

These have to match each other so that the video encoder picks up the images that have been created. You can change test to anything you want, provided you change it in both places. If you want to take more than 999 images, you'll need to add more zeros. For example, if you wanted to take up to 99,999 images, you could change the threes in both lines to fives (the variable frames stores the number of frames captured (actually it will capture one less than this number, but we left it like this for simplicity).

-s 200x200 is the final dimension of the video in pixels. This doesn't have to match the size of the images. You can change out.mpg to whatever you wish to call the final video.

If you run this and create a video, you can then view the result with any standard video player such as VLC.

Creating Live Streams

The previous method works great if you want to create a video over a predetermined amount of time, but what if you want to keep a video going indefinitely? For example, what if you wanted to create a live video stream that you can watch over the Internet? Internet lore has it that the very first webcam was set up in Cambridge (the home of the Raspberry Pi) to allow people to keep an eye on a coffee pot, so they could time their visits to the canteen with a fresh brew. The following example re-creates that webcam.

In order to send data over the Internet, you'll need some form of server. There are a few that can do the job, but for the sake of familiarity, we'll use the Tornado server that you saw in Chapter 7.

The live stream can then be created with the following code. Before running it, you'll have to create the directory /home/pi/images with mkdir /home/pi/images.

```
import tornado.ioloop import tornado.web import subprocess import time

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        subprocess.call(["raspistill", "-w", "200", "-h", "200",
            "-e", "jpg", "-n", "-t", "1", "-o", "/home/pi/images/live.jpg"])
        time.sleep(2)
        self.write('<!DOCTYPE html><head>' +
            '<META HTTP-EQUIV="refresh" +
            'CONTENT="5"></head><body>' +
```

```
'</body>') class
ImageHandler(tornado.web.StaticFileHandler):
def set_extra_headers(self, path):
self.set_header('Cache-Control',
'no-store, no-cache, must-revalidate,' + ' max-age=0')

application = tornado.web.Application([
(r"/", MainHandler),
(r"/images/(.*)", ImageHandler, {"path":"/home/pi/images"})])

if __name__ == "__main__":
application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

If this doesn't look familiar to you, pop back to Chapter 7 and have a look at the section on Tornado. There are, however, a few new things in here.

In essence, all this does is wait for a user to request the page with the live stream (this is simply the root, but you can update it to be wherever you want), then it takes a new picture. The web page is then served, and it includes the picture. To make it a live stream rather than just a live image, the web page it serves includes the tag `<META HTTP-EQUIV="refresh" CONTENT="5">`, which tells the web browser to automatically refresh the page every five seconds. It's not high-quality video, but it's enough to keep you updated on the amount of coffee in a pot.

The class `ImageHandler` extends `tornado.web.StaticFileHandler`. It does this so that it can change the default settings on caching. By default, Tornado will try to be efficient and not reserve images it's served before. That means if a web browser requests an image that Tornado has already sent to it, it'll just tell it to use the same image. Since the browser is constantly checking the image `live.jpg`, it would normally keep reserving the first image. By adding this header, you're telling it not to be so lazy and to actually reread the file each time.

This works, but it's not ideal. For example, it takes a new picture every time someone requests a page, but if a lot of people are constantly requesting pages, it'll get clogged up, as the camera can only take one picture at a time. Secondly, it doesn't store any of the pictures. This isn't a problem if you're just watching coffee, but if you're using this as a security camera for example, this could be an

issue.

To get around this, you need to separate the section that takes the images, and the one that serves them. The following code will constantly take images so they can be amalgamated into a video, and at the same time, it keeps live.jpg updated with the latest. It's on the website as chapter9-live-take-pics.py:

```
import subprocess import time
file_number = 0
dir = "/home/pi/images/"

while True:
    file_name = dir + format(file_number, "05d") + ".jpg"
    file_number = file_number + 1
    subprocess.call(["raspistill", "-w", "400", "-h", "400", "-e",
                    "jpg", "-n", "-t", "1", "-o", file_name])
    subprocess.call(["cp", "-f", file_name, dir + "live.jpg"])
    time.sleep(2)
```

The next piece of code starts the previous script automatically, and displays the latest image just as the previous example did. It's on the website as chapter9-live-image-server.py.

```
import tornado.ioloop import tornado.web import subprocess

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write('<!DOCTYPE html><head>' +
        '<META HTTP-EQUIV="refresh" + ' CONTENT="5"></head><body>' +
        '</body>')

class ImageHandler(tornado.web.StaticFileHandler):
    def set_extra_headers(self, path):
        self.set_header('Cache-Control',
        'no-store, no-cache, must-revalidate, max-age=0')

application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/images/(.*)", ImageHandler, {"path":"/home/pi/images"})])
```

```
if __name__ == "__main__":
    subprocess.Popen(["python3", "chapter 9-live-take-pics.py"])
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

This uses `subprocess.Popen()` to create a new process that lets both of the programs run at the same time (more on this next chapter).

Taking Things Further

There are still a few problems though. This code will simply run constantly until it fills up the SD card with images. At this point, the Pi will probably crash. If you want to take things further, there are a few things you could do. For example, every certain amount of time (maybe every day or every 20,000 images), you could convert the images into a video, and then delete the images. For extra bonus points you could make the archive videos available online.

To make it more secure, you could constantly upload the videos to a central server that's offsite. This means that if your Pi is stolen or destroyed, you still have access to the surveillance footage.

If you are deploying this on a publicly accessible web server, you should implement some security so only authorised people are able to view the footage online. Take a look back to Chapter 7 for details on how to do this.

Summary

After reading this chapter, you should understand the following a bit better:

- PyAudio is a great module for bringing sound into Python programs. ■
Thewave module lets you output WAVE files.
- There aren't great Python tools for everything in audio, but that doesn't cause a problem because you can use the Linux command-line tools such as sox and flac to fill in the gaps.
- You can also use web services such as Google's speech recognition to add even more audio features.
- There are two options for adding cameras to your Pi: USB webcams and the Raspberry Pi Foundation's camera module.

- USB webcams work best with standard Python tools such as PyGame and OpenCV.
- The camera module works with OpenCV, but it also has its own set of command-line tools for grabbing images and videos.
- OpenCV can be used to add computer vision features such as object recognition.
- A web server such as Tornado can be used to serve streams of images on the Internet.

Chapter 10 Scripting

THERE ARE LOADS of different tasks that you need to perform on a computer to keep it running properly. For example, you need to back up your data regularly just in case there's a problem. Other tasks can depend on exactly what you use your computer for. You might need something to keep your music collection in order, or sort photos. This chapter looks at how you can use Python to make your life easier by automating these housekeeping jobs so that your computer keeps running with minimal intervention.

This requires a lot of interaction with the underlying operating system, so the first thing to do is learn a little about Linux.

Getting Started with the Linux Command Line

By this point, you're probably fairly familiar with general use of Raspbian. Raspbian brings together several distinct parts to create an operating system. Firstly, there's the Linux kernel. This is the bit that makes everything work at the lowest level. It's constantly running and manages the hardware and memory and controls how other programs run. Then there's the command line and a wide range of tools. These provide an entirely text-based way of using Raspbian, and it's an area that you'll be looking at in more detail in this chapter. The chances are, you probably don't use this too much at the moment; instead, you probably interact with the desktop environment, which is another distinct section of Raspbian. The default GUI is LXDE, and on top of this there are a wide range of graphical programs.

If you've mostly used Windows, the first big difference you'll notice will probably be the filesystem. You won't find a C drive (or for that matter, any other lettered drive) on your Raspberry Pi. Instead, everything is built into a

single hierarchy that starts at / (known as the root). Before getting started with Python scripting, you'll need to know where everything is in this filesystem, so open LXTerminal (the window that contains the text-based way of using Raspbian) and enter the following:

```
cd / ls
```

The first line changes the directory to / (the root), then the second lists everything in the current directory. You should get output like this (see Figure 10-1):

```
bin dev home lost+found mnt proc run selinux sys usr boot etc lib media opt root  
sbin srv tmp var
```

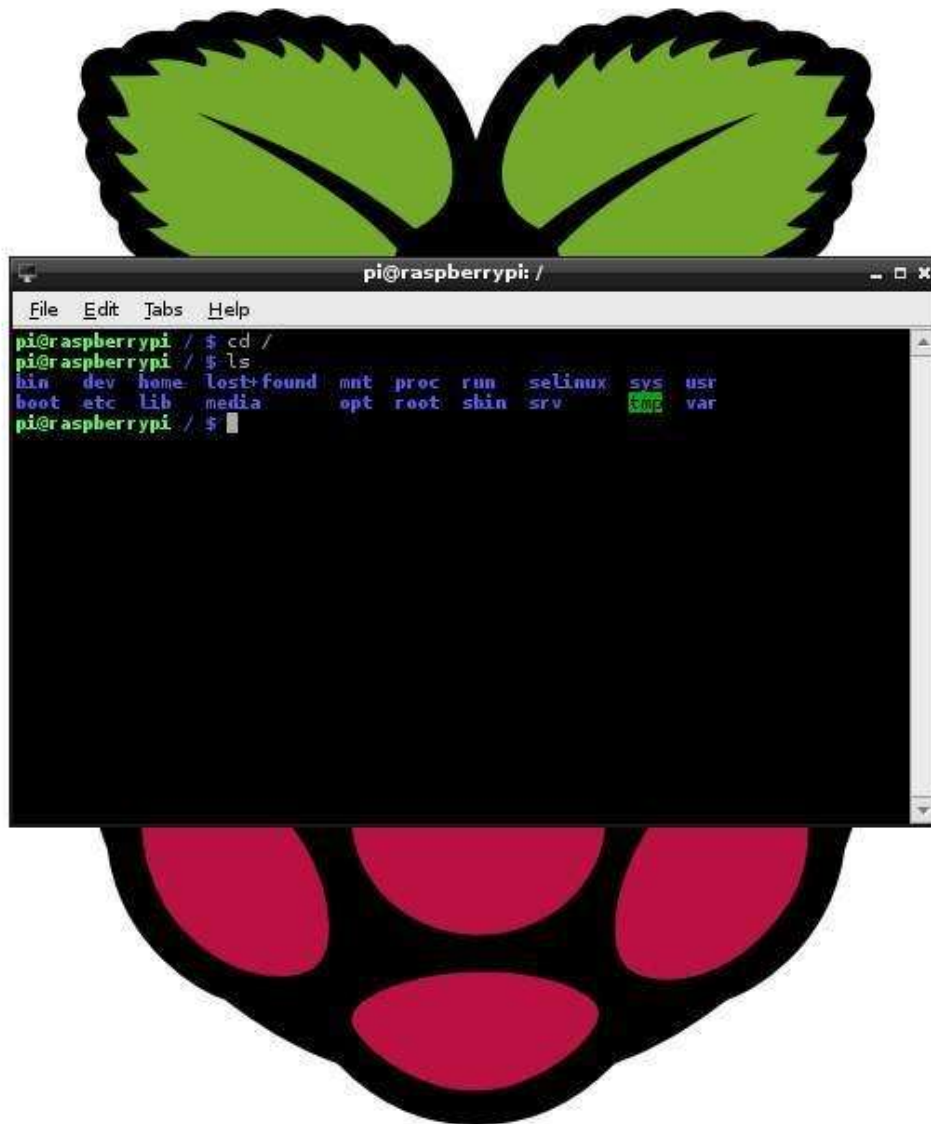


Figure 10-1: LXTerminal allows you to interact with the powerful Linux command-line environment.

These are the directories in the root (if there were any files, they'd be listed here as well, but there aren't any in this folder). If you use a different Linux system, you'll find a similar set of directories in /. Most of these directories you'll never need to touch. Raspbian will look after them for you, and keep everything up to date. The ones that you're likely to use are /home, where the users' home directories are kept, /media, where removable devices such as USB memory sticks will be found, and /etc, where system-wide application settings are kept.

It's important to realise that this filesystem doesn't directly correspond to a filesystem that's stored on a disk in the same way it does in Windows. So, for example, /home/pi is stored on the SD card. However, if you were to put in a USB memory stick called MyStick, you'll find it at /media/MyStick.

/sys and /proc don't exist on any disk. They're virtual filesystems that are created to look like normal directories and files, but are just the system's way for displaying information. For example, if you type:

```
cat /proc/cpuinfo
```

cat is a command that outputs the contents of one or more files, and /proc/cpuinfo contains the technical details of the CPU. You should get something like this:

```
pi@raspberrypi /proc $ cat /proc/cpuinfo
```

```
Processor: ARMv6-compatible processor rev 7 (v6l) BogoMIPS: 697.95  
Features: swp half thumb fastmult vfp edsp java tls CPU implementer: 0x41  
CPU architecture: 7  
CPU variant: 0x0  
CPU part: 0xb76  
CPU revision: 7
```

```
Hardware: BCM2708
```

```
Revision: 000d
```

```
Serial: 0000000074f3d523
```

The environment in LXTerminal is known as Bash. It's a powerful environment and even has its own programming language. If you want to learn more about Raspbian and Linux in general, Bash is a good place to start, and there are loads

of resources (such as www.linuxcommand.org). However, this is a book about Python, so we'll leave it here and let interested readers learn on their own.

Using the Subprocess Module

The easiest way to interact with the underlying system from Python is to use the subprocess module. We used this in Chapter 7, so you may already be a little familiar with it. Open up a Python interpreter (if you're already using LXTerminal, you can do this just by typing `python3`), and enter the following:

```
>>> import subprocess >>> subprocess.call("ls")
```

As you can see, using `subprocess.call()`, you can run any command on the underlying OS. If there are spaces in the command you want to run, you need to separate the command into a list of strings. For example, the command `cat /proc/cpuinfo` becomes:

```
>>> subprocess.call(["cat", "/proc/cpuinfo"])
```

This is all well and good, but all this really does is provide a more verbose way of running commands. After all, you could just have easily run the same commands in LXTerminal. As we said at the start, the aim of this chapter is to automate general tasks, and to do that you're going to need to read in the outputs so that you can manipulate them.

