real_number` function, 443

recipes and programs, 19, 25

recursion, 175

`recvfrom` method, 554–555, 557–558

refactoring

`find_contact` function, 280

    programs, 279–280

refactoring programs, programs, 221–223

references and lists, 282–284. *See also* class references

references to functions. *See* function references

`render` method, 631  
repeating  
    loops, 159–160  
    sequences of statements, 142–143  
`reset` method, pygame, 618  
`return` statement  
    `find_contact` function, 280  
    using, 208  
        using with functions, 186–187, 195  
reusable functions, 193–194. *See also* functions  
routing packets, 558–559  
RSS (Really Simple Syndication/Rich Site Summary), 562, 564  
running  
    programs, 46–50, 71  
    Python from desktop, 63–64

## S

sales, total and average, 236–237. *See also* `week_sales` list  
sales analysis program, functions, 223–224  
sales figures  
    reading, 246–247  
    writing, 242–243  
`save` and `load` behaviors, 289, 293–294  
`save` function, recording lists, 251  
`save` requests, eliminating, 54  
`save_contacts` function, 291  
`save_sales` function, 242–243  
saving  
    contacts, 289–291  
    files, 49–50  
scripting language, Python as, 30  
search name, removing white space, 271  
`Secret` class, 329–331  
secure code, 331

`seed` function, 637  
selection program, looping, 147  
`self` parameter  
  `StockItemEditor`, 531–532  
  using, 295–296, 315, 368  
  validation and methods, 320  
“Self-Timer” party game, making, 78–79  
`sendto` method, 555, 557–558  
serializers, 293  
server handler, 583  
server program, 578–579  
`session` class, 351–353  
session list, 355  
session record, adding, 354  
session tracking, 350. *See also* Time Tracker  
  `join` method, 361–363  
  `map` function, 355–361  
  specification, 350–353  
`session_report` method, 355, 361  
sets. *See also* `FashionShop` object; values  
  versus class hierarchies, 431–433  
  creating from strings of text, 427–429  
  investigating, 422–426  
  and tags, 426–432  
  using, 435  
`setter` method, 333  
shadowing, 192  
shell, 11, 51  
`SimpleHTTPRequestHandler`, 583  
single quote (')  
  entering, 81  
  escape sequence, 82  
`sleep` function, 60–61, 95, 167, 184–185  
slicing, 581–584

snaps framework. *See also* [functions](#)  
[`get\_string`](#) function, 134–135  
ride selector, 136

snaps library  
digital clock, 167  
[`display\_image`](#) function, 167  
[`draw\_text`](#) function, 167  
images, 67–68  
input function, 134–136  
in programs, 69, 71  
sounds, 68  
text, 66–67  
weather, 101

socket module, 553–555  
socket-based server, 572–576  
sockets, exceptions, 556  
software, life-threatening capabilities, 24  
[`sort\_high\_to\_low`](#) function, 228  
[`sort\_pass`](#) variable, 232–233  
sorting using bubble sort

alphabetizing, 234  
average sales, 236–237  
completing, 237–238  
highest values, 235–236  
list with test data, 228  
listing high to low, 228–233  
listing low to high, 234–235  
lowest values, 235–236  
total sales, 236–237

[\*\*Sound\*\*](#) class, 623  
sounds, making in snaps, 68  
Source check box, 206  
specifications, 20–21, 43  
[\*\*Sprite\*\*](#) class, 612, 618

`sprite` superclass, 611  
sprites. *See also player sprite*  
    artificial intelligence, 625–626  
    creating, 609–614  
    instances, 619–620  
    intersecting, 620–621  
    physics, 626–627  
    timing, 628  
start screen, adding to game, 629–630  
starting Python, 10–13, 26  
`startswith` function, 272  
statements  
    combining, 119–121  
    repeating sequences, 142–143  
    testing behaviors, 475  
static class attributes, 370. *See also attributes*  
static items, 437  
static methods, 321, 368. *See also methods*  
Step button, 207  
sticky formatting, 508  
stock, adding to items, 407–408  
stock data, listing, 416–417  
stock items  
    creating, 406–407  
    editing, 536  
    finding, 415–416  
    selecting, 541–542  
    selling, 409  
    storing, 414–415  
`StockItem` class, 380–381, 383–384, 387, 529–536  
`StockItem` component, 410  
`StockItem` selector, 539–543  
`StockItemEditor`, creating, 531–532  
stopping programs, 193

storing data. *See pickling*  
storyboarding, 212–213, 237  
`str` function, 243  
`__str__` method in class, 346–349, 369, 388–391  
string formatting, 348–349  
string literal, 33  
strings  
    adding numbers, 34  
    comparing in programs, 125–126  
    converting into floating-point values, 95–96  
    converting to integer values, 87  
    in functions, 209  
    marking start and end, 81–82  
    multiplying by numbers, 35  
    and number variables, 80  
    and quotes, 81  
    reading in snaps framework, 134  
    subtracting, 34  
stub functions, 224, 239  
suite, 122–123  
`sum` function, 460  
`super` function, 387  
superclasses and subclasses. *See also classes*  
    abstraction, 381  
    data in classes hierarchy, 384  
    data protection in class hierarchy, 395–396  
    inheritance, 382–384  
    item names, 387–388  
    method overriding, 388–392  
    polymorphism, 393–394  
    `__str__` method in class, 388–392  
    using, 379–381  
    version management, 392–393  
    using, 434, 436

syntax error, 55–56

# T

\t escape sequence, 82

tab, escape sequence, 82

tables, using loops, 253–254

tables of data, storing, 251–255

tags

filter on, 429–431

and sets, 432

TCP (Transmission Control Protocol), 561

telephone number, storing, 269

teletype printer, creating, 184–185

test contacts, generating, 455–456. *See also* contacts

test data generator, 453–455

testing programs

assert statement, 471–472

elements, 470–471

importance, 253, 545

`unittest` module, 472–476

tests, creating, 476–477, 485

`test.txt` file, 241–242

text. *See also* multi-line text; `ord` function; variables and text

converting numbers to, 38

displaying in snaps, 66–67

drawing in pygame, 630–633

indentation, 121–122

and numbers, 35

reading with input function, 84–85

working with, 32–33

text input function, 193–194

Text object, 527–528

time library, 60–61, 109

Time Tracker. *See also* Contact class; session tracking

`__init__` initializer method, 311  
`__str__` method in class, 346–347  
class properties, 332–336  
class variables, 317–318  
class versions, 340–345  
cohesive object, 312–313  
`Contact` object, 311  
creating, 310  
data attribute for class, 311–312  
evolve class design, 337–340  
exceptions, 323–324  
`get_hours_worked` method, 315–316  
`join` method, 361–363  
`map` function, 355–361  
method attributes for class, 314–316  
`play_sound` function, 363–367  
protected methods, 331  
protecting data attributes, 328–331  
raise exception for error, 323–326  
session tracking, 350–355  
status messages from validation method, 321–322  
string formatting, 348–349  
validating values, 318–321  
validation for methods, 316–317  
version management, 344–345  
time value table, 110  
`time_text` variable, 88  
Times Table Tutor, 161, 166  
Tiny Contacts app. *See* `Contacts` app  
Tkinter. *See also* `GUI` (Graphical User Interface)  
considering, 547  
events, 522–523  
`GUI` (Graphical User Interface), 499–506  
user interface, 500–504

tomato, adding with pygame, 625–628  
tools, downloading and installing, 6–7  
`total` variable, 74–75  
True and False values, 106–109, 115–119  
`try` construction, 156–157  
`try.except.finally` construction, 249  
tuples  
    `listen_address`, 554  
    note and duration values, 365  
    using, 257–261  
`type` function, 285  
typing errors and testing, 77–78

## U

UDP (User Datagram Protocol), 552–555  
`UNICODE`, 83  
`unittest` module, 472–477  
Untitled program, running, 47–49  
`upper()` method, 126–127  
URL (Uniform Resource Locator), 579–580  
`urlopen` object, 562  
user authentication, 591  
user input, testing, 132–133  
user interface. *See also GUI (Graphical User Interface)*  
    designing, 129–130  
    implementing, 130–131  
user interface component, 417–421  
user menu, creating, 225–227. *See also menu items*

## V

`valid_session_length` method, 320–321  
validating input, 149, 151  
validation and methods. *See also methods*  
    `add_session`, 316

adding to methods, 316–326  
class variables, 317–318  
decorators, 321  
returning status messages, 321–322  
static method for values, 318–321  
`ValueError` exception, 153  
values. *See also* `sets`  
    and attributes, 109  
    comparing, 111–112  
    holding, 31  
    working with, 287  
van Rossum, Guido, 5  
variables. *See also* `Boolean variables`; `global variables`; `local variables`  
    assignment statements, 74  
    creating, 285  
    explained, 74  
    floating-point, 92–94  
    identifying, 88  
    length of names, 103  
    limitations, 214–215  
    naming, 76–77  
    overwriting, 103  
    storing, 90  
    swapping values, 229–233  
    working with, 75  
variables and text. *See also* `text`  
    escape characters, 82  
    numeric values, 80  
    working with, 79  
version attributes, adding to classes, 341. *See also* `attributes`  
version control, 293  
version management  
    Fashion Shop application, 392–393  
    Fashion Shop program, 392–393  
    implementing, 344–345, 369

version numbers, checking, 342

Visual Studio Code. *See also* IDLE Command Shell

Community Edition, 499

debugging program, 494–498

editors, 499

installing, 490–492

interpreter, 498

program file, 493–494

project folder, 492–493

## W

weather conditions, displaying, 102

weather data, 566

weather helper, 136–137

weather snaps, 101

web applications, 590

web servers, 575–576, 591

web users, getting information from, 584–589

web-based data, 562–566

`webPageHandler` class, 588

webpages

features, 590–591

making from Python code, 589

reading, 562

serving from files, 579–584

websites

Django framework, 590

encryption, 591

Flask framework, 590

Kivy GUI (Graphical User Interface), 547

Pygame library, 65

PyQT GUI (Graphical User Interface), 547

security, 591

tools, 7

US National Weather Service, [101](#)  
Visual Studio Code, [490](#)  
Windows PC version, [7–9](#)  
Wireshark program, [591](#)  
`week_sales` list, [252](#). *See also* sales `what_would_I_do` function, [183](#)  
`while` construction  
    versus for loop, [230–231](#)  
    looping, [155](#)  
    using, [142–147](#), [168](#), [195](#)  
white space, removing from search name, [271](#)  
whole numbers, [89](#)  
windows, opening, [46](#)  
Windows PC version, [7–9](#)  
wireless devices, [567](#)  
Wireshark program, [591](#)  
`with` construction, [249–250](#), [261](#)  
`write` method, [240–242](#)  
writing into files, [260](#)

## X

XML (eXtensible Markup Language), [562–565](#), [568](#). *See also* `ElementTree` class

## Y

`yield` statement, iterator functions, [451–456](#)

## Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

"Drink your medicine after a hot bath."

