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

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.

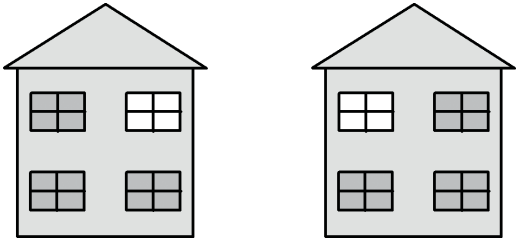


Figure 14-1 House-to-house networking

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.

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**:

```
>>> 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**:

```
>>> 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**.

```
>>> 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**.

```
>>> 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**:

```
>>> 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**:

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

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

```
>>> 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:

```
>>> 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.

```
>>> 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:

```
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 the socket module
import time                                            We will use the sleep function
                                                       from the time module
# You will need to change this to the address
# of the machine to which you are sending
target_ip = '192.168.1.55'                             Set the IP address of the
                                                       machine to which we are sending
send_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)   Create the socket
destination_address = (target_ip, 10001)               Set the destination address

while(True):                                           Loop forever
    print('Sending:')                                  Display a message
    send_socket.sendto(b'hello from me', destination_address)   Send a message
    time.sleep(2)                                      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.

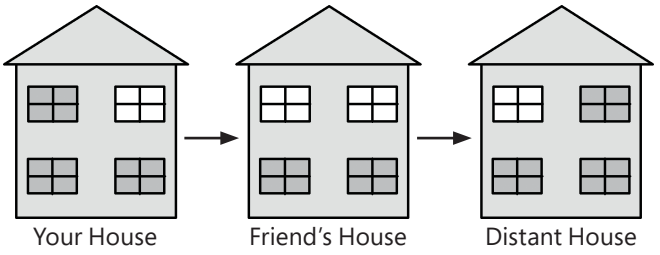


Figure 14-3 Routing from house to house to house

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.

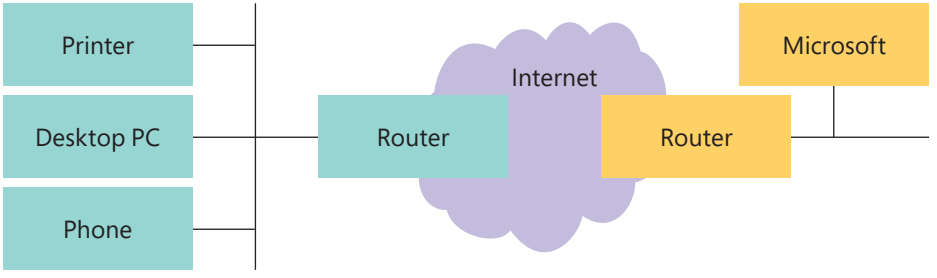


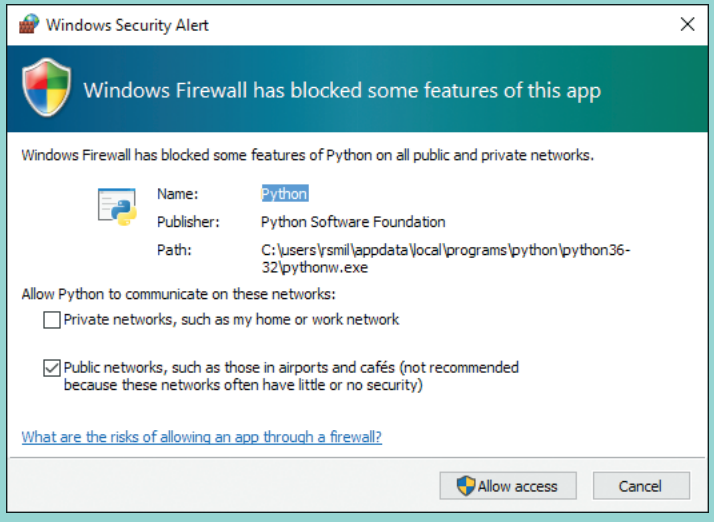
Figure 14-4 Routing and the Internet

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.

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 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)
```

Import the URL reader module

This is the URL from which the program will read

Create the web request object

Work through the web request a line at a time

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 snaps 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">                                          RSS element in the document
  <channel>                                                  Channel element in the RSS element
    <title>                                                  Element giving title of the channel
      robmiles.com                                           Title text
    </title>                                                 End of title element
    <item>                                                   Item in the blog
      <title>                                                Title of the item
        Water Meter Day
      </title>
      <category>Life</category>                              Category of the blog post
      <description>                                          Blog post content
        <![CDATA[ We had a new water meter installed yay!]]>
      </description>
    </item>                                                  End of item
    <item>                                                   Start of item
      <title>
        Python now in Visual Studio 2017
      </title>
      <category>Python</category>                            Category of the blog post
      <category>Visual Studio</category>                     Category of the blog post
      <description>
        <![CDATA[ Python is now available in Visual Studio 2017 yay! ]]>
      </description>
    </item>                                                  End of item
  </channel>                                                 End of channel
</rss>                                                       End of RSS feed
```

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

rss_text = '''
<rss version="2.0">
Sample RSS above goes here
</rss>
'''

doc = ElementTree.fromstring(rss_text)

for item in doc.iter('item'):
    title = item.find('title').text
    print(title.strip())
    description = item.find('description').text
    print('    ',description.strip())
```

Import the module and give it a name

The sample program holds all the text of the XML example

Create an `ElementTree` instance from the RSS string

Iterate through all the item elements in the document

Find the title element in the item and get the text out of it

Strip the title text of extra spaces and print it

Find the description element in the item and get the text from it

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:

```
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

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.