For example, `cat /proc/cpuinfo` returns a load of information, most of which you probably don't want to know. The following program strips out all the information except the line that tells you the type of processor the computer is running.

```
import subprocess
p = subprocess.Popen(["cat", "/proc/cpuinfo"], stdout=subprocess.PIPE)
text = p.stdout.read().decode()
for line in text.splitlines(): if line[:9] == "Processor": print(line)
```

This uses `subprocess.Popen()` rather than `subprocess.call()`. Doing this gives you much more control over what's going on because it creates a new object that you can use to get the information you want.

Whenever a command runs on a Linux machine, there are two pieces of output: `stdout` and `stderr`. `Stdout` (or standard out) is where all the normal output goes.

For example, when you run a Python program, any `print()` statements go to `stdout`. `Stderr` (or standard error) is where the system sends error messages. If you're running something in `LXTerminal`, both of these go to the screen, but when you're scripting things, it can be useful to split them up. That way, if you're running a lot of commands, you can send any error messages to one place so that you can see instantly if something's gone wrong without having to check through all the output.

The parameter `stdout=subprocess.PIPE` tells Python to keep `stdout` in the object we're creating rather than sending it to the screen. Since you're not telling it what to do with `stderr`, it will, by default, send that to the screen. So, if you change the line to:

```
p = subprocess.Popen(["cat", "/proc/cpuinfozzz"], stdout=subprocess.PIPE)
```

The error message will be printed on the screen even though it doesn't get printed in any `print()` statement.

If needed, you can also capture the `stderr` of a command. For example, take a look at the following program, which displays the contents of a file that the user enters:

```
import subprocess
f_name = input("Enter a filename: ")
p = subprocess.Popen(["cat", f_name ], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
text = p.stdout.read().decode() error = p.stderr.read().decode()
print(text)
```

```
if len(error) > 0:
print("*****ERROR*****") print(error)
```

Command-Line Flags

Linux commands often take flags. These flags come after the main command and are ways to tell it what to do. For example, in `LXTerminal`, try running the following:

```
ls
ls -a
ls --all
```

`ls` lists the contents of the current directory. By using the flag `-a`, you're telling it

to list everything in the directory (usually `ls` omits files and directories that start with a `.'`). `--all` is the same as `ls -a`. Many commands have two versions of each flag—a short version that starts with `-` and a long version (often easier to remember) that starts with `--`. To get more information on how to use `ls`, run it with either the flag `-h` or `--help`.

Flags can also take values, as you'll see in a minute.

If you're developing scripts, you should try to follow these conventions whenever possible. There's a module named `optparse` that can help. The previous example can be made to take its input from a flag rather than a prompted user input. Take a look at the following:

```
import subprocess
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f", "-file", dest="filename", help="The file to display")
options, arguments = parser.parse_args()

if options.filename:
    p = subprocess.Popen(["cat", options.filename ], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

text = p.stdout.read().decode() error = p.stderr.read().decode() else:
test = ""
error = "Filename not given" if len(error) > 0:
print("*****ERROR*****") print(error)
else:
print(text)
```

As you can see, this creates an `OptionParser` object. In this example, there's only one option, but you can add as many as you like by having more calls to `parser.add_option()`. The first two parameters to this are the short and long versions of the flag. `dest="filename"` means that the value of the flag is stored in the attribute called `filename` of the options that are returned.

Note that the parser automatically creates the flags `-h` and `--help`, and builds the help text up from the `help=` parameters in `add_options()` calls.

If you save the previous code as `print-file.py`, you can run it in `LXTerminal` by

cding to the directory it's saved in and entering `python3 print-file.py -f /proc/cpuinfo`. You can view the help with `python3 print-file.py --help`.

Regular Expressions

All this is, though, is a Python wrapper around `cat` that just does what the original does. It doesn't actually add anything.

Let's add a feature that lets the users specify which lines of the file they want to display.

Python (and many other programming languages) has a feature called regular expressions. This slightly oddly named feature (often shortened to `regex`) enables you to specify bits of text to match.

They do this with special characters. The most common special character is `*`. This means, match the preceding character zero or more times. For example: `do*g` will match `dg`, `dog`, `doog`, and so on. The character `+` will match the preceding character one or more times. For example, `do+g` will match `dog`, `doog`, `dooog`, and so on, but not `dg`. `do?g`, on the other hand will match `dg` or `dog` only.

A period will match any character other than a new line, so `.*` will match any line, while `.+` will match any line that isn't empty.

You can group characters together, so `d[io]g` will match `dig` and `dog`, and `d[io]*g` will match `dg`, `dog`, `dig`, `doog`, `dioioioig`, and anything like that.

We'll look at a few more features of regular expressions a bit later on, but for now let's get started with using them. The following code is on the website as `chapter10-regex.py`.

```
import subprocess
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-f", "-file", dest="filename", help="The file to display")

parser.add_option("-r", "-regex", dest="regex",
help="The regular expression to search for")
options, arguments = parser.parse_args()
if options.filename:

p = subprocess.Popen(["cat", options.filename ], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
```

```

text = p.stdout.read().decode() error = p.stderr.read().decode() else:
test = ""
error = "Filename not given"

if len(error) > 0:
print("*****ERROR*****")
print(error)

else:
for line in text.splitlines():
if not options.regex or (options.regex and re.search(options.regex, line)):
print(line)

```

This gets the regular expressions from the module `re`. There are two main ways of using regular expressions: `re.match()` and `re.search()`. The first one tries to match the regular expression from the start of the string, while the second one tries to find text anywhere in the string that matches the regular expression. This program does the latter because we wanted to make it as easy as possible to match lines.

There is a slightly convoluted condition in the `if` line:

```

if not options.regex or (options.regex and re.search(options.regex, line)):

```

This is to handle the case whereby the user hasn't entered a `-r` or `-regex` flag. It says to print the line if either there isn't a regex flag, or there is a flag and the line matches.

To match the first example, run it with:

```
python3 chapter.py -f /proc/cpuinfo -r Processor
```

Linux systems keep log files that register various events. These can be useful in diagnosing problems, but they can also get huge and be hard to work with.

They're all located in the `/var/log` folder. `syslog` holds much of the general information about what's been going on. For example, if you're having some difficulty with a USB device, running the following log file list all the times Raspbian registered a new USB peripheral (either hub or device):

```
python3 chapter10-regex.py -f /var/log/syslog -r"USB.*found"
```

In this case, putting the regular expression in quote marks isn't necessary, but sometimes the Linux shell will try to process special characters before they're passed to Python. Using double quote marks stops that from happening, so it's a

good habit to get into when putting regular expressions on the command line.

Let's return to features of regular expressions:

- `[^abc]` matches every character except a, b, and c, so `d[^o]g` matches dig, dxg, and dag but not dog.

- `[a-c]` matches the characters a through to c, so `d[a-j]g` matches dag, dbg, and dig, but not dkg or dog.

- `{a, b}` matches the preceding character anywhere between a and b times, so `do{2,4}g` matches doog, dooog, and dooooog, but not dog or doooooog.

In addition, there are a series of special letters that, when preceded by a slash, take on a special meaning. Table 10-1 lists these special letters.

Table 10-1 Escaped Characters in Regular Expressions

Character	Description
-----------	-------------

<code>\n</code>	The newline character
-----------------	-----------------------

<code>\t</code>	Tab
-----------------	-----

<code>\d</code>	Any digit
-----------------	-----------

<code>\D</code>	Anything except a digit
-----------------	-------------------------

<code>\s</code>	Any whitespace such as a space, Tab, or newline
<code>\w</code>	Any alphanumeric character
<code>\W</code>	Any non-alphanumeric character

Obviously, this can lead to some problems if you want to match a `\` character. For this, or any other time where you want to match a special character (such as `.`, `*` or `+`), you can escape them with a `\`. Therefore, `\\w` matches `\w` and `\w` matches any alphanumeric character.

This can create a problem when entering strings in Python, since slashes need to be escaped there as well. For example:

```
>>> print("\\") \
>>> print("\\\\") \\
```

NOTE

This isn't a problem in this example because you've been passing the strings into Python, but if you're creating strings for regex matching in Python itself, you'll need to remember to double the amount of slashes you want to use in the regex.

Testing Your Knowledge

We've covered quite a log of regular expression information in quite a short space. To make sure everything is sinking in, take a look at the following exercise.

Take the following file:

```
aaa
a10
10
Hello Helllo Helloo
```

Which regex will match which lines? Try to work it out, then run the previous program on them to find out (you'll need to create a text file in Leafpad or download it from the website, where it's called chapter10-regex-test). For example, to check the first one, run:

```
python3 chapter10-regex.py -f chapter10-regex-test -r"."
```

Remember that the program uses `re.search()` not `re.match()`

■ . ■ \d ■ \D\d ■ l{3,4} ■ e* ■ e+ ■ [Ha] ■ \d{2,3} ■ 1?

Scripting with Networking

We looked at networking in Chapter 7, and we're not going to repeat ourselves here. Instead, we're going to look at ways you may need to use the network when scripting. The most common thing you'll need to do is copy files between two computers. There is an excellent module for this called Fabric; however, at the time of writing, it doesn't support Python 3. This is likely to change, but not in the immediate future.

In the absence of a module to handle the process, you could create the Python code from scratch and copy everything across. This is certainly possible, but it'll be quite long winded. There is a Linux command-line program called scp (secure copy) that does this, and you've already seen how to run command-line programs.

There is a slight problem though. When you run scp normally, it'll ask you for your password. This will cause a problem when scripting in Python because you can't easily tell it to answer questions. scp does, though, allow you to set it up with certificates so that if you're logged in as an authorised user on one machine,

you can log in without a password on another.

scp can be used to copy information between Linux computers (such as Raspberry Pis), so it will work between two Raspberry Pis, or between a Pi and a Linux server. Other (non-Linux) servers may also support it, but you'll have to ask your system administrator to set it up.

The first thing you need to do is log into the machine you want to transfer information to and run the following on the Linux command line, such as in LXTerminal. If you have only one Linux machine, you can try this out with a single computer, so run these commands on the same machine. If you only have remote access to this machine, you can do this via ssh.

```
ssh-keygen -t rsa
```

This will create two files in the .ssh folder in your home directory, called id_rsa and id_rsa.pub. These contain the public and private keys. id_rsa.pub is the public key, and you'll need to copy it across to the computer you wish to log in from. You can do this with a USB memory stick, cloud storage (such as Dropbox or Google Drive), or even email, but since we're talking about scp, you could do it with that. The format of an scp command is:

```
scp location1 location2
```

It simply copies the file from location1 to location2. If one of the locations is a normal file path, such as /home/pi/.ssh/id_rsa.pub, then scp deals with it on the local machine. However, if the location is in the form user@machine:/home/pi, then scp tries to log in as a user on the remote machine. machine can either be an IP address or a hostname. Typically with Pis, it'll be a IP address. So, if the IP address of the machine you want to copy your id_rsa.pub file is 192.168.0.10, and you want to use the user pi, the command would be

```
scp /home/pi/.ssh/id_rsa.pub pi@192.168.0.10:/home/pi
```

If you're trying this out on a single machine, you don't actually have to move the file, but just to try out scp, you can use the machine localhost. Therefore, the copy command is

```
scp /home/pi/.ssh/id_rsa.pub pi@localhost:/home/pi
```

Then you just need to copy the contents of the file into the authorized_keys file by logging into the machine you just copied the file to and running:

```
cat /home/pi/id_rsa.pub >> /home/pi/.ssh/ authorized_keys
```

That's been a little fiddly, but you should now be able to copy files from this machine to the other one (but not the other way around) without using a password. To try it out, enter the following:

```
touch test
scp test user@machine:/home/pi/
```

Where user, machine, and /home/pi are changed as appropriate. The first line simply creates an empty file called test. If everything works correctly, you won't need to enter a password.

With all that set up, you can copy files between machines using `subprocess.call()`. For example:

```
subprocess.call(["scp", "file1.py", "pi@localhost:/home/pi"])
```

Bringing It All Together

At the start of the chapter, we promised some Python that can make your regular computer chores easier, and while we've shown you lots of cool Python, we haven't fulfilled that promise yet. Now we will. In this section, we're going to create a Python program that can help you keep backup copies of your most useful files so that if disaster strikes, and your SD card breaks, you can get your data back.

We'll do this using most of what we've covered so far in this chapter, and one more module as well, `os`. This provides access to some of the operating system's functionality. Take a look at the following code (there's an explanation after it) The code's on the website as `chapter10-backup.py`:

```
import os
import tarfile
from optparse import OptionParser
from time import localtime
import datetime
import subprocess
import re

parser = OptionParser()
```



```
parser.add_option("-f", "--file", dest="filename",
help="filename to write backup to (if no option is give, backup will be used)",
metavar="FILE")
```

```
parser.add_option("-p", "--path", dest="path",
help="path to backup (if no option is give, ~ will be used)")
```

```
parser.add_option("-v", "--verbose", action="store_true", dest="verbose",
default=False,
help="print status messages to stdout")
```

```
parser.add_option("-i", "--images", action="store_true", dest="images",
default=False,
help="backup image files")
```

```
parser.add_option("-c", "--code", action="store_true", dest="code",
default=False,
help="backup code files")
```

```
parser.add_option("-d", "--documents", action="store_true", dest="documents",
default=False,
help="backup document files")
```

```
parser.add_option("-a", "--all", action="store_true", dest="all", default=False,
help="backup all filetypes (this overrides c, d & i)")
```

```
parser.add_option("-m", "--mtime", dest="mtime", default=False, help="backup
files modified less than this many days ago")
```

```
parser.add_option("-r", "--regex", dest="regex",
help="only back up filenames that match this regex")
```

```
parser.add_option("-s", "--server", dest="server", default=False, help="copy
backup file to this remote point (should be an scp location)")
```

```
options, arguments = parser.parse_args()
```

```
if options.filename:
    backup_file_name = options.filename + '.tar.gz'
else:
```

```

backup_file_name = "backup.tar.gz"

backup_tar = tarfile.open(backup_file_name, "w:gz")

file_types = {"code":[".py"],
"image":[".jpeg", ".jpg", ".png", ".gif"], "document":[".doc", "docx", ".odt",
".rtf"]} backup_types = [] all_types = False

if options.images:
backup_types.extend(file_types["image"])
if options.code:
backup_types.extend(file_types["code"])
if options.documents:
backup_types.extend(file_types["document"])

if len(backup_types) == 0 or options.all: all_types = True
if options.mtime:
try:
mtime_option = int(options.mtime)
except ValueError:
print("mtime option is not a valid integer.", "Ignoring option")
mtime_option = -1
else:
mtime_option = -1

if options.path:
if os.path.isdir(options.path):
directory = options.path

else:
print("Directory not found. Using ~") directory = os.getenv("HOME")

else:
directory = os.getenv("HOME")
for root, dirs, files in os.walk(directory):
for file_name in files:
if not options.regex or re.match(options.regex, file_name):

name, extension = os.path.splitext(os.path.join(root, file_name))
if (extension in backup_types) or all_types: modified_days =

```

```
(datetime.datetime.now() - datetime.datetime.fromtimestamp( os.path.getmtime(
os.path.join(root,
file_name))))).days
if mtime_option < 0 or modified_days < mtime_option: if options.verbose:
print("Adding ",
os.path.join(root,file_name), "last modified", modified_days, "days ago")
backup_tar.add(os.path.join(root,file_name)) if options.server:
subprocess.call(["scp", backup_file_name, options.server])
```

As you can tell from the numerous `parser.add_option()` calls, you can change this program to work the way you want it to. Its basic function is to copy files into a tar.gz file, which is a type of compressed archive that's popular on Linux systems. This file can then be copied automatically to a safe location on a separate server. Should anything then happen to the original files, you can resurrect them from this backup.