Step1: Take a hot bath  
Step2: Drink your medicine

```
>>> 2
2
>>> 2+2
4
>>> 2+
SyntaxError: invalid syntax
```

```
>>> 'he11o  
SyntaxError: EOL while scanning string literal
```

```
>>> 'hello' - ' world'  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    'hello' - ' world'  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

```
>>> 'hello' + 2
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    'hello' + 2
TypeError: must be str, not int
```

```
>>> ord(W)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    ord(W)
NameError: name 'W' is not defined
```

===== RESTART: C:\Users\Rob\Documents\Python programs\firstProg.py =====

The answer is:

4

```
print('The answer is:', 2+2)
```

```
===== RESTART: C:\Users\Rob\Documents\Python programs\firstProg.py ======
```

The answer is: 4

```
===== RESTART: C:\Users\Rob\Documents\Python programs\firstProg.py =====
The answer is:
Traceback (most recent call last):
  File "C:\Users\Rob\Documents\Python programs\firstProg.py", line 2, in <module>
    Print(2+2)
NameError: name 'Print' is not defined
```

```
>>> random.randint(1, 6)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    random.randint(1, 6)
NameError: name 'random' is not defined
```

```
>>> import Random
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import Random
ModuleNotFoundError: No module named 'Random'
```

```
import random
print('You have rolled: ', random.randint(1, 6))
```

```
import time
print('I will need to think about that..')
time.sleep(5)
print('The answer is: 42')
```

```
time.sleep(300) # sleep while the egg cooks (300 seconds, or 5 minutes)
```

```
print('Put the egg in boiling water now') # print put the egg in boiling water now
```

```
# EG3-01 Throw a single die
import random
print('You have rolled: ' + random.randint(1, 6))
```

```
py -m pip install pygame --user
```

```
python3 -m pip install pygame --user
```

```
snapshots.display_message('This is smaller text in green on the top left',  
                           color=(0,255,0),size=50,horiz='left',vert='top')
```

```
# EG3-05 housemartins

import snaps

snaps.display_image('Housemartins.jpg')

snaps.display_message('Hull Rocks',color=(255,255,255), vert='top')
```

```
# EG3-06 Ding

import snaps

snaps.play_sound('ding.wav')
```

```
total = us_sales + world_wide_sales
```

```
message = 'the name is '+customer_name
```

```
>>> customer_age_in_years = 25  
>>> customer_name = 'Fred'
```

```
>>> customer_age_in_years+customer_name
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    customer_age_in_years + customer_name
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> customer_age_in_years='Fred'
```

```
print(''...and then Luke said "It's a trap"''')
```

...and then Luke said "It's a trap"

```
print('Welcome to Nerves of Steel')
print()
print('Everybody stand up')
print('Stay standing as long as you dare.')
print('Sit down just before you think the time will end.')
```

```
print('''Welcome to Nerves of Steel  
  
Everybody stand up  
Stay standing as long as you dare.  
Sit down just before you think the time will end. ''')
```

```
print('Item\tSales\nCar\t50\nBoat\t10')
```

```
print('...and then Luke said "It\'s a trap"')
```

```
name=input('Enter your name please: ')
```

```
input('Press Enter to continue')
```

```
name = input('Enter your name please: ')
print('Hello', name)
```

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    name=input('Enter your name please: ')
  File "<string>", line 1, in <module>
NameError: name 'Rob' is not defined
```

```
name = raw_input('Enter your name please: ')
```

```
# EG4-02 User Configurable Egg Timer

import time

time_text = input('Enter the cooking time in seconds: ')

time_int = int(time_text)

print('Put the egg in boiling water now')

time.sleep(time_int)

print('Take the egg out now')
```

```
time_int = int(input('Enter the cooking time in seconds: '))
```

```
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    x = int('kaboom')
ValueError: invalid literal for int() with base 10: 'kaboom'
```

```
from __future__ import division
```

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

How many students: 40

You will need 26.666666666666668 pizzas

```
pizza_count = (students_int/1.5)+1
```

```
# EG4-06 Seattle Weather

import snaps

desc=snaps.get_weather_desciption(latitude=47.61, longitude=-122.33)
print('The conditions are:',desc)
```

```
it_is_time_to_get_up = True
```

```
it_is_time_to_get_up = False
```

```
print(it_is_time_to_get_up )
```

```
>>> x=input("True or False: ")  
True or False: True
```

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool(0.1)
True
>>> bool('')
False
>>> bool('he11o')
True
```

```
>>> 'Hello'+True
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    'Hello'+True
TypeError: must be str, not bool
```

```
it_is_time_to_get_up = hour>6
```

```
it_is_time_to_get_up = hour>6
```

```
it_is_time_to_get_up = hour>6 and minute>29
```

```
it_is_time_to_get_up = (hour>7) or (hour==7 and minute>29)
```

```
if it_is_time_to_get_up:  
    print('IT IS TIME TO GET UP')
```

```
if (hour>7) or (hour==7 and minute>29):  
    print('IT IS TIME TO GET UP')
```

```
if (hour>7) or (hour==7 and minute>29):  
    print('IT IS TIME TO GET UP')  
    print('RISE AND SHINE')  
    print('THE EARLY BIRD GETS THE WORM')  
print('The time is',hour,':',minute)
```

```
if hour > 6:print('IT IS TIME TO GET UP');print('THE EARLY BIRD GETS THE WORM')
```

```
if hour > 6:print('IT IS TIME TO GET UP'); print('RISE AND SHINE')
    print('THE EARLY BIRD GETS THE WORM')
SyntaxError: unexpected indent
```

```
>>> if True:  
    print('True')  
    print('Still true')
```

```
>>> if True:  
    print('True')  
    print('Still true')
```

True  
Still true

```
>>> name = 'Rob'  
>>> name.upper  
<built-in method upper of str object at 0x0000021CDA0FE880>
```

Welcome to our Theme Park

These are the available rides:

1. Scenic River Cruise
2. Carnival Carousel
3. Jungle Adventure Water Splash
4. Downhill Mountain Run
5. The Regurgitator

Please enter the ride number you want: 1

You have selected the Scenic River Cruise

There are no age limits for this ride

```
# EG5-10 Ride Selector Start

print('''Welcome to our Theme Park

These are the available rides:

1. Scenic River Cruise
2. Carnival Carousel
3. Jungle Adventure Water Splash
4. Downhill Mountain Run
5. The Regurgitator
'''')

ride_number_text = input('Please enter the ride number you want: ')
ride_number = int(ride_number_text)

if ride_number == 1:
    print('You have selected the Scenic River Cruise')
    print('There are no age limits for this ride')
```

```
if ride_number == 1:  
    print('You have selected the Scenic River Cruise')  
    print('There are no age limits for this ride')  
else:  
    # We need to get the age of the user
```

```
if ride_number == 1:  
    print('You have selected the Scenic River Cruise')  
    print('There are no age limits for this ride')  
else:  
    # We need to get the age of the user  
    age_text = input('Please enter your age: ')  
    age = int(age_text)
```

```
if ride_number == 2:  
    print('You have selected the Carnival Carousel')  
    if age >= 3 :  
        print('You can go on the ride.')  
    else:  
        print('Sorry. You are too young.')
```

```
if ride_number == 3:  
    print('You have selected the Jungle Adventure Water Splash')  
    if age >= 6:  
        print('You can go on the ride.')  
    else:  
        print('Sorry. You are too young.')
```

```
if ride_number == 5:  
    print('You have selected The Regurgitator')
```

```
if ride_number == 5:  
    print('You have selected The Regurgitator ')  
    if age >= 12:  
        # Age is not too low  
        if age > 70:  
            # Age is too high  
            print('Sorry. You are too old.')  
        else:  
            # Age is in the correct range  
            print('You can go on the ride.')  
    else:  
        # Age is too low  
        print('Sorry. You are too young.')
```

```
# EG5-12 Snaps get_string function

import snaps

name = snaps.get_string('Enter your name: ')
snaps.display_message('Hello ' + name)
```

```
# EG5-13 Theme Park Snaps Display

import snaps

snaps.display_image('themepark.png')

prompt = '''These are the rides that are available

1: Scenic River Cruise
2: Carnival Carousel
3: Jungle Adventure Water Splash
4: Downhill Mountain Run
5: The Regurgitator

Select your ride: '''

ride_number_text = snaps.get_string(prompt,vert='bottom',
                                     max_line_length=3)

confirm='Ride ' + ride_number_text
snaps.display_message(confirm)
```

```
# EG5-14 Weather helper

import snaps

temp = snaps.get_weather_temp(latitude=47.61, longitude=-122.33)

print('The temperature is:', temp)

if temp < 40:
    print('Wear a coat - it is cold out there')
elif temp > 70:
    print('Remember to wear sunscreen')
```

```
import random
if random.randint(1,6)<4:
    print('You will meet a tall, dark stranger')
else:
    print('You will not meet anyone at all')
```

```
>>> while True:  
    print('Loop')  
Loop  
Loop  
Loop  
LoopTraceback (most recent call last):  
  File "<pyshell#8>", line 2, in <module>  
    print('Loop')  
KeyboardInterrupt  
>>>
```

```
while True:  
    print('Inside loop')  
print('Outside loop')
```

```
while False:  
    print('Inside loop')  
print('Outside loop')
```

```
# EG6-01 Loop with boolean flag
flag = True
while flag:
    print('Inside loop')
    flag = False
print('Outside loop')
```

```
flag = True
while flag:
    print('Inside loop')
    Flag = False
print('Outside loop')
```

```
# EG6-02 Loop with counter
count = 0
while count < 5:
    print('Inside loop')
    count = count+1
print('Outside loop')
```

```
1. # We need to get the age of the user
2. age_text = input('Please enter your age: ')
3. age = int(age_text)
4. while age < 1 and age > 95:
5.     # repeat this code while the age is invalid
6.     print('This age is not valid')
7.     age_text = input('Please enter your age: ')
8.     age = int(age_text)
9. # when we get here, we have a valid age value
10. print('Thank you for entering your age')
```

4. **while** age < 1 **or** age > 95:

```
1. #EG6-04 Handling invalid text
2. ride_number_valid = False           # create a flag value and set it to False
3. while ride_number_valid == False:   # repeat while the flag is False
4.     try:                           # start of code that might throw exceptions
5.         ride_number_text = input('Please enter the ride number you want: ')
6.         ride_number = int(ride_number_text) # convert the text into a number
7.         ride_number_valid = True # if we get here, we know the number is OK.
8.     except ValueError:          # the handler for an invalid number
9.         print('Invalid number text. Please enter digits.') # display an error
10.    # When we get here, we have a valid ride number
11.    print('You have selected ride', ride_number)
```

```
Please enter the ride number you want:
```

```
Traceback (most recent call last):
```

```
  File "C:/Users/Rob/OneDrive/Begin to code Python/Part 1 Final/Ch 06 Loops/code  
/samples/#EG6-04 Handling invalid text.py", line 5, in <module>
```

```
    ride_number_text = input('Please enter the ride number you want: ') # read in  
    some text
```

```
KeyboardInterrupt
```

```
1. #EG6-04 Handling invalid text
2. ride_number_valid = False          # create a flag value and set it to False
3. while ride_number_valid == False:  # repeat while the flag is False
4.     try:                          # start of code that might throw exceptions
5.         ride_number_text = input('Please enter the ride number you want: ')
6.         ride_number = int(ride_number_text) # (might raise exception)
7.         ride_number_valid = True # if we get here, we know the number is OK.
8.     except ValueError:           # the handler for an invalid number
9.         print('Invalid number text. Please enter digits.') #
10.    except KeyboardInterrupt:    # the handler for an invalid number
11.        print('Please do not try to stop the program.') #
12.    # When we get here, we have a valid ride number
13.    print('You have selected ride', ride_number)
```

```
1. # EG6-06 Using break to exit loops
2. while True:      # repeat until we break out of the loop
3.     try:                  # start of code that might throw exceptions
4.         ride_number_text = input('Please enter the ride number you want: ')
5.         ride_number = int(ride_number_text) # (might raise exception)
6.         break                 # number OK - break out of loop
7.     except ValueError:        # the handler for an invalid number
8.         print('Invalid number text. Please enter digits.') # display error
9. # When we get here, we have a valid ride number
10. print('You have selected ride', ride_number)
```

```
1. # EG6-07 Loop with condition ending early
2. count=0
3. while count<5:
4.     print('Inside loop')
5.     count = count+1
6.     if count == 3:
7.         break
8. print('Outside loop')
```

```
# EG6-09 Times Table Tutor
count = 1
times_value = 2
while count < 13:
    result = count * times_value
    print(count,'times', times_value,'equals',result)
    count = count + 1
```

```
1. # EG6-09 Times Table Tutor
2. count = 1
3. times_value = 2
4. while count < 13:
5.     result = count * times_value
6.     print(count,'times', times_value,'equals',result)
7.     count = count + 1
```

```
names=('Rob','Mary','David','Jenny','Chris','Imogen')
```

```
# EG6-10 Name printer
names=('Rob', 'Mary', 'David', 'Jenny', 'Chris', 'Imogen')
for name in names:
    print(name)
```

```
1. # EG6-12 Code Analysis 1
2. for count in range(1, 13):
3.     if count == 5:
4.         break
5.     print(count)
6. print('Finished')
```

```
1. # EG6-13 Code Analysis 2
2. for count in range(1, 13):
3.     if count == 5:
4.         continue
5.     print(count)
6. print('Finished')
```

```
1. # EG6-14 Code Analysis 3
2. for count in range(1, 13):
3.     count = 13
4.     print(count)
5. print('Finished')
```

```
1. # EG6-15 Code Analysis 4
2. while True:
3.     break
4. print('Finished')
```

```
1. # EG6-16 Code Analysis 5
2. while True:
3.     continue
4.     print('Looping')
```

```
1. # EG6-17 Code Analysis 6
2. for letter in 'hello world':
3.     print(letter)
```

```
>>> def greeter():
        print('Hello')
```

```
>>>
```

```
# EG7-01 Pathfinder
def m2():
    print('the')

def m3():
    print('sat on')
    m2()

def m1():
    m2()
    print('cat')
    m3()
    print('mat')

m1()
```

```
def print_times_table(times_value):
```

```
# EG7-02 Times Table
def print_times_table(times_value):
    count = 1
    while count < 13:
        result = count * times_value
        print(count, 'times', times_value, 'equals', result)
        count = count + 1

print_times_table(6)
```

1 times six equals six

2 times six equals sixsix

3 times six equals sixsixsix

4 times six equals sixsixsixsix

5 times six equals sixsixsixsixsix

6 times six equals sixsixsixsixsixsix

7 times six equals sixsixsixsixsixsix

8 times six equals sixsixsixsixsixsixsix

9 times six equals sixsixsixsixsixsixsix

10 times six equals sixsixsixsixsixsixsixsix

11 times six equals sixsixsixsixsixsixsixsix

12 times six equals sixsixsixsixsixsixsixsix

```
result = count * times_value
```

```
Traceback (most recent call last):
  File "C:/EG7-03 Safer Times Table.py", line 11, in <module>
    print_times_table('six')
  File "C:/ EG7-03 Safer Times Table.py", line 4, in print_times_table
    raise Exception('print_times_table only works with integers')
Exception: print_times_table only works with integers
```

```
# EG7-04 Two Parameter Times Table
def print_times_table(times_value, limit):
    count = 1
    while count < limit+1:
        result = times_value * count
        print(count, 'times', times_value, 'equals', result)
        count = count + 1
```

```
print_times_table(6, 5)
```

1 times 6 equals 6  
2 times 6 equals 12  
3 times 6 equals 18  
4 times 6 equals 24  
5 times 6 equals 30

```
print_times_table(12, 7)
```

```
# EG7-05 Keyword Arguments  
print_times_table(times_value=12, limit=7)
```

```
print_times_table(limit=7, times_value=12)
```

```
print_times_table(times_value=7)
```

```
# EG7-07 Parameters as values
def what_would_I_do(input_value):
    input_value = 99

    test = 0
    what_would_I_do(test)
print('The value of test is', test)
```

```
def teletype_print(text, delay=0.1):
```

```
for ch in text:  
    print(ch)  
    time.sleep(delay)
```

```
name = input('Enter your name please : ')
```

```
def get_value(prompt, value_min, value_max):
```

```
ride_number=get_value(prompt='Please enter the ride number you want:',  
value_min=1, value_max=5)
```

```
# EG7-08 get_value investigation 1
def get_value(prompt, value_min, value_max):
    return 1
    return 2

ride_number=get_value(prompt='Please enter the ride number you want:',
value_min=1,value_max=5)
print('You have selected ride:',ride_number)
```

You have selected ride: 1

```
# EG7-09 get_value investigation 2
def get_value(prompt, value_min, value_max):
    return

ride_number=get_value(prompt='Please enter the ride number you want:',
value_min=1,value_max=5)
print('You have selected ride:', ride_number)
```

You have selected ride: None

```
ride_number=get_value(prompt='Please enter the ride number you want:',  
value_min=1, value_max=5)  
print('You have selected ride:', ride_number)
```

```
# EG7-11 Local Variables

def func_2():
    i = 99

def func_1():
    i = 0
    func_2()
    print('The value of i is: ', i)

func_1()
```

```
name = read_text(prompt='Please enter your name: ')
```

Please enter your name: Rob

```
def read_text(prompt='Please enter some text: '):
```

Please enter some text: Rob

```
1. def read_text(prompt):
2.     while True: # repeat forever
3.         try:
4.             result=input(prompt) # read the input
5.             # if we get here, no exception was raised
6.             # break out of the loop
7.             break
8.         except KeyboardInterrupt:
9.             # if we get here, the user pressed Ctrl+C
10.            print('Please enter text')
11.    return result
```

```
def read_text(prompt):
    'Displays a prompt and reads in a string of text'
```

```
def read_text(prompt):
    """
    Displays a prompt and reads in a string of text.
    Keyboard interrupts (Ctrl+C) are ignored

    returns a string containing the string input by the user
    """
```

```
>>> import pydoc
>>> pydoc.help(read_text)
Help on function read_text in module __main__:

read_text(prompt)
    Displays a prompt and reads in a string of text.
    Keyboard interrupts (Ctrl+C) are ignored
    returns a string containing the string input by the user
```

```
>>> pydoc.help(print)
Help on built-in function print in module builtins:

print(*args, **kwargs)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
        file: a file-like object (stream); defaults to the current sys.stdout.
        sep: string inserted between values, default a space.
        end: string appended after the last value, default a newline.
        flush: whether to forcibly flush the stream.
```

```
def read_float(prompt):
    """
    Displays a prompt and reads in a number.
    Keyboard interrupts (Ctrl+C) are ignored
    Invalid numbers are rejected
    returns a float containing the value input by the user
    """

    while True: # repeat forever
        try:
            number_text = read_text(prompt)
            result = float(number_text) # read the input
            # if we get here, no exception was raised
            # break out of the loop
            break
        except ValueError:
            # if we get here, the user entered an invalid number
            print('Please enter a number')

    # return the result
    return result
```

```
age=read_float('Please enter your age: ')
```

```
age=read_float_ranged('Please enter your age: ', min_value=5, max_value=90)
```

```
age=read_float_ranged('Enter your age:', min_value=90, max_value=5)
```

```
>>> pydoc.help(read_float_ranged)
Help on function read_float_ranged in module __main__:

read_float_ranged(prompt, min_value, max_value)
    Displays a prompt and reads in a number.
    min_value gives the inclusive minimum value
    max_value gives the inclusive maximum value
    ** Does not detect if max and min are reversed **
    Keyboard interrupts (Ctrl+C) are ignored
    Invalid numbers are rejected
    returns a float containing the value input by the user
```

```
if min_value > max_value:  
    # If we get here, the min and the max  
    # are reversed
```

```
age = read_float_ranged('Enter your age:',min_value=5,max_value=.90)
```

```
if min_value > max_value:  
    # If we get here, the min and the max  
    # are the wrong way around  
    raise Exception('Min value is greater than max value')
```