If you run `python3 chapter10-backup.py -help`, you'll get the following, which describes what it does:

Usage: chapter10-backup.py [options]

Options:

-h, --help show this help message and exit

-f FILE, --file=FILE filename to write backup to (if no option

is given, backup will be used)

-p PATH, --path=PATH path to backup (if no option is given, ~ will be used)

-v, --verbose print status messages to stdout

-i, --images backup image files

-c, --code backup code files

-d, --documents backup document files

-a, --all backup all filetypes (this overrides c, d & i)

-m MTIME, --mtime=MTIME

backup files modified less than this many days ago

-r REGEX, --regex=REGEX

only back up filenames that match this regex

-s SERVER, --server=SERVER
copy backup file to this remote point
(should be an scp location)

The basic usage is

```
python3 chapter10-backup.py -f backup.tar.gz -p /home/pi -s \  
pi@192.168.0.10:/home/pi/backups/
```

This tells the program to go through /home/pi and every subdirectory looking for files. It does this using the `os.walk()` function. This simply returns a collection of directories and files that you can move through using a for loop. This is done in the lines:

```
for root, dirs, files in os.walk(directory):  
    for file_name in files:
```

The first line will go through every directory in turn and return the path to that directory (root), the subdirectories (dirs), and the files (files). Since this program only cares about files, there is only one inner loop that iterates through all the files. `os.walk()` automatically goes through all the subdirectories, so you don't need to direct it to do that.

There are then some options you can choose to limit which files get selected. The basic ones limit it by filetype. `-i`, `-c`, and `-d` limit it to just images, code, and documents, respectively, while `-a` overrides these and selects all files (the default). These don't select perfectly, but work based on the dictionary of filetypes:

```
file_types = {"code":[".py"],  
"image":[".jpeg", ".jpg", ".png", ".gif"], "document":[".doc", ".docx", ".odt",  
".rtf"]}
```

If one or more of these is selected, it'll only back up files that end with extensions in the appropriate list. The program finds the file extension in the line:

```
name, extension = os.path.splitext(root, file_name)
```

The function `os.path.splitext()` (note that's split-ext notsplit-text) splits the filename into its two basic components, and returns these as two separate values. The first is the main part of the filename (which is captured in the `name` variable), and the part that comes after the final. (which is captured in the

extension variable). All that's left to do is check whether the extension is in the list `backup_types`, which we made by joining the lists of specified types. If you use these options, you should make sure that the entries in the list cover all the file types you actually want to back up.

The `-r` or `--regex` flag can be used to specify regular expressions that filenames must match to be included in the backup. This is in the line:

```
if not options.regex or re.match(options.regex, file_name):
```

Note that this uses `re.match()` rather than `re.search()`. This means that the entire filename must match the regular expression. For example, the regex `".*\..py"` will match all files with the extension `.py` (which is equivalent to using the `-c` flag).

The final option you can use is `-m` or `--mtime`, which is short for modified time. In other words, it backs up all files that were modified more recently than this number of days. It does this with the rather convoluted line:

```
modified_days = (datetime.datetime.now() -  
datetime.datetime.fromtimestamp( os.path.getmtime(  
os.path.join(root,file_name))))).days
```

```
if mtime_option < 0 or modified_days < mtime_option:
```

The part of this that does most of the work is `os.path.getmtime()`. This takes a filename complete with a path (that is, it needs `/home/pi/filename` rather than just `filename`), and it returns the timestamp when the file was last modified.

`os.path.join()` takes two arguments, a path and a filename, and it joins them to create what the previous function needs (you can't just join two strings together because sometimes paths have a `/` on the end and sometimes they don't).

The timestamp returned by `os.path.getmtime()` isn't a regular date, but the number of seconds since January 1st 1970 (this was the standard for Unix systems, and is used on Linux systems as well). Therefore, to get the number of days between now and when the file was last changed, we first have to convert it into a Python datetime using `datetime.datetime.fromtimestamp()`, and then take it off the current time.

With all that done, all that's left is to see if the result is less than the number of days given as an option. The initial clause in the `if (mtime_option < 0)` is

because the program sets `mtime_option` to -1 if there's a problem with what the user entered, or if there isn't an `mtime` option.

These are all the options that limit whether a file is selected. Once they've all been checked, the only thing left to do is add it to the tar file. This is done using the module `tarfile`, which provides a really simple way to handle these archives. You just need to open the file at the start. This is done with the line:

```
backup_tar = tarfile.open(backup_file_name, "w:gz")
```

The second parameter (`w:gz`) specifies that the file should be opened for writing and that it's a gzipped (that is, compressed) file. The appropriate files can then be added to this archive with:

```
backup_tar.add(os.path.join(root,file_name))
```

You should recognise the code that uses `scp` to copy the file to a remote server if the option is specified.

This program is one example of a script that can help keep your computer in order. It makes the task of creating backups trivial; however, you still have to remember to run it to create said backups. Python can't help you here, but there's a Linux feature called `crontab` that takes care of running programs at specific times. Because all the options of this program are on the command line, all you have to do is decide what you want to run and set `crontab` to start it at the appropriate time.

There are two main options for `crontab`: `-l` displays the list of the programs it currently has set to run, and `-e` opens up a text editor where you can edit what programs run when. Each task to run is on a separate line, and consist of five numbers or asterisks separated by spaces followed by the command. The five numbers relate to the minute of the hour, hour of the day, day of the month, month of the year, and day of the week the command should run, and an asterisk means any. Take, for example, the following (as a single line):

```
0 0 1 * * python3 /home/pi/chapter10-backup.py -s  
pi@192.168.0.10:/home/pi/backups
```

This will take a backup of your home directory (the default) at midnight on the first of every month. The following will run every day at midday:

```
12 0 * * * python3 /home/pi/chapter10-backup.py -s  
pi@192.168.0.10:/home/pi/backups
```

Working with Files in Python

In this chapter, you've seen a lot of ways to deal with files. However, you haven't yet actually opened any of them in Python to read or write data to (except the tar archive, but that was a special case). In this section, you'll see how to store information in text files, and then read it back.

This is actually really easy. All you need to do is call the `open()` function. For example, to open the file `myfile.txt` and print out every line, you need the following:

```
file_a = open("myfile.txt", encoding="utf-8")
for line in file_a: print(line.strip())
file_a.close()
```

The `open()` function creates a file object. It can take a number of parameters. The essential one is the filename, and encoding is a particularly useful one since it tells Python what format the file is in. Most text files are utf-8, and that's the one you should use when creating your own files.

NOTE

Note that you can't use `file` as a variable name because it's used for other things in Python.

Once you've opened the file, you can loop through it with a `for` loop. The only slightly unusual thing here is the `.strip()` that we've called on `line`. This is because each line of the file contains a newline character which will be printed, and the `print` function then adds its own newline, so without this you'd get a blank line between each of the printed lines.

Once you've opened a file, you should always close it. There is another way you can write this code so that it automatically closes. That is

```
with open("myfile.txt", encoding="utf-8") as file_a:
    for line in file_a:
        print(line.strip())
```

The two pieces of code do exactly the same thing. The `with` block will automatically close the file when the code block ends, so it's useful if you're prone to forgetting to close files. Writing to files is almost as easy. You just need to add a `mode="w"` to the parameters of `open()`, then you can write. Take a look at the following:

```
with open("myfile.txt", mode="w", encoding="utf-8") as file_a: for letter in
"abcde":
file_a.write(letter + "\n")
with open("myfile.txt", encoding="utf-8") as file_a: for line in file_a:
print(line.strip())
```

This will overwrite myfile.txt. However, if you want to add to the end of the file, you can use mode="a" (append). This will leave the original text intact and add new information to the bottom of the file.

Summary

After reading this chapter, you should understand the following a bit better:

- The Raspberry Pi runs a version of Linux.
- Linux has a different filesystem to Windows, based around the root directory, /.
- Linux has an entirely text-based mode.
- You can run commands on this text-based mode using the subprocess module.
- Commands in Linux can output to stdout and stderr.

- When writing scripts in Python, it's useful to get all the input as command-line flags if possible.

- Regular expressions are a way of matching patterns of text.
- scp can be used to copy files between computers.

- There are loads of functions in the os module to help you interact with the operating system (far more than we could cover here; take a look in the Python documentation for more information).

- The open() function can be used to open files for reading or writing.

Chapter 11

Interfacing with Hardware

UNLIKE MOST COMPUTERS, a Raspberry Pi has a series of General Purpose Inputs and Outputs (GPIOs) that allow you to interact with the world outside. These are the metal pins that stick up next to the SD card. You can use

them a bit like programmable switches to turn things on and off, or you can use them to get information from other sources. In short, they allow you to expand your Pi in any way you want. They're widely used by digital artists to create interactive displays and by robot builders to bring their creations to life. With a bit of imagination, there really is no limit to what you can achieve with the Pi's GPIOs and a few components.

Since this chapter is all about controlling things outside of the Pi, you will need a bit more equipment to try the examples here. It needn't be expensive though, and you can get started for just a few pounds. Even as you improve, most of the bits you'll need are cheap and easily available both online and in hobbyist stores.

Setting Up Your Hardware Options

Before jumping in and building circuits, the first thing you'll need is a way of connecting to the GPIO pins on the Pi. Since you can't just connect wires straight to the pins (actually you can solder directly onto them, but it's not recommended), there are a few options for accessing them, covered in the next sections.

Female to Male Jumper Wires

These are probably the simplest option. They simply fit over the top of the GPIO pins and allow you to then connect them to a solderless breadboard. This is the simplest way to get access to the GPIOs. It's also the method you'll see in the pictures of this book. It's fine for connecting a few pins, but it can get a little confusing if you're accessing a lot of pins at once. Take a look at Figure 11-1.

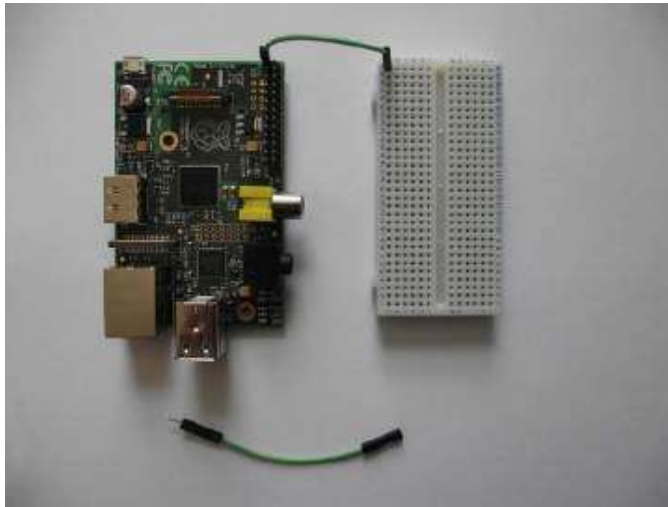


Figure 11-1: These jumper wires

have a female end that slots over the GPIO pins and a male end that you can fit into the solderless breadboard.

Pi Cobbler

The Pi Cobbler is a really simple design that takes the GPIO pins and connects them to a header that can be pushed into a solderless breadboard (see Figure 11-2). It doesn't add anything that you don't get by using jumper wires, but it's a bit tidier and it's less likely to get into a confusing knot of wires.

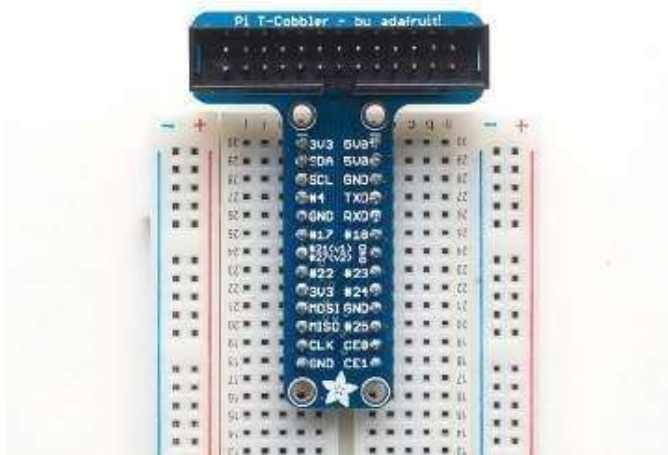


Figure 11-2: The Pi Cobbler gets all the GPIO pins onto the breadboard without risk of tangled wires or confusion.