```
age = BTCInput.read_float_ranged('Enter your age:', min_value=5, max_value=90)
```

```
from BTCInput import read_float_ranged
age = read_float_ranged('Enter your age:', min_value=5, max_value=90)
```

```
from BTCInput import *
age = read_float_ranged('Enter your age:', min_value=5, max_value=90)
```

```
# EG8-01 Finding the largest sales
if sales1>sales2 and sales1>sales3 and sales1>sales4 \
    and sales1>sales5 and sales1>sales6 and sales1>sales7 \
    and sales1>sales8 and sales1>sales9 and sales1>sales10:
    print('Stand 1 had the best sales')
```

```
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    sales[2]
IndexError: list index out of range
```

```
# EG8-02 Read and Display
```

```
from BTCInput import *  
sales = []  
  
for count in range(1,11):  
    prompt = 'Enter the sales for stand ' + str(count) + ': '  
    sales.append(read_int(prompt))  
  
print(sales)
```

Import the number reading functions

Create the sales list

For each stand numbered 1 to 10

Build the prompt string

Read in the sales value for that stand

Print out the sales list

```
for count in range(1, 101):
    prompt = 'Enter the sales for stand ' + str(count) + ': '
    sales.append(read_int(prompt))
```

```
no_of_stands = read_int('Enter the number of stands: ')
for count in range(1, no_of_stands+1):
    prompt='Enter the sales for stand ' + str(count) + ': '
    sales.append(read_int(prompt))
```

```
# EG8-03 Read and Display Loop

#fetch the input functions
from BTCInput import *

#create an empty sales list
sales = []

# read in 10 sales figures
for count in range(1, 11):
    # assemble a prompt string
    prompt='Enter the sales for stand ' + str(count) + ': '
    # read a value and append it to sales list
    sales.append(read_int(prompt))

# print a heading
print('Sales figures')
# initialize the stand counter
count = 1
# work through the sales figures and print them
for sales_value in sales:
    # print an item
    print('Sales for stand', count, 'are', sales_value)
    # advance the stand counter
    count = count + 1
```

```
Enter the sales for stand 1: 50
Enter the sales for stand 2: 54
Enter the sales for stand 3: 29
Enter the sales for stand 4: 33
Enter the sales for stand 5: 22
Enter the sales for stand 6: 100
Enter the sales for stand 7: 45
Enter the sales for stand 8: 54
Enter the sales for stand 9: 89
Enter the sales for stand 10: 75
Sales figures
Sales for stand 1 are 50
Sales for stand 2 are 54
Sales for stand 3 are 29
Sales for stand 4 are 33
Sales for stand 5 are 22
Sales for stand 6 are 100
Sales for stand 7 are 45
Sales for stand 8 are 54
Sales for stand 9 are 89
Sales for stand 10 are 75
```

```
sales=[50,54,29,33,22,100,45,54,89,75]
```

```
def sort_high_to_low():
    """
    Print out a list of the sales figures sorted low to high
    """
    if sales[0]<sales[1]:
        # these two items are in the wrong order
        # the program must swap them
```

```
if sales[0]<sales[1]:  
    # these two items are in the wrong order  
    # the program must swap them  
    sales[0]=sales[1]  
    sales[1]=sales[0]
```

```
if sales[0]<sales[1]:  
    # these two items are in the wrong order  
    # the program must swap them  
    temp=sales[0]  
    sales[0]=sales[1]  
    sales[1]=temp
```

```
if sales[1]<sales[2]:  
    # these two items are in the wrong order  
    # the program must swap them  
    temp=sales[1]  
    sales[1]=sales[2]  
    sales[2]=temp
```

```
1. for count in range(0,len(sales)-1):
2.     if sales[count]<sales[count+1]:
3.         temp=sales[count]
4.         sales[count]=sales[count+1]
5.         sales[count+1]=temp
```

```
# EG8-07 Bubble sort first pass
def sort_high_to_low():
    """
    Print out a list of the sales figures sorted high to low
    """
    for count in range(0,len(sales)-1):
        if sales[count]<sales[count+1]:
            temp=sales[count]
            sales[count]=sales[count+1]
            sales[count+1]=temp
```

```
# EG8-08 Bubble sort multiple passes
def sort_high_to_low():
    """
    Print out a list of the sales figures sorted high to low
    """
    for sort_pass in range(0,len(sales)):
        for count in range(0,len(sales)-1):
            if sales[count]<sales[count+1]:
                temp=sales[count]
                sales[count]=sales[count+1]
                sales[count+1]=temp
    print_sales()
```

**Sales figures**

Sales for stand 1 are 100

Sales for stand 2 are 89

Sales for stand 3 are 75

Sales for stand 4 are 54

Sales for stand 5 are 54

Sales for stand 6 are 50

Sales for stand 7 are 45

Sales for stand 8 are 33

Sales for stand 9 are 29

Sales for stand 10 are 22

```
for sort_pass in range(0,len(sales)):
    for count in range(0,len(sales)-1-sort_pass):
        if sales[count]<sales[count+1]:
            temp=sales[count]
            sales[count]=sales[count+1]
            sales[count+1]=temp
```

```
for count in range(0, len(sales)-1-sort_pass):
```

```
# EG-09 Efficient Bubble Sort
for sort_pass in range(0,len(sales)):
    done_swap=False
    for count in range(0,len(sales)-1-sort_pass):
        if sales[count]<sales[count+1]:
            temp=sales[count]
            sales[count]=sales[count+1]
            sales[count+1]=temp
            done_swap=True
        if done_swap==False:
            break
```

```
# EG8-10 Sort low to high
if sales[count]>sales[count+1]:
    temp=sales[count]
    sales[count]=sales[count+1]
    sales[count+1]=temp
```

```
if(new value > highest I've seen)
  highest I've seen = new value
```

```
highest=sales[0]
for sales_value in sales:
    if sales_value>highest:
        highest=sales_value
```

```
lowest=sales[0]
for sales_value in sales:
    if sales_value<lowest:
        lowest=sales_value
```

```
def highest_and_lowest():
    """
    Print out the highest and the lowest sales values
    """

    highest=sales[0]
    lowest=sales[0]
    for sales_value in sales:
        if sales_value>highest:
            highest=sales_value
        if sales_value<lowest:
            lowest=sales_value
    print('The highest is:', highest)
    print('The lowest is:', lowest)
```

```
# EG8-12 Total Sales
def total_sales():
    """
    Print out the total sales value
    """
    total=0
    for sales_value in sales:
        total = total+sales_value
    print('Total sales are:', total)
```

```
# EG8-13 Average Sales
def average_sales():
    """
    Print out the average sales value
    """
    total=0
    for sales_value in sales:
        total = total+sales_value
    average_sales=total/len(sales)
    print('Average sales are:', average_sales)
```



```
# EG8-14 Complete Program

# Start by reading in the sales
read_sales(10)

# Now get the command from the user

menu= '''Ice-cream Sales

1: Print the sales
2: Sort High to Low
3: Sort Low to High
4: Highest and Lowest
5: Total Sales
6: Average sales
7: Enter Figures

Enter your command: '''

# Now repeatedly read commands and act on them
while True:
    command=read_int_ranged(menu,1,7)
    if command==1:
        print_sales()
    elif command==2:
        sort_high_to_low()
    elif command==3:
        sort_low_to_high()
    elif command==4:
        highest_and_lowest()
    elif command==5:
        total_sales()
    elif command==6:
        average_sales()
    elif command==7:
        read_sales(10)
```

## **Ice-cream Sales**

- 1: Print the Sales**
- 2: Sort High to Low**
- 3: Sort Low to High**
- 4: Highest and Lowest**
- 5: Total Sales**
- 6: Average Sales**
- 7: Enter Figures**
- 8: Save Sales**
- 9: Load Sales**

```
# EG8-15 Load and Save
def save_sales(file_path):
    """
    Saves the contents of the sales list in a file
    file_path gives the path to the file to save
    Raises file exceptions if the save fails
    """
    print('Save the sales in:', file_path)

def load_sales(file_path):
    """
    Loads the sales list from a file
    file_path gives the path to the file to load
    Raises file exceptions if the load fails
    """
    print('Load the sales from:', file_path)
```

```
import os.path
if os.path.isfile('text.txt'):
    print('The file exists')
```

```
output_file.write('First Line\n')
output_file.write('Second Line\n')
output_file.close()
```

```
# EG8-16 File Output

output_file=open('test.txt','w')
output_file.write('line 1\n')
output_file.write('line 2\n')
output_file.close()
```

```
path = 'c:/data/2017/June/sales.txt'
```

```
1. def save_sales(file_path):
2.     """
3.         Saves the contents of the sales list in a file
4.         file_path gives the path to the file to save
5.         Raises file exceptions if the save fails
6.     """
7.     print('Save the sales in:', file_path)
8.     # Open the output file
9.     output_file=open(file_path,'w')
10.    # Work through the sales values in the list
11.    for sale in sales:
12.        # write out the sale as a string
13.        output_file.write(str(sale)+'\n')
14.    # Close the output file
15.    output_file.close()
```

```
input_file=open('test.txt','r')
```

```
# EG8-18 File Input

input_file=open('test.txt','r')
for line in input_file:
    print(line)
input_file.close()
```

```
input_file=open('test.txt','r')
total_file=input_file.read()
print(total_file)
input_file.close()
```

```
1. def load_sales(file_path):
2.     """
3.         Loads the sales list from a file
4.         file_path gives the path to the file to load
5.         Raises file exceptions if the load fails
6.     """
7.     print('Load the sales from:', file_path)
8.     # Clear the sales list
9.     sales.clear()
10.    # Open the file for input
11.    input_file=open(file_path,'r')
12.    for line in input_file:
13.        line=line.strip()
14.        sales.append(int(line))
15.    input_file.close()
```

```
print(week_sales[1][0])
```

Statement 1: week\_sales[0][0] = 50

Statement 2: week\_sales[8][7] = 88;

---

---

Statement 3: week\_sales[7][10] = 100;

```
# EG8-22 Tables of sales data
```

```
total_sales=0
```

Set the total sales to 0

```
for day_sales in week_sales:
```

Work through each day of the week

```
    for sales_value in day_sales:
```

Work through each ice-cream stand

```
        total_sales=total_sales+sales_value
```

Add the sales to the total

```
# Read in a set of sales values
read_sales(10)
# Add the daily sales figures to the week
week_sales.append(sales)
```