© Adafruit Industries

Solderless Breadboard

You'll probably need one of these whichever option you go with. It's a way of connecting components to build circuits quickly, and allowing them to be taken apart again when you're finished. They come in different sizes, but they all follow the same basic layout. Down the long sides there are two parallel lines of pins that can be used as positive and negative rails. In between these, there are two banks of pins with a gap in the middle. These are connected in strips perpendicular to the long edge (see Figure 11-3). In the pictures, we'll be using a small solderless breadboard that doesn't have the positive and negative rails; however, you can do them on whatever size board you have. If you are interested in electronics, it's well worth getting a full-sized one as it will become the core of many of your projects.

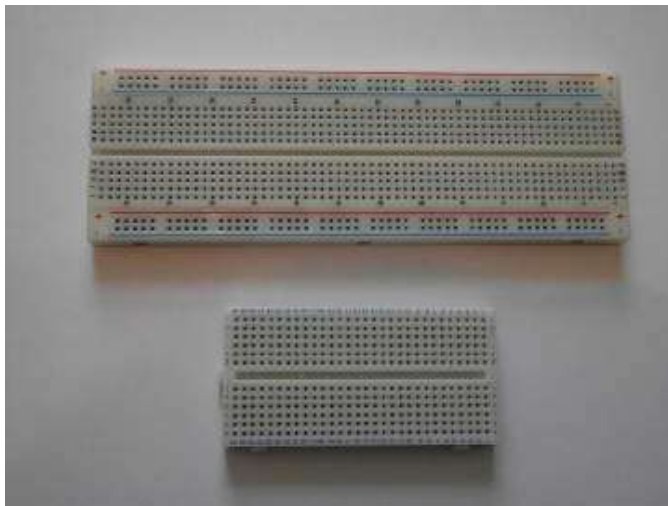


Figure 11-3: With solderless

breadboards, you can easily prototype circuits, and then dismantle them and build new circuits using the same components.

You can push most components straight into the holes, and make connections using either male-male jumpers, or pieces of single-core wire.

Stripboards and Prototyping Boards

Once you've prototyped your circuits on a solderless breadboard, you may wish to build them on a stripboard. This will permanently join all the components together, and is far more durable than a solderless breadboard. These are a little beyond the content in this chapter, but if you want to take things further, you'll probably soon find yourself using these.

An alternative is the prototyping board. This simply contains lots of holes you can solder into. There are no connections between the holes so you have to solder on whatever connections you want.

PCB Manufacturing

The most advanced option involves making your own PCBs (Printed Circuit Boards). There are a number of options for doing this, including commercial printing. This is for the final stage when you have a complete design. If you want to go down this route, Fritzing is an excellent resource (see <http://fritzing.org>). They produce software to help you design the boards, and a service to print them.

Getting the Best Tools

You won't necessarily need any tools to build simple circuits (as long as you have male-male jumpers for the breadboard), but there are a few that'll make life easier for you.

Wire Cutters/Strippers

These are pretty self-explanatory. You'll need these (they usually come as a single tool) if you're planning on using single-core wire to make connections. However, if you have a set of jumpers for your breadboard, these aren't necessary.

Multimeters

These devices give you the ability to check a range of different things, including the voltage, current, and resistance. If you're having problems with a circuit, they're invaluable tools to help you find out what's wrong. Without one, it's hard to tell if a particular connection is conducting well, or if a component is broken. They're also a lazy way of checking the value of a resistor (discussed later in this chapter).

Soldering Irons

Soldering irons are for creating permanent connections between two components or between one component and a circuit board. You'll need one if you're using stripboard, or if you buy a Raspberry Pi add-on that needs soldering together. We won't cover soldering in this chapter, but if you need to do it, there's an excellent guide called "Soldering Is Easy" at http://mightyohm.com/files/soldercomic/FullSolderComic_EN.pdf.

All of these tools are shown in Figure 11-4.



Figure 11-4: A set of tools for building your own hardware. None of them is essential for the projects in this chapter, though.

Hardware Needed for this Chapter

We've tried hard to keep the hardware for this chapter as simple, as easy to get, and as cheap as possible. In order to follow along, you'll need at least the following:

- Light emitting diode (LED)
- Resistors: 220 ohm, 1.1K ohm, and 6.2K ohm
- Solderless breadboard (any size should do)
- Jumpers for the breadboard (or single-core wire and a wire cutter)
- A way of connecting the breadboard to the Pi — either female-to-male jumpers or a Pi Cobbler
- MCP3008 chip
- Push switch
- Light-dependent resistor (LDR)

Each of these is discussed in the following sections.

The First Circuit

Before getting too far into the details of circuitry, let's create a simple circuit. You'll need a breadboard, an LED, a 220 ohm resistor, and some way of

connecting the GPIOs to the breadboard (either female-male jumpers or a Pi Cobbler).

This circuit is simply going to let you turn an LED (a type of light) on and off from Python. LEDs are a bit like mini light bulbs except for two things: firstly, they're much more powerefficient, so they shouldn't get too warm when running normally, and secondly they will only run one way round. That is, they have a positive leg and a negative leg, and the positive has to be connected to the positive and vice versa. The base of the LED should be round with a small flat section on one side. The flat side is next to the negative leg.

The resistor is there to stop too much power flowing through the circuit. As a general rule, you should always have at least one resistor of at least 220 ohms in a circuit; otherwise, you risk damaging your Pi and the other components. We'll look at this in a bit more detail later.

In this circuit, the resistor doesn't have to be 220 ohms; anywhere between 220 and 470 should be fine (you can try it with higher values, but the LED will be dim).

Resistors are colour-coded so you can tell their values. There are typically four or five bands of colour (a 220 ohm one will usually have four) and this should be red, red, black, and then silver or gold. Again, we'll discuss what this means a bit later.

The resistor and LED should be connected on the breadboard, as shown in Figure 11-5. The positive leg of the resistor (that is, the one not next to the flat side) should be the one that connects to the resistor.

R1 220 \pm 5%

1 51015 20

A
B
C
D
E

LED1 F Red (633nm)

G
H

I
J

1 5101520

Figure 11-5: Two diagrams for the circuit. The left one shows how to physically connect it while the right one shows how it's linked together.

In order to make sure the circuit works, connect the wire from the resistor to one of the 3.3v pins on the Raspberry Pi (see Figure 11-6), while the lead coming from the LED should go to a ground pin. If it's connected properly, the LED should light up.

SD
Card

3.3v 5v
*1 5v
*2 Ground

GPIO 4 GPIO 14
Ground GPIO 15
GPIO 17 GPIO 18
*3 Ground GPIO 23 GPIO 23
3.3v GPIO 24
GPIO 10 Ground GPIO 9 GPIO 25
GPIO 11 GPIO 8 Ground GPIO 7

*1 -- GPIO 0 on Revision 1 board GPIO 2 on Revision 2 boards *2 -- GPIO 1 on Revision 2 boards GPIO 3 on Revision 3 boards *3 -- GPIO 21 on Revision 1 boards
GPIO 26 on Revision 2 boards

Figure 11-6: Raspberry Pi pin layout. Don't try to work out a logic for the pin numbering; there is none.

This, though, is just using the Raspberry Pi as a power source. In order to be able to control the circuit from Python, you first need to install the RPi.GPIO module. First make sure you have pip (a tool to help you access modules) installed with the following command (in LXTerminal, not in Python):

```
sudo apt-get install python3-pip
```

Then get the library with this command (also in LXTerminal):
`sudo pip-3.2 install RPi.GPIO`

When you're working with the GPIO pins, you have to access the Raspberry Pi at a low level. Because of this, you can't run the Python scripts normally. Instead, you need to run them with superuser permissions. This sounds fancy, but in fact it just means prefixing commands with `sudo`. So, for example, if you want to run a script in LXTerminal, you need to run:

```
sudo python3 your-script.py
```

Alternatively, you can start a Python shell with:

```
sudo python3
```

Or you can start IDLE 3 with superuser permissions by running the following in LXTerminal:

```
sudo idle3
```

Once you've installed RPi.GPIO, you just need to connect the circuit to a one of the GPIO pins. Disconnect the pin from 3.3v and connect it to pin 22. Once this is done, open a Python session (don't forget `sudo`!) and enter the following:

```
>>> import RPi.GPIO as GPIO >>> GPIO.setmode(GPIO.BCM) >>>
GPIO.setup(22, GPIO.OUT) >>> GPIO.output(22, True) >>> GPIO.output(22,
False) >>> GPIO.output(22, True) >>> GPIO.output(22, False)
```

As you can see, setting the pin to `True` turns the LED on, while `False` turns it off. Figure 11-7 shows the running circuit.

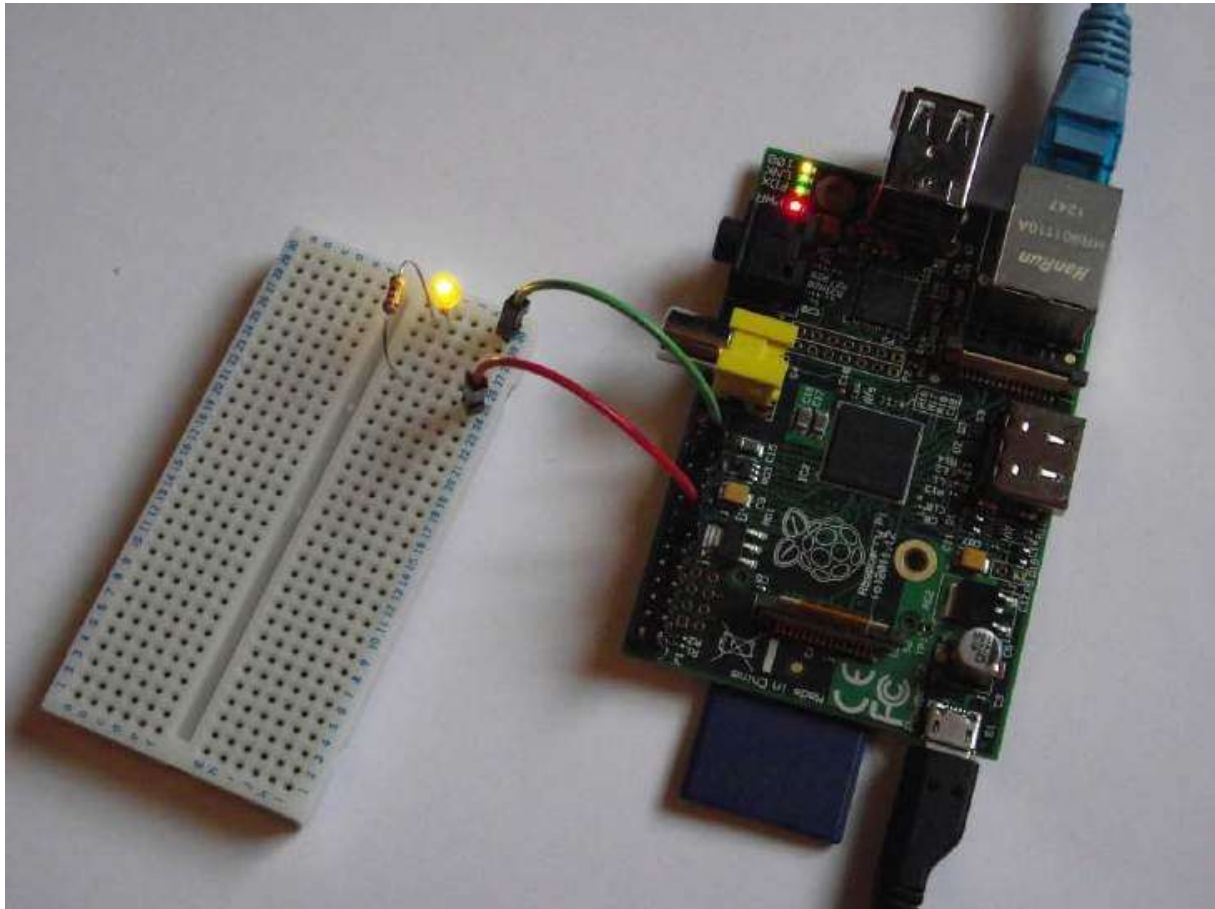


Figure 11-7: The fully connected circuit on a small, solderless breadboard.

About Circuits

In essence, a circuit contains three things — a source of power, something that does something, and a place for the power to go (that is, a ground). Circuits always have to have all three. Nothing will happen if you connect a power source, but no ground. If you just connect a power source to the ground, then you have what's called a short circuit and it can draw a very large current and damage the power supply (see the section “Protecting Your Pi”).

Circuits can vary from the very simple (like the one you made here) to the hugely complicated (such as a computer), but they all follow the same principles. In this chapter, we only have space to talk about the basics of using the Raspberry Pi's inputs and outputs, but if you're interested in circuits, you can take it as far as you want to go. If you want to take this further, the Penguin Tutor website has a good course to help you understand a little more about

what's going on: www.penguintutor.com/electronics/.

Protecting Your Pi

In general, it's very hard to physically damage a computer by programming. You might be able to corrupt some files (although even this is rare), but generally, no matter how much you mess things up, simply reinstalling the operating system will sort things out. However, when you're adding things to the GPIO, you're sending power directly to the CPU, and you can damage it. There are two properties of electricity that can cause problems, voltage and current.

With voltage, the rule is simple and very important. NEVER CONNECT MORE THAN 3.3 VOLTS TO A GPIO PIN. That's very important, which is why we're shouting. Quite a lot of hobby electronics components are designed to run at 5v because other processors run at that voltage level. However, if you connect these directly to the Pi, it can cause irreparable damage to the board. The result is known as bricking the Pi because afterwards its only use is as a brick.

CAUTION Never connect more than 3.3 volts to a GPIO pin!

You'll notice that the Raspberry Pi has a 5v pin. This is only there to power external circuits that don't come back to the Pi. If you need to connect a 5v device to the GPIO ports, you'll need a logic-level converter. These are available for a few pounds and convert 5v signals into 3.3v and vice versa.

Whereas voltage can be thought of as the amount of energy electricity has, current is the amount of it that's flowing through the wires. The two are connected by Ohm's law, which states:

Voltage = Current \times Resistance

Or, to put it another way:

Current = Voltage / Resistance

Current is measured in amps, voltage in volts, and resistance in ohms. In the previous circuit, there were 3.3 volts and 220 ohms, so that meant:

Current = $3.3 / 220 = 0.015\text{A}$ or 15mA

When using a Raspberry Pi, you must never draw more than 16mA from any one pin, or 50mA from all the GPIO pins combined. That means that you can light up only three LEDs at a time (or use more resistance to decrease the amount of current each one draws). It also means that you should never connect a GPIO pin

to a circuit unless there is at least 220 ohms of resistance in it. If you're ever unsure about resistors, always err on the side of caution and use larger ones than you have to.