```
annual_sales.append(week_sales)
```

Enter the Monday sales figures for stand 2:

```
# EG8-23 Day Name If
if day_number==0:
    day_name= 'Monday'
elif day_number==1:
    day_name= 'Tuesday'
elif day_number==2:
    day_name= 'Wednesday'
elif day_number==3:
    day_name= 'Thursday'
elif day_number==4:
    day_name= 'Friday'
elif day_number==5:
    day_name= 'Saturday'
elif day_number==6:
    day_name= 'Sunday'
```

```
# EG8-24 Day Name List
day_names=['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday']

day_name=day_names[day_number]
```

```
# EG8-25 Day Name Tuple  
day_names[5]='Splatterday'
```

```
Traceback (most recent call last):
  File "C:/Ch 08 Collections/code/samples/EB8-26 Day Name Tuple.py", line 9, in <module>
    day_names[5]='Splatterday'
TypeError: 'tuple' object does not support item assignment
```

```
day_names=('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday')
```

```
def get_treasure_location():
    # get the location from the pirate
    return ('The old oak tree',20,30)
```

```
location=get_treasure_location()
print ('Start at',location[0], 'walk',location[1],'paces north and',
      location[2],'paces east')
```

```
# EG8-26 Pirate Treasure Tuple
def get_treasure_location():
    """
    Get the location of the treasure
    returns a tuple:
    [0] is a string naming the landmark to start
    [1] is the number of paces north
    [2] is the number of paces east
    ...
    # get the location from the pirate

    return ('The old oak tree',20,30)
```

```
# EG8-27 Pirate Treasure Tuple Function
landmark, north, east = get_treasure_location()
print ('Start at',landmark, 'walk', north,'paces north and', east,'paces east')
```

## Tiny Contacts

1. New Contact
2. Find Contact
3. Exit program

Enter your command:

Create new contact

Enter the contact name: Rob Miles

Enter the contact address: 18 Pussycat Mews, London, NE1 410S

Enter the contact phone: +44(1234) 56789

Contact record stored for Rob Miles

[Find contact](#)

Enter the contact name: Rob Miles

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Phone: +44(1234) 56789

Find contact

Enter the contact name: [Fred Bloggs](#)

This name was not found.

```
sales.append(read_int(prompt))
```

```
# Create the lists to store contact information
names=[]
addresses=[]
telephones=[ ]
```

```
def new_contact():
    """
    Reads in a new contact and stores it
    """

    print('Create new contact')
    names.append(read_text('Enter the contact name: '))
    addresses.append(read_text('Enter the contact address: '))
    telephones.append(read_text('Enter the contact phone: '))
```

```
# EG9-03 Tiny Contacts Quick Search
if name.startswith(search_name):
    # if the names match, end the loop
    break
```

```
>>> class Contact:  
    pass
```

```
>>>
```

```
>>> x=Contact()
```

```
>>>
```

```
>>> x.name='Rob Miles'
```

```
>>> x.name  
'Rob Miles'  
>>> x.name = x.name + ' is a star'  
>>> x.name  
'Rob Miles is a star'  
>>>
```

## Tiny Contacts

1. New Contact
2. Find Contact
3. Edit Contact
4. Exit program

Edit contact

Enter the contact name:Rob

Name: Robert Miles

Enter new name or . to leave unchanged: .

Enter new address or . to leave unchanged: .

Enter new telephone or . to leave unchanged: +44 (1482) 465079

This name was not found.

```
c=find_contact('Mysterious X')  
print(c.address)
```

```
    print(c.address)
AttributeError: 'NoneType' object has no attribute 'address'
```

```
rob=find_contact('Rob Miles')
```

```
test.name='Robert Miles Man of Mystery'
```

```
contacts[0]=contacts[1]
```

Edit contact

Enter the contact name:Rob

Name: Robert Miles

Enter new name or . to leave unchanged: .

Enter new address or . to leave unchanged: .

Enter new telephone or . to leave unchanged: 123-456-7890

AttributeError: 'Contact' object has no attribute 'address'

```
output_file = open('contacts.pickle', 'wb')
```

```
def save_contacts(file_name):
    """
    Saves the contacts to the given file name
    Contacts are stored in binary as a pickled file
    Exceptions will be raised if the save fails
    """

    print('save contacts')
    with open(file_name, 'wb') as out_file:
        pickle.dump(contacts,out_file)
```

```
>>> class InitPrint:  
    def __init__(self):  
        print('you made an InitPrint instance')  
  
>>>
```

```
>>> x=InitPrint()
you made an InitPrint instance
>>>
```

```
>>> class InitName:  
        def __init__(self,new_name):  
            self.name=new_name
```

```
>>>
```

```
>>> x=InitName('fred')  
>>>
```

```
>>> y=InitName()
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    y=InitName()
TypeError: __init__() missing 1 required positional argument: 'name'
```

```
rob=Contact(name='Rob Miles',address='18 Pussycat Mews, London, NE1 410S',  
telephone='+44(1234) 56789')
```

```
# EG9-07 Tiny Contacts with initializer
def new_contact():
    """
    Reads in a new contact and stores it
    """

    print('Create new contact')
    # add the data attributes
    name=read_text('Enter the contact name: ')
    address=read_text('Enter the contact address: ')
    telephone=read_text('Enter the contact phone: ')
    # create a new instance
    new_contact=Contact(name=name,address=address,telephone=telephone)
    # add the new contact to the contact list
    contacts.append(new_contact)
```

```
class Contact:  
    def __init__(self, name, address, telephone='No telephone'):  
        self.name=name  
        self.address=address  
        self.telephone=telephone
```

```
rob=Contact(name='Rob Miles',address='18 Pussycat Mews, London, NE1 410S')
```

```
>>> prices['Latte']
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    prices['Latte']
KeyError: 'Latte'
```

```
>>> prices['espresso']=3.0
>>> prices['tea']=2.5
>>> prices
{'latte': 3.6, 'espresso': 3.0, 'tea': 2.5}
>>>
```

```
>>> prices={'latte': 3.6, 'espresso': 3.0, 'tea': 2.5, 'americano':2.5}
>>>
```

```
access_control={1234:'complete', 1111:'limited', 4342:'limited'}
```

```
# EG9-08 Pirate Treasure Dictionary
def get_treasure_location():
    """
    Get the location of the treasure
    returns a dictionary:
    ['start'] is a string naming the landmark to start
    ['n'] is the number of paces north
    ['e'] is the number of paces east
    """
    # get the location from the pirate
    return {'start':'The old oak tree','n':20,'e':30}

location=get_treasure_location()
print ('Start at',location['start'], 'walk',location['n'],'paces north',
      'and', location['e'],'paces east')
```

```
c = contact_dictionary['Rob Miles']
```

```
Enter your command: 4
add hours
Enter the contact name: Rob
Name: Rob Miles
Previous hours worked : 0
Session length : 3
Updated hours worked : 3.0
```

**Find contact**

Enter the contact name: Rob Miles

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours worked : 3.0

```
session_length = read_float_ranged(prompt='Session length : ', min_value=0.5,  
max_value=3.5)  
contact.hours_worked = contact.hours_worked+session_length
```

```
session_length = read_float_ranged(prompt='Session length : ', min_value=0.5,  
max_value=4.0)  
contact.hours_worked = contact.hours_worked+session_length
```

```
class Contact:  
    def __init__(self, name, address, telephone):  
        self.name = name  
        self.address = address  
        self.telephone = telephone  
        self.hours_worked = 0  
  
    def get_hours_worked(self):  
        """  
        Gets the hours worked for this contact  
        """  
        return self.hours_worked
```

```
rob_work = rob.get_hours_worked()
jim_work = jim.get_hours_worked()
if rob_work > jim_work:
    print('More work for rob')
else:
    print('More work for jim')
```

```
# EG10-02 Time Tracker with method attributes

class Contact:

    def add_session(self, session_length):
        ...
        Adds the value of the parameter
        onto the hours worked for this contact
        ...
        self.hours_worked = self.hours_worked + session_length
```

```
rob.add_session(-10)
```

```
class Contact:  
  
    min_session_length = 0.5  
    max_session_length = 3.5
```

```
print(Contact.validate_session_length(5))
```

```
def add_session(self, session_length):
    """
    Adds the value of the parameter
    onto the hours spent with this contact
    Returns True if it works,
    or False if the session length is invalid
    """
    if not Contact.validate_session_length(session_length):
        return False
    self.hours_worked = self.hours_worked + session_length
    return True
```

```
>>> x=int('rob')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    x=int('rob')
ValueError: invalid literal for int() with base 10: 'rob'
```

```
if not Contact.validate_session_length(session_length):
    raise Exception('Invalid session length')
```

## Time Tracker

1. New Contact
2. Find Contact
3. Edit Contact
4. Add Session
5. Exit Program

Enter your command: 1

Create new contact

Enter the contact name: Rob Miles

Enter the contact address: 18 Pussycat Mews, London, NE1 410S

Enter the contact phone: 1234 56789

```
Enter your command: 4
add session
Enter the contact name: Rob Miles
Name: Rob Miles
Previous hours worked: 0
Session length: 2
Updated hours worked: 2.0
```

```
Enter your command: 4
add session
Enter the contact name: Rob Miles
Name: Rob Miles
Previous hours worked: 2.0
Session length: 4
Traceback (most recent call last):
  File "C:/Users/Rob/EG10-06 Time Tracker with exception.py", line 197, in <module>
    add_session_to_contact()
  File "C:/Users/Rob/EG10-06 Time Tracker with exception.py", line 145, in add_
    session_to_contact
      if contact.add_session(session_length):
  File "C:/Users/Rob/EG10-06 Time Tracker with exception.py", line 45, in add_
    session
      raise Exception('Invalid session length')
Exception: Invalid session length
```

Enter the contact name: Rob Miles

Name: Rob Miles

Previous hours worked: 2.0

Session length : -1

Add failed: Invalid session length

```
def get_hours_worked(self):
    """
    Gets the hours spent with this contact
    """
    return self._hours_worked
```

```
>>> class Secret:  
        def __init__(self):  
            self._secret=99  
            self.__top_secret=100
```

```
>>>
```

```
>>> x.__top_secret
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    x.__top_secret
AttributeError: 'Secret' object has no attribute '__top_secret'
```

```
>>> x._Secret__top_secret
```

```
>>> x._Secret__top_secret  
100
```

```
>>> class Prop:  
    @property  
    def x(self):  
        print('property x get')  
        return self.__x  
    @x.setter  
    def x(self,x):  
        print('property x set:', x)  
        self.__x = x
```

```
>>>
```

```
def __init__(self, name, address, telephone):  
    self.name = name  
    self.address = address  
    self.telephone = telephone  
    self.__hours_worked = 0
```

```
rob = Contact(name='Rob', address='18 Pussycat Mews, London, NE1 410S',  
telephone='1234 56789')
```

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours on the case: 2.0

Billing amount: 130.0

```
@property  
def billing_amount(self):  
    return self.__billing_amount
```

```
print('Rob owes:', rob.billing_amount)
```

```
amount_to_bill = 30 + (50 * session_length)
```

```
self.__billing_amount = self.__billing_amount+amount_to_bill
```

```
class Contact:  
  
    __open_fee = 30  
    __hourly_fee = 50
```

```
amount_to_bill = Contact.__open_fee + (Contact.__hourly_fee * session_length)
```

```
def add_session(self, session_length):
    """
    Adds the value of the parameter
    onto the hours spent with this contact
    Raises an exception if the session length is invalid
    """
    if not Contact.validate_session_length(session_length):
        raise Exception('Invalid session length')
    self.__hours_worked = self.__hours_worked + session_length
    amount_to_bill = Contact.__open_fee + (Contact.__hourly_fee * session_length)
    self.__billing_amount = self.__billing_amount + amount_to_bill
    return
```

```
def display_contact():
    """
    Reads in a name to search for and then displays
    the contact information for that name or a
    message indicating that the name was not found
    """

    print('Find contact')
    search_name = read_text('Enter the contact name: ')
    contact = find_contact(search_name)
    if contact != None:
        # Found a contact
        print('Name:', contact.name)
        print('Address:', contact.address)
        print('Telephone:', contact.telephone)
        print('Hours on the case:', contact.hours_worked)
        print('Amount to bill:', contact.billing_amount)
    else:
        print('This name was not found.')
```

```
Traceback (most recent call last):
  File "C:/Users/Rob/EG10-11 Time Tracker with Billing Amount.py", line 257,
    in <module>
      display_contact()
  File "C:/Users/Rob/EG10-11 Time Tracker with Billing Amount.py", line 160,
    in display_contact
      print('Amount to bill:', contact.billing_amount)
  File "C:/Users/Rob/ EG10-11 Time Tracker with Billing Amount.py", line 79,
    in billing_amount
      return self._billing_amount
AttributeError: 'Contact' object has no attribute '_Contact__billing_amount'
```

```
def check_version(self):
    """
    Checks the version number of this instance of
    Contact and upgrades the object if required.
    """
    pass
```

Enter your command: 1

Create new contact

Enter the contact name: Rob Miles

Enter the contact address: 18 Pussycat Mews, London, NE1 410S

Enter the contact phone: 1234 56789

Enter your command: 2

Find contact

Enter the contact name: Rob

Version: 1

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours on the case: 0

Enter your command: 5  
save contacts

Enter your command: 2

Find contact

Enter the contact name: Rob

Version: 2

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours on the case: 0

Billing amount: 0

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours on the case: 3.0

Amount to bill: 180

```
>>> name = 'Rob Miles'  
>>> age = 21
```

```
>>> template = 'My name is {0} and my age is {1}'
```

```
>>> template.format(name,age)
```

'My name is Rob Miles and my age is 21'

```
template = 'My name is {0:20} and my age is {1:10}'  
'My name is Rob Miles           and my age is      21'
```

```
template = 'My name is {0:>20} and my age is {1:<10}'  
'My name is' Rob Miles and my age is 21'
```

```
template = 'My name is {0:20} and my age is {1:10.2f}'  
'My name is Rob Miles           and my age is      21.00'
```

## Time Tracker

1. New Contact
2. Find Contact
3. Edit Contact
4. Add Session
5. Exit Program

Enter your command: 2

Enter the contact name: Rob

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours on the case: 10.0

Amount to bill: 470.0

Sessions

Date: Mon Jul 10 11:30:00 2017 Length: 1.0

Date: Tue Jul 12 11:30:00 2017 Length: 2.0

Date: Wed Jul 19 11:30:00 2017 Length: 2.5

Date: Wed Jul 26 10:30:20 2017 Length: 2.5

Date: Mon Jul 31 16:51:45 2017 Length: 1.0

Date: Mon Aug 14 16:51:45 2017 Length: 1.0

```
class Session:

    __min_session_length = 0.5
    __max_session_length = 3.5

    @staticmethod
    def validate_session_length(session_length):
        """
        Validates a session length and returns
        True if the session is valid or False if not
        """
        if session_length < Session.__min_session_length:
            return False
        if session_length > Session.__max_session_length:
            return False
        return True

    def __init__(self, session_length):
        if not Session.validate_session_length():
            raise Exception('Invalid session length')
        self.__session_length = session_length
        self.__session_end_time = time.localtime()
        self.__version = 1
```

```
session_record = Session(session_length)
```

```
def check_version(self):  
    pass
```

```
@property
def session_length(self):
    return self._session_length

@property
def session_end_time(self):
    return self._session_end_time
```

```
>>> code = ['line1', 'line2', 'line3']
>>>
```

```
>>> code
['line1', 'line2', 'line3']
>>>
```

```
>>> def indent(x):  
    return '    '+x
```

```
>>>
```

```
>>> indented_code = map(indent, code)
>>>
```

```
>>> indented_code
<map object at 0x00000211E6FCBA58>
>>>
```

```
>>> for s in indented_code:  
    print(s)
```

```
>>> for s in indented_code:  
    print(s)
```

```
line1  
line2  
line3
```

```
>>> indented_code = map(indent, code)
```

```
>>> indented_code.__next__()
```

```
>>> indented_code.__next__()  
'line1'  
>>>
```

```
>>> indented_code.__next__()  
'line2'  
>>> indented_code.__next__()  
'line3'  
>>> indented_code.__next__()  
Traceback (most recent call last):  
  File "<pyshell#67>", line 1, in <module>  
    indented_code.__next__()  
StopIteration  
>>>
```

```
>>> indent_iterator = map(indent, code)
```

```
>>> indented_code = list(indent_iterator)
>>>
```

```
>>> indented_code  
['    line1', '    line2', '    line3']  
>>>
```

```
>>> i1 = map(indent, code)
>>> i2 = map(indent, i1)
```

```
>>> list(i2)
['line1', 'line2', 'line3']
```

```
report_strings = map(str, self.__sessions)
```

```
report_result = '\n'.join(report_strings)
```

```
>>> report_strings = ['report1', 'report2', 'report3', 'report4']  
>>>
```

```
>>> '***'.join(report_strings)
```

```
>>> '**'.join(report_strings)
'report1**report2**report3**report4'
>>>
```

```
>>> print('\n'.join(report_strings))
report1
report2
report3
report4
>>>
```

```
>>> ''.join(report_strings)
'report1report2report3report4'
>>>
```

```
# EG10-18 Twinkle Twinkle
import time
import snaps

snaps.play_note(0)
time.sleep(0.4)
snaps.play_note(0)

time.sleep(0.4)
snaps.play_note(7)
time.sleep(0.4)
snaps.play_note(7)
time.sleep(0.4)
snaps.play_note(9)
time.sleep(0.4)
snaps.play_note(9)
time.sleep(0.4)
snaps.play_note(7)
time.sleep(0.8)
```

```
def __str__(self):
    template = 'Note: {0} Duration: {1}'
    return template.format(self.__note, self.__duration)
```

```
tune_strings = map(str,tune)
print('\n'.join(tune_strings))
```

## Mary's Fashion Shop

- 1: Create new stock item
- 2: Add stock to existing item
- 3: Sell stock
- 4: Stock report
- 5: Exit

Enter your command:

Dress: stock reference: 'D0001' price: 100.0 color: red pattern: swirly size: 12  
Pants: stock reference: 'TR12327' price:50 color: black pattern: plain length: 30  
waist: 30



```
# EG11-01 Separate classes

class Dress:
    def __init__(self, stock_ref, price, color, pattern, size):
        self.stock_ref = stock_ref
        self.__price = price
        self.__stock_level = 0
        self.color = color
        self.pattern = pattern
        self.size = size

    @property
    def price(self):
        return self.__price

    @property
    def stock_level(self):
        return self.__stock_level

class Pants:
    def __init__(self, stock_ref, price, color, pattern, length, waist):
        self.stock_ref = stock_ref
        self.__price = price
        self.__stock_level = 0
        self.color = color
        self.pattern = pattern
        self.length = length
        self.waist = waist

    @property
    def price(self):
        return self.__price

    @property
    def stock_level(self):
        return self.__stock_level

x = Dress(stock_ref='D0001', price=100, color='red', pattern='swirly', size=12)
y = Pants(stock_ref='TR12327', price=50, color='black', pattern='plain', length=30,
waist=25)
print(x.price)
print(y.stock_level)
```

```
x = Dress(stock_ref='D0001', price=100, color='red', pattern='swirly', size=12)
```

```
super(Dress, self).__init__(stock_ref, price, color)
```

```
class StockItem(object):

    @property
    def item_name(self):
        return 'Stock Item'
```

```
class Dress(StockItem):

    @property
    def item_name(self):
        return 'Dress'
```

```
>>> print(o)
<object object at 0x0000020B57A59070>
```

```
<__main__.Contact object at 0x0000018E5E9EBB70>
```

```
>>> class StrTest(object):
        def __str__(self):
            return 'string from StrTest'
```

```
>>>
```

```
>>> t1 = StrTest()  
>>> print(t1)  
string from StrTest
```

```
>>> class StrTestSub(StrTest):
        def __str__(self):
            return super().__str__() + '..with sub'
```

```
>>>
```

```
>>> t2 = StrTestSub()  
>>> print(t2)  
string from StrTest..with sub
```

```
# EG11-04 Stock Items with str

x = Dress(stock_ref='D001', price=100, color='red', pattern='swirly', size=12)
print(x)
Stock Reference: D001
Price: 100
Stock level: 0
Color: red
Pattern: swirly
Size: 12
```