Technically, this isn't actually correct because LEDs are a little different from many other components. This circuit will draw less power than that. However, unless you understand voltage drop on LEDs, it's best to stick with these guidelines.

If you need to draw more current, or just want to protect your Pi in case you accidentally draw more current, you can use an expansion board that has buffered input/output ports such as the PiFace, or alternatively, use a buffer-integrated circuit (IC or chip) to protect the GPIO ports.

Power Limits

The amount of current your Pi can supply is also limited by the amount of current it can get. Some power supplies will struggle to deliver much power, especially if there are several other things attached to the Pi like optical mice and USB memory. If you find that your Raspberry Pi becomes unstable or starts turning itself off when you're using GPIOs, then insufficient power may be the problem.

To combat this problem, you can upgrade to a power supply that can provide more current, or reduce the amount of current your Pi draws. Things like reducing overclocking in raspiconfig and removing non-essential peripherals will help.

Getting Input

In the previous example, you used the GPIOs to turn an LED on and off. This was the output side of GPIO — now it's time to look at the input. You'll also use the previous circuit, so leave it connected, but add a button.

The push button switch is a really simple component. When it's open (that is, not pressed) it doesn't connect the two pins, so no current can pass. When it's pressed, it connects the two pins and therefore acts just like a wire.

As you've seen, you'll need one resistor to stop too much current from flowing through the circuit. Just to be extra safe, we used a 1.1K ohm resistor here.

However, if you just connected a power supply to a resistor, to a button, to the GPIO, then when the switch is open, there won't be a circuit. If there's not a circuit, then the GPIO isn't on or off (or True or False, if you prefer). You need to ensure that there's a circuit that ties the GPIO pin to the ground (and therefore False) when the switch is open. This is known as a pull-down resistor because it pulls the GPIO down to 0v if there's nothing else connected to it. It needs to have quite a high value compared to the other resistor to ensure that enough of the electricity goes to the GPIO when the switch is closed. We used a 6.2K ohm resistor, but a 10K ohm one would work just as well.

Take a look at Figure 11-8 to see how to connect it. Note that the resistor between the GPIO and ground is the larger of the two. The button should, as the diagram shows, be connected between a 3.3v power source and GPIO 4.

```

15 10 15 20 A
B
C
D
E

F G H I J

15 10 15 20
R2
1k LED1±5% Red (635nm)

R1
220 ±5%

R3
6.8k ±5%
```

Figure 11-8: A circuit diagram that should help you understand what's going on.

Once it's set up, the following code will create a simple reactions game. It'll wait a random amount of time before turning the LED on. Then you have to press the button as soon as you can and it'll then tell you your reaction time. It's on the website as `chapter11-reaction.py`. Figure 11-9 shows the game in action.

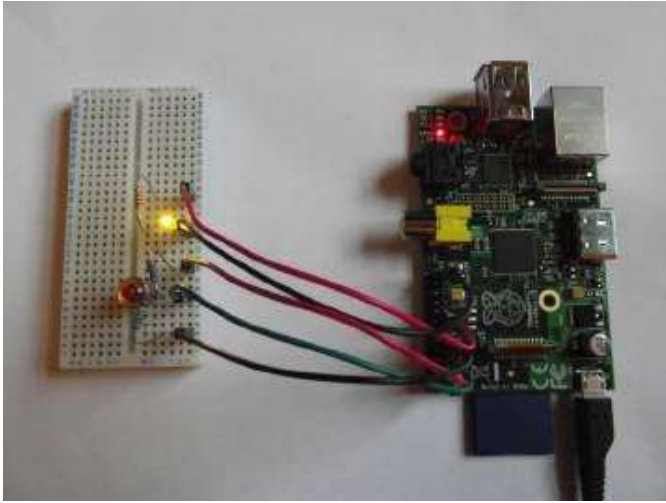


Figure 11-9: You can expand this

simple circuit to fit a wide range of projects.

```
import RPi.GPIO as GPIO
import time
import random
from datetime import datetime

GPIO.setmode(GPIO.BCM)
GPIO.setup(22, GPIO.OUT)
GPIO.setup(4, GPIO.IN)

GPIO.output(22, False)
random.seed()

while True:
    time.sleep(random.random()*10)
    start = datetime.now()
    GPIO.output(22, True)
    while not GPIO.input(4):
        pass
    print("Your reaction time: ",
          (datetime.now() - start).total_seconds())
    print("Get ready to try again.")
    GPIO.output(22, False)
```

Expanding the GPIO Options with I2C, SPI, and Serial

When you're connecting two computers together, you can use an Ethernet network. This is a series of standards that define things such as the physical

cable you use, and the way your computer addresses other computers on the network. As long as all the computers are compatible with Ethernet, they should all be able to talk to each other.

There are also communications protocols designed for working with smaller pieces of hardware, such as different chips. There are three main ones we'll look at in this chapter—SPI, I2C, and Serial.

The SPI Communications Protocol

SPI or Serial Peripheral Interface uses four wires to provide a two-directional communication channel between two or more devices—a master (usually the Raspberry Pi) and one or more slaves (typically chips). The four wires include a clock wire that keeps everything in time, the Master Out Slave In (MOSI), Master In Slave Out (MISO), and a Slave Select (SS). Simply connect the pins from the Raspberry Pi to the corresponding pins on the slave, and you should be ready to go.

There are a wide range of expansion options for SPI, for example, analogue input.

Raspberry Pis have a range of inputs and outputs, but they're all digital. That is, they can only read on or off. That's fine for some things, such as buttons and controlling LEDs, but sometimes you'll need to read or write data on a scale. For example, you might want to read data from a sensor such as a light or temperature sensor. These don't give on-or-off values, but a range.

Values like this (that fall within a range) are known as analogue values (as opposed to digital). In order to read them, you'll need an analogue-to-digital converter (ADC). For this chapter, we're using an MCP3008, which is a chip that provides eight analogue channels and communicates with the Raspberry Pi using SPI.

Figure 11-10 shows what the pins on the MCP3008 do, and Figure 11-11 shows how to connect the circuit.

Analogue channel 0

MCP3008

Analogue channel 7

Power in

Reference voltage Analogue ground Clock
 Data out
 Data in
 Slave select
 Digital ground

Figure 11-10: The eight pins on the left input 0 to 7, while the pins on the right are to control the chip. This diagram is based on looking down from the top. You should see a semicircle cut into the plastic case. This should match the one in the diagram.

15 10 15 20

A
 B
 C
 D



116
 E IC
 215

F
 G
 H 314
 I
 J

15 10 15 20 413
 512
 611
 710
 89

R2 220 LDR $\pm 5\%$

Figure 11-11: How to wire the circuit. You can change which GPIO pins the pins on the MCP3008 connect to by altering the appropriate values in the code. The code for it is as follows. You'll find it on the website as chapter11-spiadc.py. Figure 11-12 shows the connected circuit.

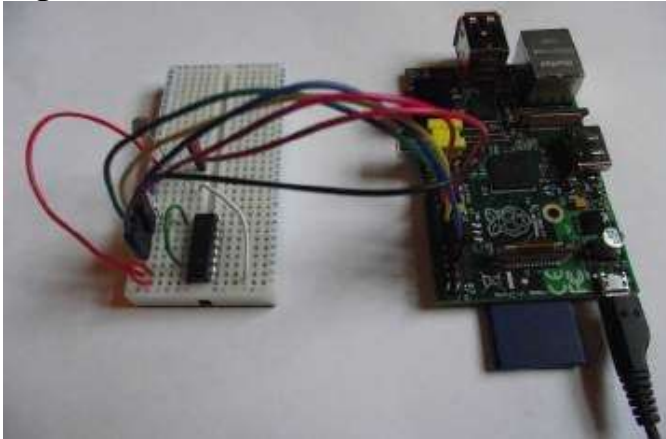


Figure 11-12: The MCP3008 being used to convert the analogue signal from the LDR to a digital signal for the Raspberry Pi.

```
import time
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

# read SPI data from MCP3008 chip, 8 possible adc's (0 thru 7) def
readadc(adcnun, clockpin, mosipin, misopin, cspin): if ((adcnun > 7) or
(adcnun < 0)): return -1
GPIO.output(cspin, True)
GPIO.output(clockpin, False) # start clock low GPIO.output(cspin, False) #
bring CS low

commandout = adcnun
commandout |= 0x18 # start bit + single-ended bit commandout <= 3 # we only
need to send 5 bits here for i in range(5):

if (commandout & 0x80):

GPIO.output(mosipin, True)
else:
GPIO.output(mosipin, False) commandout <= 1
GPIO.output(clockpin, True)
GPIO.output(clockpin, False)
```



```

adcout = 0
# read in one empty bit, one null bit and 10 ADC bits for i in range(12):

GPIO.output(clockpin, True)
GPIO.output(clockpin, False)
adcout <<= 1
if (GPIO.input(misopin)):

adcout |= 0x1
GPIO.output(cspin, True)
adcout >>= 1 # first bit is 'null' so drop it return adcout

SPICLK = 4 SPIMISO = 17 SPIMOSI = 18 SPICS = 23

# set up the SPI interface pins GPIO.setup(SPIMOSI, GPIO.OUT)
GPIO.setup(SPIMISO, GPIO.IN) GPIO.setup(SPICLK, GPIO.OUT)
GPIO.set'up(SPICS, GPIO.OUT)

ldr_adc = 0; last_read = 0
tolerance = 5

while True:
# we'll assume that the light didn't change input_changed = False
# read the analog pin
ldr_value = readadc(ldr_adc, SPICLK,

SPIMOSI, SPIMISO, SPICS) ldr_movement = abs(ldr_value - last_read)
if ( ldr_movement > tolerance ): input_changed = True

if ( input_changed ):
print('Light = ', int(ldr_value)) last_read = ldr_value

# hang out and do nothing for a half second time.sleep(0.5)

```

Don't worry too much about how the readadc() function works. It manipulates the individual bytes and sends and receives them down the MOSI and MISO wires. You can simply take it and use it to read the other ports on the ADC if you want to add more. Support for SPI is planned for the RPi.GPIO library, but at the time of writing, hadn't been implemented. You can see how the project's progressing at <http://code.google.com/p/raspberrypi-gpio-python/>.

The main loop then uses this function to get the value of the ADC connected to the LDR and (if it's changed by more than the threshold), print its value onto the screen.

The I2C Communications Protocol

Inter-Integrated Circuit (I2C or I²C, pronounced “I squared C”) is a more powerful protocol than SPI. It still uses four wires, but one of them is the power line, and another is the ground, so there are only two data wires. Up to 127 devices can be connected to an I2C bus, and it has addressing capabilities rather than the rather simplistic slave select on SPI.

Quick2wire makes a series of I2C boards that expand the functionality of the Pi. You don't need these boards to use I2C, although they do make the process a bit easier. They also make a Python module that can communicate over I2C regardless of whether you're using their boards. You can find it at <https://github.com/quick2wire/quick2wire-python-api>.

Like SPI, support for I2C is planned for RPi.GPIO. Again, follow the website for up-to-date information.

The Serial Communications Protocol

The previous two methods are generally used for sending binary data between devices. Serial communication, on the other hand, is generally used for sending text back and forwards (although there are exceptions in both cases).

Serial communications are supported by the pyserial module that you can get with the following command:

```
sudo pip install pyserial
```

Once the pyserial module is installed, you just have to create a serial connection, then you can use write() and read() methods to send and receive data. For example, the following code will send the message “Hello” out of the serial port. It was created to work with a Ciseco PiLite, which will then scroll the letters across the screen.

```
>>> import serial
>>> pilite = serial.Serial("/dev/ttyAMA0", baudrate=9600)
>>> pilite.open()
>>> pilite.write(bytes("Hello", "utf-8"))
```

Taking the Example Further

If you feel like taking this example further, try the following ideas.

Arduino

If you're interested in hobby electronics, perhaps the most useful kit is an Arduino. These are microcontroller boards slightly larger than Raspberry Pis. They have far more GPIOs (the exact number depends on the model), as well as analogue inputs and in some cases analogue outputs as well.

Perhaps the most useful thing about Arduinos, though, is the number of expansion boards (known as shields) that can slot onto them. With these you can quickly create powerful hardware with little (if any) wiring.

It is perfectly possible to connect your Raspberry Pi GPIOs to an Arduino to get them to talk via I2C or SPI, although most Arduino models run at 5v, so you'll need to use logic-level converters. They can also communicate over the USB using a serial connection. Some people consider Arduinos to be overkill when connected to a Pi, and it's true that almost anything you can do with an Arduino, you can also do with a Pi. However, the sheer amount of existing Arduino hardware and code makes them very useful companions to Raspberry Pis.

Perhaps the biggest drawback for readers of this book is that they're programmed in a dialect C++ rather than Python. The language has a different layout, but it's based on the same general principles. If you've read this far into the book, you shouldn't have too much difficulty picking up basic C++, although we won't deal with it here.

PiFace

The PiFace is an expansion board that slots straight onto the GPIO pins of the Raspberry Pi and is a really useful addition to the GPIO. As well as buffering the input and output (and so protecting your Pi and providing up to 500mA of current), it also has a pair of relays that can be used to drive higher powered components such as motors.

It's the same size as the Raspberry Pi, and fits neatly over the top. It's a great choice for getting started with simple robotics. It's around £25 (\$41) so, while not as cheap as getting the components yourself, it's a great value for its ease of

use and the range of projects it supports. Figure 11-13 shows the PiFace connected to a “Model A Pi.”



Figure 11-13: The buffered outputs

of the PiFace mean you can control far more without worrying about the small current limits on the Raspberry Pi.