```
Traceback (most recent call last):
  File "C:/Users/Rob/FashionShop.py", line 198, in <module>
    new_item = get_new_item()
  File "C:/Users/Rob//FashionShop.py ", line 165, in get_new_item
    pattern=pattern, size=size)
  File "C:/Users/Rob//FashionShop.py ", line 42, in __init__
    super().__init__(price, color)
TypeError: __init__() missing 1 required positional argument: 'location'
```

```
result = getattr(self, '_location', None)
```

```
new_dress = Dress(price=100, color='red', pattern='swirly', size=12)
new_dress.location = 'Front of shop'
```

```
if hasattr(new_dress,'location'):
    print('The dress is located: ', new_dress.location)
else:
    print('The dress does not have location information')
```

```
RESTART: C:/Users/Rob/EG11-06 Instrumented Stock Items.py
Instrumented classes ready for use
>>>
```

```
>>> new_dress = Dress('D001', 100, 'red', 'swirly', 12, 'shop window')
```

```
>>> new_dress = Dress('D001',100, 'red', 'swirly', 12, 'shop window')
** Dress __init__ called
** StockItem __init__ called
>>>
```

```
>>> new_jeans = Jeans('J1',50, 'blue', 'plain', 30, 30, 'flared', 'shop window')
** Jeans __init__ called
** Pants __init__ called
** StockItem __init__ called
>>>
```

```
>>> print(new_jeans)
** Jeans __str__ called
** Pants __str__ called
** StockItem __str__ called
** Jeans get item_name called
** StockItem get price called
** StockItem get stock level called
Stock Reference: J1
Price: 50
Stock level: 0
Color: blue
Location: shop window
Pattern: plain
Length: 30
Waist: 30
Style: flared
>>>
```

```
>>> new_dress.check_version()
** Dress check_version called
** StockItem check_version called
>>>
```

```
>>> StockItem.show_instrumentation = False
>>> print(new_jeans)
Stock Reference: J1
Price: 50
Stock level: 0
Color: blue
Location: shop window
Pattern: plain
Length: 30
Waist: 30
Style: flared
>>>
```

## Mary's Fashion Shop

- 1: Create new stock item
- 2: Add stock to existing item
- 3: Sell stock
- 4: Stock report
- 5: Exit

Enter your command:

```
# EG11-09 Selling stock

class StockItem(object):
    """
    Stock item for the fashion shop
    """

    def sell_stock(self, count):
        if count < 1:
            raise Exception('Invalid number of items to sell')

        if count > self.__stock_level:
            raise Exception('Not enough stock to sell')

        self.__stock_level = self.__stock_level - count
```



```
# EG11-10 FashionShop template

class FashionShop:

    def __init__(self):
        pass

    def save(self, filename):
        """
        Saves the fashion shop item to the given file name
        Exceptions will be raised if the save fails
        """
        pass

    @staticmethod
    def load(filename):
        """
        Loads a fashion shop item from the given file name
        Exceptions will be raised if the load fails
        """
        return None

    def store_new_stock_item(self, item):
        """
        Create a new fashion shop item
        The item is indexed on the stock_ref attribute
        Raises an exception if the item is already
        stored in the fashion shop
        """
        pass

    def find_stock_item (self, stock_ref):
        """
        Gets an item from the stock
        Returns None if there is no item for
        this stock reference
        """
        return None

    def __str__(self):
        return ''
```

```
shop = FashionShop()  
shop.save('FashionShop.pickle')
```

```
loaded_shop = FashionShop.load('FashionShop.pickle')
```

```
dress = Dress(stock_ref='D001', price=100, color='red', pattern='swirly', size=12,  
location='front')  
shop = FashionShop()  
shop. store_new_stock_item(dress)
```

```
item = shop.find_stock_item('D001')
if item == None:
    print('Item not found')
else:
    print(item)
```

```
dress = Dress(stock_ref='D001', price=100, color='red', pattern='swirly', size=12,
location='front')
shop = FashionShop()
shop.store_new_stock_item(dress)
print(shop)
Items in Stock

Stock Reference: D001
Type: Dress
Location: front
Price: 100
Stock Level: 0
Color: red
Pattern: swirly
Size: 12
```

```
ui = FashionShopShellApplication('fashionshop.pickle')
```

```
ui = FashionShopShellApplication('dressshop1.pickle')
ui.main_menu()
```

```
set('hello world')
{'o', 'l', ' ', 'h', 'r', 'w', 'd', 'e'}
```

```
sorted('Rob Miles')
[' ', 'M', 'R', 'b', 'e', 'i', 'l', 'o', 's']
sorted(set('hello world'))
[' ', 'd', 'e', 'h', 'l', 'o', 'r', 'w']
```

```
pocket = {'axe', 'apple', 'herbs', 'flashlight'}
```

```
apple_potion = {'apple', 'herbs'}
if apple_potion.issubset(pocket):
    print('You have the ingredients for the apple potion')
```

```
pocket = pocket - apple_potion  
pocket.add('apple potion')
```

Enter tags (separated by commas): **outdoor,spring,informal,short**

```
tag_string = read_text('Enter tags (separated by commas): ')
```

```
tag_string = str.lower(tag_string)
```

```
tag_list = str.split(tag_string, sep=',')
```

Enter tags (separated by commas): outdoor, spring, informal, short

```
tag_list = map(str.strip,tag_list)
```

```
@staticmethod
def get_tag_set_from_text(tag_text):
    """
    Converts a comma-separated list into a set
    of individual items
    Converts the text to lowercase and trims each
    word
    """
    # Convert the string to lowercase
    tag_text = str.lower(tag_text)

    # Make a list of all the words in the string
    # separated by the comma character
    tag_list = str.split(tag_text, sep=',')

    # Remove any spaces at the start or
    # end of each string in the list
    tag_list = map(str.strip, tag_list)

    # return a set created from the list
    return set(tag_list)
```

Enter the tags to look for (separated by commas): outdoor,spring

Stock Reference: BL343

Type: Blouse

Location: blouse rail

Price: 100

Stock level: 0

Color: pink

Tags: {'spring', 'friendly', 'city', 'outdoor'}

Size: 14

Style: plain

Pattern: check

```
def match_tags(item):
    """
    Returns True if the tags in the item
    contain the search tags
    """
    return search_tags.issubset(item.tags)
```

```
filtered_list = filter(match_tags, stock_list)
```

```
def find_matching_with_tags(self, search_tags):
    """
    Returns the stock items that contain
    the search_tags as a subset of their tags
    """

    def match_tags(item):
        """
        Returns True if the tags in the item
        contain the search tags
        """
        return search_tags.issubset(item.tags)

    return filter(match_tags, self.__stock_dictionary.values())
```

```
What kind of item do you want to add: 1
Creating a Dress
Enter stock reference: D001
Enter price: 120
Enter color: red
Enter location: shop window
Enter tags (separated by commas): evening, long
Enter pattern: swirly
Enter size: 12
```

Enter stock reference: D001

Enter price: 120

Enter tags (separated by commas): dress,color:red,location:shop  
window,pattern:swirly,size:12,evening,long

hello from function 1  
hello from function 2

```
Traceback (most recent call last):
  File "C:/Users/Rob/EG12-02 Invalid Function References.py", line 7, in <module>
    x(99)
TypeError: func_1() takes 0 positional arguments but 1 was given
```

```
age = BTCInput.read_int('Enter your age: ')
```

```
def read_float(prompt):
    return read_number(prompt=prompt, number_converter=float)
```

```
age = read_number(prompt='Enter your age in roman numerals',  
number_converter=roman_converter)
```

Dance starting  
robot moving forward  
robot moving back  
robot moving left  
robot moving right  
Dance over

```
increment = lambda x : x+1
```

```
>>> numbers = [1,2,3,4,5,6,7,8]
```

```
def increment(x):  
    return x+1  
  
new_numbers = map(increment, numbers)
```

```
>>> new_numbers = map(lambda x : x+1, numbers)
```

```
>>> list(new_numbers)
[2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> adder = lambda x, y : x+y
```

```
hour = 8 # set the value of hour to the current hour of the time
print('morning' if hour < 12 else 'afternoon')
```

```
day_prompt = lambda hour:'morning' if hour < 12 else 'afternoon'
```

```
for i in range(1, 5):
    print(i)
```

```
>>> def mr_yield():
        yield 1
        yield 2
        yield 3
        yield 4
```

```
>>>
```

```
>>> for i in mr_yield():
    print(i)
```

```
>>> for i in mr_yield():
    print(i)
```

```
1
2
3
4
```

```
contacts = list(Contact.create_test_contacts())
```

```
def yield_return():
    yield 1
    yield 2
    return 3
    yield 4

for i in yield_return():
    print(i)
```

```
def forever_tens():
    result = 0
    while True:
        yield result
        result = result + 10
```

```
for result in forever_tens():
    print(result)
    if result > 100:
        break
```

```
print('Hello world')
print('The answer is',42)
```

```
>>> def add_function(x, y):  
    return x + y  
>>>
```

```
>>> add_function(1, 2, 3)
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    add_function(1, 2, 3)
TypeError: add_function() takes 2 positional arguments but 3 were given
```

```
>>> add_function(*numbers)
10
```

```
def add_function(*values):  
    return sum(values)
```

```
def readme():
    print('''Welcome to the BTCInput functions version 1.0
```

You can use these to read numbers and strings in your programs.

The functions are used as follows:

```
text = read_text(prompt)
int_value = read_int(prompt)
float_value = read_float(prompt)
int_value = read_int_ranged(prompt, max_value, min_value)
float_value = read_float_ranged(prompt, max_value, min_value)
```

Have fun with them.

```
Rob Miles'''
```

```
# all the BTCInput functions go here  
  
# Have the BTCInput module introduce itself once the  
# functions have been defined  
readme()
```

```
if __name__ == '__main__':
    # Have the BTCInput module introduce itself
    readme()
```

`FashionShopShellUI.py`

`ShellUI`

`BTCInput.py`

`FashionShopShell.py`

`__init__.py`

`Storage`

`FashionShop.py`

`StockItem.py`

`__init__.py`

```
from Storage import FashionShop
```

```
shop = FashionShop.FashionShop
```

```
shop = DiskStorage.FashionShop
```

```
>>> class VarTest:  
    def __init__(self):  
        print('making a VarTest')
```

```
>>>
```

```
shop = FashionShop.FashionShop
```

```
# Loads a user interface class and a data manager class
# and then uses these to create an application

# Get the module containing the user interface class
# from the ShellUI package
from ShellUI import FashionShopShell

# Get the user interface manager class from this module
ui = FashionShopShell.FashionShopShell

# Get the module containing the data storage class
# from the Storage package
from Storage import FashionShop
# Get the data manager class from the storage module
shop = FashionShop.FashionShop
```

```
# Now call the main_menu function on the app  
app.main_menu()
```

```
Traceback (most recent call last):
  File "C:/Users/Rob/Test Program.py", line 20, in <module>
    assert item.stock_level == 0
AssertionError
```

.

---

Ran 1 test in 0.037s

OK

```
..  
test_StockItem_init (__main__.TestShop) ... ok
```

```
-----  
Ran 1 test in 0.013s
```

```
OK
```

```
def test_that_fails(self):  
    self.assertEqual(1, 0)
```

```
.F
=====
FAIL: test_that_fails (__main__.TestShop)
-----
Traceback (most recent call last):
  File "C:/Users/Rob/OneDrive/Begin to code Python/ tinytest.py",
    line 16, in test_that_fails
      self.assertEqual(1, 0)
AssertionError: 1 != 0

-----
Ran 2 tests in 0.008s

FAILED (failures=1)
```

```
with self.assertRaises(Exception):
    item.add_stock(-1)
```

F.

```
=====
FAIL: test_StockItem_add_stock (__main__.TestShop)
-----
Traceback (most recent call last):
  File "C:\Users\Rob\OneDrive\Begin to code Python\Part 2 Final\Ch 12
    Python Libraries\code\samples\EG12-09 TestFashionShopApp\tinytest.py",
  line 25, in test_StockItem_add_stock
    item.add_stock(1)
AssertionError: Exception not raised

-----
Ran 2 tests in 0.049s

FAILED (failures=1)
```

PS C:\Users\Rob\Desktop\EG12-10 TestFashionShopApp Doc> python -m pydoc

`pydoc` – the Python documentation tool

`pydoc <name> ...`

Show text documentation on something. `<name>` may be the name of a Python keyword, topic, function, module, or package, or a dotted reference to a class or function within a module or module in a package. If `<name>` contains a '\', it is used as the path to a Python source file to document. If name is 'keywords', 'topics', or 'modules', a listing of these things is displayed.

`pydoc -k <keyword>`

Search for a keyword in the synopsis lines of all available modules.

`pydoc -p <port>`

Start an HTTP server on the given port on the local machine. Port number 0 can be used to get an arbitrary unused port.

`pydoc -b`

Start an HTTP server on an arbitrary unused port and open a Web browser to interactively browse documentation. The -p option can be used with the -b option to explicitly specify the server port.

`pydoc -w <name> ...`

Write out the HTML documentation for a module to a file in the current directory. If `<name>` contains a '\', it is treated as a filename; if it names a directory, documentation is written for all the contents.

PS C:\Users\Rob\Desktop\EG12-10 TestFashionShopApp Doc> python -m pydoc -b

```
PS C:\Users\Rob\Desktop\EG12-10 TestFashionShopApp Doc> python -m pydoc -b  
Server ready at http://localhost:57591/  
Server commands: [b]rowser, [q]uit  
server> q  
Server stopped
```

```
"""
Starts a Fashion Shop running with a Python Command Shell user interface

Runs only if it is started as the main program
"""

if __name__ == '__main__':
    # Loads a user interface class and a storage manager class
    # and then uses these to create an application
```

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

Python:Select Workspace Interpreter

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

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

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

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

```
>>>
```

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

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

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

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

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

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

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

```
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
        """
```

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

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

```
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)
```

```
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)
```

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

```
from tkinter import messagebox
```

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



```
"""

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()
```

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

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

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

```
>>>
```

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

>>> 283 277

290 297

290 306

289 307

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

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

```
>>>
```

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

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

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

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

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

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



```
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
```

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

```
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')
```

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

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

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

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

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

```
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
```

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

```
>>> listen_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
>>> listen_address = ('localhost', 10001)
```

```
>>> listen_socket.bind(listen_address)
```

```
>>> result = listen_socket.recvfrom(4096)
```

```
>>> send_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
>>> listen_address = ('localhost', 10001)
```

```
>>> send_socket.sendto(b'hello from me', send_address)
```

```
>>> send_socket.sendto(b'hello from me', send_address)
13
>>>
```

```
>>> print(result)
(b'hello from me', ('127.0.0.1', 51883))
```

The IP address of this computer is: 192.168.1.55

Listening:

Water Meter Day

We had a new water meter installed yay!

Python now in Visual Studio 2017

Python is now available in Visual Studio 2017 yay!

>>>

RESTART: C:/Users/Rob/EG14-03 Tiny socket web server.py  
Open your browser and connect to: http://localhost:8080

```
>>>
RESTART: C:/Users/Rob/EG15-01 Tiny socket web server.py
Got connection from: ('192.168.1.56', 51221)
GET / HTTP/1.1
Host: 192.168.1.56:8080
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/60.0.3112.113 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
>>>
```

```
request_string = network_message.decode()
```

```
connection, address = listen_socket.accept()
```

```
self.wfile.write(message_bytes)
```

```
my_server = http.server.HTTPServer(host_address, WebServerHandler)
```

```
>>> 'Robert'[0:3]  
'Rob'
```

```
<html>
<body>
<p> This is the index page for our tiny site.</p>
<a href="page.html">This is another page</a>
</body>
</html>
```

```
<html>
<body>
<p>This is another page in our tiny website.</p>
<a href="index.html">This takes us back to the index</a> </body>
</html>
```

```
<form method="post">
  <textarea name="message"></textarea>
  <button id="save" type="submit">Save Message</button>
</form>
```

```
length = int(self.headers['Content-Length'])
```

```
post_body_bytes = self.rfile.read(length)
```

```
post_body_text = post_body_bytes.decode()
```



```
message = query_strings['message'][0]
```

```
>>> surface = pygame.display.set_mode(size)
```

```
>>> pygame.display.set_caption('An awesome game by Rob')
```

```
>>> start = (0,0)
>>> end = (500, 300)
```

```
>>> pygame.draw.line(surface, red, start, end)
```

```
>>> pygame.draw.line(surface, red, start, end)
<rect(0, 0, 501, 301)>
```

```
>>> pygame.display.flip()
```

```
>>> white = (255, 255, 255)
>>> surface.fill(white)
>>> pygame.display.flip()
```

```
cheeseImage = pygame.image.load('cheese.png')
```

```
surface.blit(cheeseImage, cheesePos)
```

```
>>> import pygame
>>> pygame.init()
(6, 0)
>>> size = (800, 600)
>>> surface = pygame.display.set_mode(size)
```

```
>>> for e in pygame.event.get():
    print(e)
```

```
>>> for e in pygame.event.get():
    print(e)

<Event(17-VideoExpose {})>
<Event(16-VideoResize {'size': (800, 600), 'w': 800, 'h': 600})>
<Event(1-ActiveEvent {'gain': 0, 'state': 1})>
<Event(2-KeyDown {'unicode': 'r', 'key': 114, 'mod': 0, 'scancode': 19})>
<Event(3-KeyUp {'key': 114, 'mod': 0, 'scancode': 19})>
<Event(2-KeyDown {'unicode': 'o', 'key': 111, 'mod': 0, 'scancode': 24})>
<Event(3-KeyUp {'key': 111, 'mod': 0, 'scancode': 24})>
<Event(2-KeyDown {'unicode': 'b', 'key': 98, 'mod': 0, 'scancode': 48})>
<Event(3-KeyUp {'key': 98, 'mod': 0, 'scancode': 48})>
<Event(1-ActiveEvent {'gain': 1, 'state': 1})>
>>>
```

```
def draw(self):
    ...
    Draws the sprite on the screen at its
    current position
    ...
    self.game.surface.blit(self.image, self.position)
```

```
self.background_sprite = Sprite(image=background_image, game=self)
```

```
if self.position[0] + self.image.get_width() > self.game.width:  
    self.position[0] = self.game.width - self.image.get_width()
```

```
self.position[0] = (self.game.width - self.image.get_width())/2  
self.position[1] = (self.game.height - self.image.get_height())/2
```

```
class Cracker(Sprite):
    """
    The cracker provides a target for the cheese
    When reset, it moves to a new random place
    on the screen
    """
    def reset(self):
        self.position[0] = random.randint(0,
                                         self.game.width-self.image.get_width())
        self.position[1] = random.randint(0,
                                         self.game.height-self.image.get_height())
```

```
cracker_image = pygame.image.load('cracker.png')
self.cracker1 = Cracker(image=cracker_image, game=self)
self.cracker2 = Cracker(image=cracker_image, game=self)
self.cracker3 = Cracker(image=cracker_image, game=self)
```

```
for sprite in self.sprites:  
    sprite.update()
```

```
for sprite in self.sprites:  
    sprite.draw()
```

```
cracker_eat_sound = pygame.mixer.Sound('burp.wav')
```

```
def reset(self):
    self.entry_count = 0
    self.friction_value = 0.99
    self.x_accel = 0.2
    self.y_accel = 0.2
    self.x_speed = 0
    self.y_speed = 0
    self.position = [-100,-100]
```

```
self.font = pygame.font.Font(None, 60)
```

```
text = self.font.render('hello world', True, (255,0,0))
```

```
self.surface.blit(text, (0,0))
```

```
text_position[0] = text_position[0]+2
```

```
def draw_start(self):
    self.start_background_sprite.draw()
    self.display_message(message='Top Score: ' + str(self.top_score), y_pos=0)
    self.display_message(message='Welcome to Cracker Chase', y_pos=150)
    self.display_message(message='Steer the cheese to', y_pos=250)
    self.display_message(message='capture the crackers', y_pos=300)
    self.display_message(message='BEWARE THE KILLER TOMATOES', y_pos=350)
    self.display_message(message='Arrow keys to move', y_pos=450)
    self.display_message(message='Press G to play', y_pos=500)
    self.display_message(message='Press Escape to exit', y_pos=550)
```

```
def end_game(self):
    self.game_running = False
    if self.score > self.top_score:
        self.top_score = self.score
```

```
def update(self):
    ' position update code for the tomato here'
    if self.intersects_with(game.cheese_sprite):
        self.game.end_game()
```