Gertboard

The Gertboard is the Goliath of the Raspberry Pi GPIO options. It crams just about everything you could want into a board. It has the similar ATmega chips to Arduinos for learning microprocessing, a motor controller, digital-to-analogue converters, analogue-to-digital converters, push buttons, and a whole bunch of LEDs. Of course, all this comes at a cost, not just in terms of price (expect to pay a little under £60/\$98), but also in terms of size, which could make it unsuitable for many projects where this is important (such as with robotics).

It is perhaps best thought of as a board for learning about electronics and control, and once you have a good idea what’s going on, you can design a smaller circuit for implementing your project.

The fact that it’s designed by the Raspberry Pi Foundation’s hardware supremo Gert Van Loo means you can trust that it’s been built by someone who really knows the Raspberry Pi inside and out (literally).

Wireless Inventor’s Kit

Everything we’ve done here so far has used wires to connect various components. For most projects, this is fine. However, some projects require a bit

more freedom. Ciseco has put together a Wireless Inventor's Kit, which contains everything you need to get started with simple radio communications and the Raspberry Pi.

It uses a radio system that's designed to work at a lower level than WiFi networking, and it's best used to connect sensors to your Pi remotely, or to use your Pi to control circuits from a distance.

Trying Some Popular Projects

You can create almost anything with a handful of components and a Raspberry Pi, but to get you started, the following sections include a few ideas.

Robots

These can range from simple wheeled devices powered by two motors, to vastly complex humanoid walking robots, to the bizarre such as those that move like snakes or spiders. To work with robots, you'll need to learn how to control motors (the GPIOs don't provide enough power to do this straight from the pin). The PiFace has a pair of relays, which is one option for getting started.

You'll also probably need some servos. Servos are a bit like motors, but can be turned a few degrees at a time. There are also a vast array of sensors that can be hooked up to your Pi to help your creation see the world around them.

Home Automation

Imagine a world where you can control your heating and lights from your smartphone. It's not an impossible dream—it can be achieved with a Raspberry Pi, some circuitry, and Python. Chapter 7 showed you how to control a Python program from the web, so combined with what you've learned here, you should be able to make this dream a reality.

Burglar Alarms

Maybe these are less glamorous than home automation, but they might keep you safe. With a Raspberry Pi's camera and a few things like passive infrared (PIR) movement sensors, you should be able to create your very own Fort Knox.

Digital Art

Art doesn't have to be lifeless painting in dusty rooms. Nor does it have to be sculptures or poetry or dance. It can be anything you want it to be. One of the emerging forms is digital art, which uses some form of computing to add a new dimension to an installation. It could be, for example, to add shifting light patterns, or pressure sensitive pads to create feedback for people touching it. You can let your imagination run wild, and then use a Raspberry Pi to give life to your imagination's creation.

Summary

After reading this chapter, you should understand the following a bit better:

- The pins that stick up from one corner of the Raspberry Pi are General Purpose Input and Outputs (GPIOs) that can be programmed from within Python.
- The `RPi.GPIO` library offers a simple interface for switching the pins on and off, or reading values from them.
- You have to be careful when working with GPIOs because sending too much voltage, or drawing too much current, can cause irreparable damage to your Pi.
- You can expand your Pi by adding devices using protocols such as SPI, I2C, and Serial, all of which can be programmed from Python.
- One example of this is the MCP3008 chip, which has eight analogue-to-digital converters that can be read via SPI.
- Your Pi can control circuits of almost any complexity. The only limit is your imagination.

Chapter 12 Testing and Debugging

THERE ARE ALWAYS times when your code won't do what it should be doing. You see the inputs and work through the code, but somehow, it spits out an output that just shouldn't be possible. This can be one of the most infuriating parts of programming.

Investigating Bugs by Printing Out the Values

There are loads of ways to find out exactly what's happening, but one of the simplest is judicious use of `print()` statements. By using these to print out the value of every variable, you can usually get to the bottom of what's going on.

Take a look at the following code for a simple menu system. It doesn't produce any errors, but whatever input you give it, it always says "Unknown choice". (It's on the website as chapter12-debug.py):

```
choices = {1:"Start", 2:"Edit", 3:"Quit"}
for key, value in choices.items():
    print("Press ", key, " to ", value)
user_input = input("Enter choice: ")
if user_input in choices.values():
    print("You chose", choices[user_input])
else:
    print("Unknown choice")
```

Perhaps you've seen the problem already, but if you haven't, what's the best way to find it? The problem is that the if statement isn't correctly identifying when the user_input is valid, so add some print() statements to see what's happening:

```
choices = {1:"Start", 2:"Edit", 3:"Quit"}
for key, value in choices.items():
    print("Press ", key, " to ", value)
user_input = input("Enter choice: ")
print("user_input: ", user_input)
print("choices: ", choices)
print("choices.values(): ", choices.values())
if user_input in choices.values():
    print("You chose", choices[user_input])
else:
    print("Unknown choice")
```

Straight away you should see the problem: choices.values() should be choices.keys(). The code was checking the wrong part of the choices dictionary. Make the change in the code and try running it again. With that bug fixed, everything should be fine. Oh no, it still doesn't work! There must be another bug. Have another look at the output from the print statements:

```
user_input: 1
choices: {1: 'Start', 2: 'Edit', 3: 'Quit'}
choices.values(): dict_values(['Start', 'Edit', 'Quit'])
```

Can you see why it's failing? After the value of variable, the second most important thing is the data type of that value, so expand the print statements to

include more details about what's going on there:

```
print("user_input: ", user_input)
print("choices: ", choices)
print("choices.values(): ", choices.values())
print("type(user_input): ", type(user_input))
for key in choices.keys():
    print("type(key): ", type(key), "key: ", key)
If you run this, it should output:
Press 1 to Start
Press 2 to Edit
Press 3 to Quit
Enter choice: 1
user_input: 1
choices: {1: 'Start', 2: 'Edit', 3: 'Quit'}
choices.values(): dict_values(['Start', 'Edit', 'Quit'])
type(user_input): <class 'str'> type(key): <class 'int'> key: 1
type(key): <class 'int'> key: 2
type(key): <class 'int'> key: 3
Unknown choice
```

Now you can see that the cause of the problem is that `user_input` is a string, but the keys of `choices` are integers. The easiest way to solve this is to change the invocation of `choices` to make the keys strings:

```
choices = {"1": "Start", "2": "Edit", "3": "Quit"}
```

Now the basic logic of the program is working as expected; however, it still spits out loads of extra text that the user doesn't want to see. Obviously you could just delete the `print()` statements, but they may be useful again in the future. You can keep them in the code, but have a flag that can be set to turn them off and on like as follows:

```
debug = True
choices = {"1": "Start", "2": "Edit", "3": "Quit"}
for key, value in choices.items():
    print("Press ", key, " to ", value)
user_input = input("Enter choice: ")
if debug:
```



```
print("DEBUG user_input: ", user_input)
print("DEBUG choices: ", choices)
print("DEBUG choices.values(): ", choices.values()) print("DEBUG
type(user_input): ", type(user_input))
for key in choices.keys():
print("DEBUG type(key): ", type(key), "key: ", key)
if user_input in choices.keys():
print("You chose", choices[user_input])
else:
print("Unknown choice")
```

If you encounter any problems in the future, all you need to do is change the debug variable to True. Prefixing all the lines with DEBUG also makes it easy to see which output is normal, and which is debugging.

Finding Bugs by Testing

Debugging is the process of getting rid of problems in your programs. It can be quite challenging, but it can be even more difficult to find the problems in the first place. This might sound silly, but it's true. As a program gets larger, the number of different ways it can be used increases, and the more different ways something can be used, the more places there are to check for bugs.

Imagine, for example, a word processor that has options for style, page layouts, file formats, layout managers, and so on. Bugs could lurk in any of these areas, so it's important for the developers to check to make sure everything's running as it should. They could even hide in combinations; for example, a problem may occur only if a particular font is used with a particular layout.

Checking Bits of Code with Unit Tests

The most basic form of testing programs is the unit test. This is where you take one small piece of code and make sure it's behaving as it should. Typically, these are used to check that individual methods and functions are working properly.

Essentially, all a unit test does is run a piece of code with a particular set of inputs and check that their outputs are correct.

Take, for example, a function that takes a string of characters and returns a string with the same letters, but converted to uppercase. This could be implemented

and tested as follows:

```
def capitalise(input_string):
    output_string = ""
    for character in input_string:
        if character.isupper():
            output_string = output_string + character
        else:
            output_string = output_string + chr(ord(character)-32)
    return output_string
print(capitalise("helloWorld"))
```

This should behave as expected. It works because the UTF-8 character encoding that Python uses stores characters as numbers, and uppercase letters are 32 places below their lowercase counterparts.

In this example, we've used a simple test case, and we're printing it to the screen to check manually. We can get Python to check the test case for us using the unittest module in the following code:

```
import unittest
def capitalise(input_string):
    output_string = ""
    for character in input_string:
        if character.isupper(): output_string = output_string + character
        else:
            output_string = output_string + chr(ord(character)-32)
    return output_string
class Tests(unittest.TestCase):
    def test_1(self):
        self.assertEqual("HELLOWORLD", capitalise("helloWorld"))
if __name__ == '__main__':
    unittest.main()
```

This does more or less what the previous code did. If you run it, it'll check one string to make sure "helloWorld" goes to "HELLOWORLD". At this level, it's not much better or worse than just having a print statement.

When you run `unittest.main()`, Python runs every method in subclasses of

unittest. TestCase that start with test_. In this case it's just test_1. The real advantage of using unit tests is that you can combine lots of tests in order to check at a glance whether things have worked properly.

You can add a second test case that checks that the string "hello world" capitalises to "HELLO WORLD":

```
def test_2(self):
```

```
self.assertEqual("HELLO WORLD", capitalise("Hello World"))
```

If you run this, you should get the following output:

```
FAIL: test_2 (__main__.Tests)
```

```
-----Traceback (most recent call last):
```

```
File "capitalise.py", line 17, in test_2
```

```
self.assertEqual("HELLO WORLD", capitalise("Hello World"))
```

```
AssertionError: 'HELLO WORLD' != 'HELLO\x00WORLD'
```

```
- HELLO WORLD
```

```
? ^
```

```
+ HELLOWORLD
```

```
? ^
```

Oh dear, it looks like the test failed. You can see that the space wasn't properly dealt with. If you go back to the original code, you can see that the problem is that everything that isn't an uppercase character gets 32 taken off its UTF-8 value. Since space isn't uppercase, this happens to it too, but this isn't what the program should do.

Test cases should be designed to try a wide range of valid inputs. For example:

```
class Tests(unittest.TestCase):
```

```
def test_1(self):
```

```
self.assertEqual("HELLOWORLD", capitalise("helloWorld"))
```

```
def test_2(self):
```

```
self.assertEqual("HELLO WORLD", capitalise("Hello World"))
```

```
def test_3(self):
```

```
self.assertEqual('!"$%&*()_+==',
```

```
capitalise('!"$%&*()_+=='))
```

```
def test_4(self):
```

```
self.assertEqual("1234567890", capitalise("1234567890"))
```

```
def test_5(self):
```

```
self.assertEqual("HELLO WORLD", capitalise("HELLO WORLD"))
```

```
def test_6(self):
```

```
self.assertEqual("`~#@;:,<>/?",
capitalise("`~#@;:,<>/?"))
```

If you run these, you'll see that most fail. The problem is a flaw in the program logic. The code leaves it alone if it's an uppercase letter and changes it otherwise. However, what you want is for the code to change it if it's a lowercase letter, and leave it alone otherwise.

If you change the capitalise function to the following, it'll do this:

```
def capitalise(input_string):
    output_string = ""
    for character in input_string:
        if character.islower():
            output_string = output_string + chr(ord(character)-32)
        else:
            output_string = output_string + character
    return output_string
```

Now if you run the code, you should find that it passes all the tests.

By default, unittest will give you details only if one or more tests fail. Otherwise, it just returns an overall OK. For most purposes, this is what you want, but you can specify how verbose you want the output to be in two ways. If you're running the script from the command line, you can add the -v flag for more output. So, for example, if you've saved the program as capitalise.py, you can run the tests with verbose output using:

```
python3 capitalise.py -v
```

Alternatively, you can specify that you want a more verbose output in the code itself by changing:

```
if __name__ == '__main__':
    unittest.main()
to:
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Before going any further, we should point out that the capitalise() function is here for this example. If you actually need to capitalise text, you should use upper() method of the string class. For example:

```
>>> 'hello world'.upper()
```

Getting More Assertive

All these tests have a call to `self.assertEqual()`. This line tells the unit test module what the output of the test should be. That is, the test should pass if the two values passed are the same, and fail if they're different. This covers a large proportion of cases, but you may wish to check different things. There are a number of different assert methods that you can use in your tests.

These check that various structures are the same:

■ `assertSequencesEqual(sequence1, sequence2)` ■ `assertListEqual(list1, list2)` ■
`assertTupleEqual(tuple1, tuple2)` ■ `assertSetEqual(set1, set2)` ■
`assertDictEqual(dict1, dict2)`

With these structures, you may want to check that the value is in the structure rather than if two structures are the same. The following methods check if a value is in a structure:

■ `assertIn(value, structure)` ■ `assertNotIn(value, structure)`

Strings are a special type of structure and have their own method:

■ `assertMultiLineEqual(string1, string2)`

You can also check values using tests other than equality using these assert methods:

■ `assertNotEqual(value1, value2)` ■ `assertGreater(value1, value2)` ■
`assertGreaterEqual(value1, value2)` ■ `assertLess(value1, value2)` ■
`assertLessEqual(value1, value2)`

There are also a few that allow a margin of error:

■ `assertAlmostEqual(value1, value2)` ■ `assertNotAlmostEqual(value1, value2)`

These check that the two values differ (or not) by less than 0.000001. These are useful if you're testing floating-point functions that might have small rounding errors that are acceptable. You can also test anything that you can reduce to a True or False value using:

■ `assertTrue(value)` ■ `assertFalse(value)`

Whatever you want to check, each `test_` method should have one assert method call that is used to determine the success or failure of that particular test.

You can use these unit tests in a number of ways. There is a style of development called testdriven development that says that the tests are the first thing you

should write and then you use those tests as the specifications for the code. Most programmers, though, write the tests towards the end of development to make sure everything's working properly.

Using Test Suites for Regression Testing

Writing programs isn't usually a single effort. You don't usually sit down, create software, and then stop and go on to do something else. Instead, you generally code some of the features, distribute it to users, then fix bugs and add new features in later versions.

There is a risk of breaking things that once worked as you add new features, so it's important to test not only newly created things, but also older things that have worked. Testing older code is called regression testing, and having a properly ordered set of tests makes it really easy.

To make sure that you're not introducing bugs into previously working code, you should rerun the tests after you make any changes. However, as your programs become bigger, you'll end up with more and more tests. Eventually, you'll get to the point where it's not practical to run every test every time. You can group tests together into test suites. These allow you to test just particular areas of your program at a time.

Using the previous code, you can change the final code block to:

```
if __name__ == '__main__':
    letters_suite = unittest.TestSuite()
    symbols_suite = unittest.TestSuite()
    letters_suite.addTest(Tests("test_1"))
    symbols_suite.addTest(Tests("test_2")) symbols_suite.addTest(Tests("test_3"))
    symbols_suite.addTest(Tests("test_4"))
    symbols_suite.addTest(Tests("test_5"))
    symbols_suite.addTest(Tests("test_6"))
    all_suite = unittest.TestSuite()
    all_suite.addTest(letters_suite)
    all_suite.addTest(symbols_suite)
    unittest.TextTestRunner(verbosity=2).run(all_suite)
```

This bit of code by itself does exactly what the previous code block did. That is, it runs all the tests. However, it has grouped them into different test suites.

There's `letters_suite` that runs the test that checks letters, `symbols_suite` that runs the tests that check symbols, and `all_suite` that combines both of them. You can use the final line to run any of these three suites.

Using this code, you should find it quite easy to build a simple testing menu to help you make sure that everything's running smoothly.

Testing the Whole Package

Unit testing is great because you can automate it, and quickly check that everything's running properly. However, it doesn't cover everything. Even though everything seems to work properly by itself, you may still find that there are problems when everything comes together.

In commercial software development, after the unit tests have been done, the code will be passed to the quality assurance team to make sure everything's working fine. This team will outline a series of test cases that cover how the software will be used. It should check every aspect of the program and test it with a variety of inputs to make sure it behaves as expected. This is sometimes done manually with testers interacting with the software just as users would, and sometimes by specialist testing software that can simulate mouse and keyboard input. Of course, it's unlikely that you'll have a quality assurance team to help you with your software, but there are some things you can take from the professional approach. You should be methodical. Before you start testing, make a list of everything that the program does, and come up with test input and expected outputs. You can then go through this list and make sure it's all functioning correctly.

It's probably a bit excessive to do this after every code change, but you should do it periodically, and especially before any big releases.

Making Sure Your Software's Usable

By the time you've finished a program, you know everything there is to know about it. You know how to interact with it, how to get the best out of it, and what all the various options are. Your users, however, don't have any of this knowledge. Your software has to help them understand it and provide enough information for them to know what to do. After all, it doesn't matter how awesome your features are if the users don't know how to invoke them.

User testing is the area of testing devoted to making sure this is possible. In an

ideal world, you'd get a room full of people, sit them down in front of your software, and ask them to perform certain tasks and see how they get on. Again, you're unlikely to be able to do this. Sometimes you may be able to persuade a friend or relative to help you out, but the more programs you create, the fewer volunteers you seem to find. The only real solution to this is to listen to people using the software, and make sure to ask for feedback.

How Much Should You Test?

There's an old saying about software bugs that goes, "Absence of proof isn't proof of absence." Basically, no matter how much you test your software, there's no way of ever proving that there aren't any bugs in it. In fact, it's almost impossible to write software that doesn't have any bugs in it. The purpose of testing isn't to make perfect software, but to make software that's good enough. What "good enough" means will vary from project to project. The more important the software, the more you should test it, but all software deserves at least some testing. It's not as glamorous as implementing new features, but most of the time it is more important to have a few features that are properly tested than have loads that are buggy, so it's worth spending some time writing unit tests and making sure everything's working properly. After all, it could well be your data that the program loses when there's a problem.

Summary

After reading this chapter, you should understand the following a bit better:

- Debugging is the process of removing any problems from the code.
- Judicious use of `print()` statements can help you find out what the problems are.
- It sometimes helps to have a way of switching these `print()` statements on and off so you can reuse them if you find more problems.
- Testing is the process of finding bugs in code.
- Unit tests are the most basic form of testing and can be automated using the `unittest` module.
- Test cases can be grouped together into test suites to help you test particular areas of a program.
- It's easy to accidentally introduce new bugs when you add features, so you should always regression test after you make changes to your code.

- Unit tests won't pick up all problems though, so you should also test at a complete system level.
- Usability problems are also bugs, so you need to listen to your users to make sure they are addressed.

This brings us to the end of the book. Hopefully, you're now confident and knowledgeable enough to create your own programs. Don't worry if you feel you don't know everything about every aspect of Python, very few people do. If you ever get stuck, you can always refer back to this book, or the Python documentation at <http://docs.python.org/3/>.

Hopefully you've seen that programming isn't overly complex, and if you break up a problem into small steps, it's usually quite straightforward to code. The main thing to remember is that programming should be fun! Find an area that interests you and explore it. Despite its small size, there's very little you can't do with a Raspberry Pi.

Index

SYMBOLS

* (asterisk), 203
 : (colon), 17
 {} (curly braces), 37, 113, 114, 137 {{{}} (double curly braces), 153 == (double equal sign), 19
 "" (double quote marks), 30, 205
 - (hyphen), 9
 # (number), 16
 . (period), 203
 + (plus sign), 203
 " (quote marks), 30, 31, 114, 205 () round brackets, 34
 ; (semicolon), 113, 137
 [] (square brackets), 34
 /// (three slashes), 73
 <> (triangular brackets), 149

Application Programming Interface (API), 145, 159
 apt-get command, 7, 10, 77
 Arch, 7

- Arduino, 236–237
- arguments, 101–102, 154, 157, 158, 214
- array_spec (vertex_attrib), 114
- ArraySpec object, 114
- array_spec.glsl (), 114
- assert method call, 252
- asterisk (*), 203
- ATMega chips, 237
- attribute shader, 137
- attributes, 50
- audio. See sound
- audio Swiss Army knife (sox), 184, 196
- AWB, 192

A

- a flag, 202
- actions, as methods to run when event happens, 60
- ADC (analogue-to-digital converter), 232, 237–238, 239
- add_options () calls, 203
- addStretch () method call, 67
- all value, 157
- all, 202
- all_suite, 253
- Alpha (transparency) value, 111
- Alt+Tab, to switch back to non-Minecraft window, 164
- ambient light value, 122
- analogue input/signal/values, 232, 233
- analogue-to-digital converter (ADC), 232, 237–238, 239
- animation, 86, 169
- answering questions, by program, 181–182
- apple_freq loops, 174
- apple_in variable, 174

B

- babbage, 15, 17, 49
- Babbage, Charles (inventor of concept of computer), 15
- babbage.color () method, 19

- back button, in web browser, 65
- background image, for platform game, 99
- backups, 215
- backup_types list, 213
- Bash environment, 199
- begin_fill () method, 18
- bind (), 136
- birthday () method, 50
- BLEND_RGB_SUB flag, 187
- blit (), 187
- blit (draw), 186
- block types 18/22/246, 168
- Blue value, 111
- bookmarks picker, 68
- bool type, 31–32, 38, 42, 55
- box layout, 63, 66, 74
- boxsize variable, 24
- breadboard, 220, 221, 223, 224 break statement, 41
- bricking, 228
- brightness variable, 120, 121 Browser class, 65, 66, 68, 69
- Browser's self.menu_bar, 65 BrowserWindow class, 71, 72 buffered input/output ports, 229 buffer-integrated circuit (IC), 229 buffers, 112
- bumpedsphere.py, 136
- burglar alarms, 239
- button clicks, events as, 60, 65
- button presses, 74
- buttons, 7, 59, 65, 66, 73, 74, 76, 103, 229, 230, 232, 238
- bytes data type, 179

C

- C language, GLSL as similar to, 113
- C++ language, 237
- camera module, 5, 184, 190–193
- camera object, 186
- cameras, 177, 184, 188
- capital letters, 49–50
- capitalise () function, 250
- cascade file, 189

- cat command, 199
- cat/proc/cpuinfo, 200
- cd command, 8, 165
- Celsius (temperature), 148
- Central Processing Unit (CPU), 109, 112, 199
- chat server, building of, 141–144
- checkbounds () function, 24
- choices.keys (), 242
- choices.values (), 242
- Chrome, 68
- chunk_size, 179
- circle () method, 15
- circlearea (1) function, 20
- circuits, 224–227, 230, 232, 233
- classes. See also specific classes
 - advantage of, 51
 - building objects with, 49–54
 - defined, 49, 55
 - inheritance, 58
 - instances of, 50
 - in Qt graphical toolkit, 59
- client, 141
- clock wire, 231
- clock.tick (fps), 83
- closing tag, 149
- code, reusability of, 43–46, 51, 55
- code blocks
 - indentation in, 38–39
 - loops as, 17
- code editor, 21
- code examples
 - adding light to spinning cube, 123–126
 - chat server, 142–143
 - circuit with MCP3008, 233–235
 - converting speech to text, 180
 - copying files into tar.gz file, 209–212
 - for creating browser window, 63–64
 - design of platform game, 79–82
 - as downloads, 21
 - for making game Snake, 169–172
 - for saving sound, 177–179
 - speech recognition oracle (Pyri), 182–184
 - spinning cube, 116–119

- for spinning cube visualiser, 130–135
- collided () method, 96
- collide_get_y () method, 92
- colliderect () method, 88
- collisions, in platform game, 88–90, 96
- colon (:), 17
- color (colour1, colour2)
 - method, 18
- color variable, 120
- combo box/combo box changes, 68, 74
- command line, 8, 9. See also Linux
- command-line flags, 202–203
- communications protocols
 - Inter-Integrated Circuit (I2C), 235–236, 239
 - serial, 236, 239
 - Serial Peripheral Interface (SPI), 231–235, 236, 239
- comparison operators for numerical types, 29
- complex layouts, 63
- compressed archive, 212
- computer vision features, adding of with OpenCV, 187–190
- computers
 - communicating between two, 142, 144, 159
 - getting sound into, 177–181
 - as under-utilised, 1
- conditional logic, flow of, 19
- conditionals, 18–20
 - <condition>B, 19
- conditions, as true or false, 19
- connect calls, 60
- connection variable, 141
- controls, adding, 60–62
- convert (), 100
- cookies, 155, 160
- coordinate systems, 107
- copy.deepcopy (), 45
- count (), 36
- CPU (Central Processing Unit), 109, 112, 199
- crashing, 6, 196
- Creative Commons, 85
- crontab feature, 215
- Ctrl+C, 39, 41, 82
- curly braces ({}), 37, 113, 114, 137
- current, 228
- cursor, as turtle, 20
- cv2.Cascade Classifier (), 189

D

data types, 32, 243
database, 46–48, 51–54
datetime.datetime.

fromtimestamp (), 214 DEBUG prefix, 245
debugging, 241–245, 255
demos subfolder, 136
depth, creating sense of, 100
.dest="filename" flag, 203 detectMultiScale (), 189, 190 /dev/video0, 186
development files, 77
dialogs, defined, 72, 74
dictionaries

grouping values in, 55
non-sequential values in, 37–38 as odd data type, 40

dictionary of dictionaries, 48
Digia, 57
digital art, 239
digital inputs/outputs/signal, 232, 233 digital-to-analogue converter, 237
directory tree, 8
dirs subdirectories, 213
dmesg, use of to locate source of

problems, 12
Doom class, 93, 95, 96
dot product function, 122, 123 double curly braces ({}), 153 double equal sign (==), 19
double quote marks (""), 30, 205 draw () method call, 137
drawing pictures, 13–21, 166–167 Dropbox, 208
DVI input, 6

E

-e jpg option, 192
electricity properties, 228
ElementTree, 182

- elif, as short for else if, 19
- elif (else-if) statements, 42
- else statements, 42
- end_fill () method, 18
- equipment, required/optional, 5
- error messages, 6, 201
- escaped characters, in regular expressions, 206 /etc, 199
- Ethernet network, 231
- etho0, 140
- exceptions, catching, 42–43
- exercise 1, chapter 3, 33
- exercise 1, chapter 4, 62
- exercise 1, chapter 7, 149
- exercise 2, chapter 3, 43
- exercise 2, chapter 4, 73
- exercise 2, chapter 7, 155
- exercise 3, chapter 3, 48
- expansion boards (shields), 236
- extension .dev, 77
- extension .py, 11
- extra features, getting of from modules, 54–55 eye_matrix, 129

F

- F1, 25
- Fabric module, 207 factor_down value, 190 factor_up value, 190
- False value, 31–32, 38, 39, 55, 88, 91, 92, 226, 229, 252
- Fedora, 7
- female to male jumper wires, 219–220, 223
- file, cannot be used as variable name, 216
- filename attribute, 203
- filenames, 72, 213, 214
- files, working with, 216–217
- filter, 72
- Fireball class, 99
- fireball_high_speed global variable, 95
- fireball_low_speed global variable, 95

- fireball_plain, 96
- fireballs, in platform game, 94–96
- Firefox, 68
- 5v pin, 228
- FLAC format, 180, 184
- flac program, 184, 196
- flag -h, 202, 203
- flags, 187, 202–203, 205, 214, 250
- flip (), 186
- float, 120
- float () function, 32
- float type, 55
- floating-points, 32, 120
- for loops, 39–40, 55, 179, 187, 213, 216
- forecast_dict data structure, 148
- forms, 153–155, 160
- forums, 7
- forward button, in web browser, 65
- forward () method, 15
- 4D vector, 115
- fragment shader, 111, 113, 115, 116, 120
- fragment_glsl variable, 111, 113
- frame_rate variable, 127
- frames, 127
- freezing, how to counteract, 107
- Fritzing, 222
- from_byte () method, 127
- functions. See also specific functions
 - defining, 24
 - making code reusable with, 43–46, 55 methods as defined much like, 50
 - starting with lowercase letters, 49
 - use of to structure code, 20–21
 - value of in allowing code to be reused, 25

G

- games
 - cat and mouse, 21–25
 - distribution of, 99

- Minecraft, 163
- platform, 77–108
- simple reactions, 230
- Snake, 169–175

- General Purpose Inputs and Outputs (GPIOs), 2, 219
- Gertboard, 237–238

- get method, 152

- GET requests, 153, 154, 157

- getBlock () method, 173

- getOpenFileName () method, 72

- github.com/ashtons/picam, 191

- github.com/quick2wire/

- [quick2wire-python-api](https://github.com/quick2wire/python-api), 235

- glesutils.Texture.from_surface () method, 136

- gl_FragColor, 115, 120

- GL_LINE_STRIP, 116

- GL_POINTS, 116

- gl_PointSize, 115

- gl_Position variable, 114, 115, 129

- GLSL (Graphics Library Shader Language), 113, 122, 137

- GL_TRIANGLE_STRIP, 116

- Go button, in web browser, 66

- go_btn_clicked () method, 67

- Google

- Drive, 6, 208

- home page, 64

- request, 183

- search, 69

- speech-to-text web service, 180, 196

- google.com, 140

- GPIO options, expansion of, 231–236

- GPIO pins, 219, 220, 224–226, 228, 229, 232, 236, 237, 239

- GPIOs (General Purpose Inputs and Outputs), 2, 219

- graphical programming, 57–74

- graphical system (LXDE), 7. See also

- LXDE (Lightweight X11 Desktop Environment)

- graphical user interface (GUI) programming, 58–60, 128, 197
- graphics cards, 109
- Graphics Library Shader Language (GLSL), 113, 122, 137
- Graphics Processing Units (GPUs), 109, 111–112
- gray_frame image, 189
- Green value, 111
- grid layout, 59, 63, 66, 74
- grow_in list, 173
- GTK graphical toolkit, 57
- GUI, 197
- gzipped file (compressed file), 215

H

- Haar cascades, 188, 189, 190
- hardware, interfacing with, 219–239
- HDMI-to-DVI converter, 6
- HDMI-to-VGA converter, 6
- headers parameter, 180
- “Hello World!” website, 151
- HelloHandler class, 153, 154
- hello-template.html, 153, 155
- hello-template.html.
- self.get_argument (), 154
- help= parameters, 203
- help/--help flag, 9, 202, 203
- hierarchy, 197
- high-definition multimedia interface (HDMI)
 - video output, 6
- hobby electronics, 236
- home automation, 239
- home directory, 9
- /home, 199
- /home/pi root directory (HTTP), 150, 153 /home/pi/images with mkdir/home/
pi/images , 193

- horizontal boxes, 63
- hostnames, 140
- hosts, 139–140, 159
- HTML (Hypertext Markup Language), 149–150, 152, 153, 158, 160
- HTTP (Hypertext Transfer Protocol), 149, 150, 153, 157, 160, 180
- http://, use of, 67, 70
- HTTPError, 183
- https://, use of, 70
- http.server module, 151, 160 hyphen (-), 9

I

- I2C (Inter-Integrated Circuit) communications protocol, 235–236, 239
- IC (buffer-integrated circuit), 229
- icosa.py, 136
- IDE (Integrated Development Environment), 11, 21
- identical code, repetition of, 16
- IDLE 3, 11, 13, 18, 21, 226
- id_rsa file, 208
- id_rsa.pub file, 208
- if block, 41, 42, 89, 90, 100
- if clause, 20
- if condition, 19
- if .. elif .. else block, 18
- if line, 205
- if statements, 25, 31, 41–43
- ifconfig, 140
- ImageHandler class, 194
- images
 - background image for platform game, 99 creation of training ones, 190
 - for games, 84
- JPEGs, 192
- manipulation of, 187–190
- import copy, 45

- import lines, 54
- increment () function, 46
- indentation, 38–39, 41, 114
- index (), 36
- indexes, need for, 37
- index.html file, 150
- indices_face_1 tuple, 111
- information, storage of in text files, 216–217
- inheritance, 51, 58, 84
- initialise pygame, 83
- __init__ method, 50, 51, 60, 87, 98
- input, getting, 229–231
- input () function, 43
- instability, of Raspberry Pi, 229
- instances, created/initiated, 50
- int class, 127
- int () function, 32
- int type, 55
- integer (int), 29, 32, 244
- Integrated Development Environment (IDE), 11, 21
- Inter-Integrated Circuit (I2C) communications protocol, 235–236, 239
- Internet, sending data over, 193
- Internet Protocol (IP) address, 139, 140, 144
- IPv4 (version 4, IP address), 139, 144
- IPv6 (version 6, IP address), 139

J

- JavaScript Object Notation (JSON), 147–149, 159, 181
- JPEG image, 192
- json module, 147
- jump () method, 90, 99
- jumpers, 223
- jumping sound effect, 98

K

- Kelvin (temperature), 148

- kernel buffer, 7
- keyboard, 184
- keypress events, 58, 91, 96
- keys, need for, 37
- key/value pairs, in dictionaries, 37, 40

L

- LANs (local area networks), 144
- laplacian transform, 187
- layouts

- box layout, 63, 66, 74
- complex layouts, 63
- grid layout, 59, 63, 66, 74
- QGridLayout, 59
- QHBoxLayout, 63
- Qt graphical toolkit, 59
- QVBoxLayout, 63

- LDR (light-dependent resistor), 223, 233
- Leafpad, 11, 103
- LED (light emitting diode), 223, 224–225, 228, 229, 230, 238
- left () method, 15
- length () function, 121
- letters_suite, 253
- LibreOffice’s Writer, 11
- Lightweight X11 Desktop Environment (LXDE), 7–8, 57, 197
- lines image, 187
- Linux
 - command line, 180, 188, 197–199
 - command-line book, 9
 - command-line flags, 202–203
 - command-line program scp, 207
 - command-line tools, 196
 - compressed archive, 212
 - crontab feature, 215
 - flexibility of, 2
 - kernel, 197
 - log files, 205
 - as open source, 85
 - output from running command on, 201

- start of directories in, 8
- system commands, 157
- timestamp, 214
- using Arch or Fedora, 7
- list of lists, 35, 48
- listen (), 141
- lists, 33–34, 35–36, 37, 45, 55
- live streams, 193–196
- live.jpg, 194
- local area networks (LANs), 144
- local variables, 50, 84
- localhost, 140, 150
- local-only address, 144
- lo.eth0, 140
- logic-level converter, 228
- login method, 159
- login-fail.html, 158
- login-template.html, 158
- logout_template.html, 159
- loops
 - for, 39–40, 55, 179, 187, 213, 216 apple_freq loops, 174
 - bool type used in conditions for, 31 in cat and mouse game, 25
 - defined, 16
 - nested, 40–41
 - as way to control how Python moves through program, 25
 - while, 38, 55, 173, 187, 189
- lowercase letters, 49
- ls, 202
- LXDE (Lightweight X11 Desktop Environment), 7–8, 57, 197
- LXTerminal application, 8–11

M

- main function, 114, 115
- MainHandler class, 152
- master, 231
- Master In Slave Out (MISO), 231
- Master Out Slave In (MOSI), 231
- matrices, 112

- matrix multiplication, 129
- mc object, 165
- MCP3008 chip, 223, 232, 233, 239
- /media, 199
- /media/MyStick, 199
- menus, adding, 62, 71, 74
- metering mode, 192
- method calls, 74
- methods. See also specific methods

- in babbage, 15
- defined, 49
- as defined much like functions, 50 starting with lowercase letters, 49 use of to structure code, 20–21
- use of to take care of much of the work, 25

- micro USB power supply, 6
- microcontroller boards, 236
- microphones, 177
- microprocessing, 237
- Midori, 68, 85, 139
- Minecraft, 163–169

- minecraft-pi-0.1.1.tar.gz , 164 MISO (Master In Slave Out), 231
- mixer, initialising, 98
- m option, 214
- modified time (--mtime option), 214 module_example import line, 54 modules. See also specific modules

- advantages to creating, 55
- getting extra features from, 54–55 importation of, 25
- as time-savers for getting functionality, 66

- monitors, 5, 6
- MOSI (Master Out Slave In), 231
- motor controller, 237
- motors, 237, 238
- mouse, 57, 184, 229
- move (), 90

- `move_ip ()`, 90
- movement variable, 173
- `move_x ()` method, 94
- `move_y ()`, 89
- movies, making, 184–196
- MP3 sound files, 98, 126
- mpg123 command-line tool, 126
- `-mtime` option (modified time), 214 multimedia

- making movies, 184–196
- using PyAudio to get sound into your computer, 177–184

- multimeters, 222, 223
- music
 - copyrights on, 99
 - downloading source, 127
 - for spinning cube, 126–136
- mutable data types, 45
- `myfile.txt`, 217
- `mylevel` file, 101
- `MyStick`, 199

N

- name server, 140
- namespace clashes, 55 nested loops, 40–41 network interfaces, 140 network port, on model B Raspberry Pi, 5 network value, 157
- networking, scripting with, 207–209
- Nokia, 57
- non-sequential values, 37–38
- NOOBS, 7
- normal, as vector that sticks out of a face at 90

- degrees, 121–122, 123, 136
- `normalize ()` function, 122
- number (#), 16
- numbers, 29–30
- numerical operations, 29
- numerical operators, 30

O

- object recognition, 188
- objects
 - building of with classes, 49–54
 - as mutable data type, 45
- OGG files, 98
- Ohm's law, 228
- open () function, 216, 217
- open source, 84, 85
- Open Web Application Security Project, 159
- OpenCV, 196
- openCV module, 187–190, 191
- opencv_traincascades
 - program, 188
- opengameart.org, 84, 97, 98
- OpenGL, 109, 111, 112, 136
- OpenGL ES, 120
- OpenGL ES 2.0, 136
- opening tag, 149
- OpenWeatherMap.org, 147, 155
- operations on lists, 35–36
- operations on sets, 38
- operators for numerical types, comparison, 29
- OptionParser object, 203
- #options, 86
- optparse module, 202
- os module, 209, 217
- os.path.getmtime (), 214
- os.path.join (), 214
- os.path.splitext () function, 213
- os.walk () function, 213
- output.wav file, 179
- overclocking, 10, 107, 190
- owasp.org, 159

P

- page IDs, 153
- parallax scrolling, 100
- parameters
 - defined, 44
 - in methods start with self, 50
 - optional ones, 46

use of brackets with, 60–61

Parent class, 50, 51

parser.add_option (), 203, 212 pass statement, 82

passiver infrared (PIR) movement

sensors, 239

passwords, 8, 207, 209

PCBs (Printed Circuit Boards), 222 Penguin Tutor website, 227

period (.), 203

peripherals, 6, 205, 229

permissions, superuser, 226

Person class, 50, 51

physics, realistic game, 103–108

Pi Cobbler, 220, 223

pi directory, 9

pictures, drawing of, 13–21, 166–167 PiFace, 229, 237, 238

pi.minecraft.net, 164

pin 22, 226

pin joint, 106

pip (tool), 224–225

PIR (passive infrared) movement

sensors, 239

platform game

adding movement to sprites, 86

building of, 79–82

creating a world, 86–88

detecting collisions, 88–90

drawing sprites for, 84–85

making a challenge, 93–97

making it your own, 97–103

moving left and right, 90–92

reaching the goal, 92–93

Player class, 83, 84, 86, 90, 94, 98, 99 player object, 165

player_image variable, 84

player_plain, 85, 96

player.setPos(x, y, z), 175 plus sign (+), 203

PNG files, 84, 100, 187, 192

- pop (), 173
- pop (x), 36
- port 22, 140
- port 80, 140, 150
- port 8000, 150
- port 8888, 152
- ports, 140, 141
- position_matrix, 129
- POST requests, 153, 154, 160
- postTo-Chat () method, 173
- power
 - as most common cause of problems with Raspberry Pi, 6
 - with power comes complexity, 137
 - supply/limits, 5, 229
- powered USB hub, 5, 6
- prefixes
 - DEBUG prefix, 245
 - with module name, 55
- protocol prefix, 67, 70, 73
- sudo prefix, 11, 226
- <pre>B tags, 158
- print () function, 20, 43
- print () statements, 201, 241, 242, 243, 244, 255
- Printed Circuit Boards (PCBs), 222
- print-file.py, 203
- private key, 208
- problems, common ones, 6
- processes value, 157
- /proc, 199
- /proc/cpuinfo, 199
- program flow, control of, 38–39
- programmers/programming, 1
- projects
 - burglar alarms, 239
 - digital art, 239
 - home automation, 239
 - robots, 238
 - protection, for Pi, 228–229

- protocol prefix, 67, 70, 73
- prototyping boards, 221
- pseudo-widgets, 67
- public address, 144
- public key, 208
- pull-down resistor, 229
- push button switch, 223, 229, 238
- PyAudio module, 177, 196
- PyAudio object, 179
- pygame directory, 78, 185
- PyGame mixer, 127
- PyGame module, 77, 82–86, 103, 106, 107, 110, 173, 185, 187, 188, 196
- pygame.key.get_pressed (), 91 pygame.sprite.Sprite, 84
- PyMunk module, 103, 106, 107, 108
- Pyri (speech recognition oracle), 182
- pyserial module, 236
- pyside module, 57, 62, 73, 74, 77
- Pythagorean theorem, 121
- Python
 - bringing everything together, 46–48
 - building objects with classes, 49–54
 - controlling the way the program flows, 38–43
 - documentation for, 25, 255
 - getting extra features from modules, 54–55
 - making code reusable with functions, 43–46
 - shell, 11, 226
 - storing values in structures, 33–38
 - turtle module, 13–21
 - using of from saved programs, 11
 - using of from shell, 11
 - variables, values, and types, 27–33
 - ways to write programs for, 11
- Python 2, 187
- Python 3, 185, 187, 207
- Python API, 164
- Python interpreter, 11, 27, 30, 39, 165, 200 Python variable to print, 153
- python3 client.py, 142 python3 server.py, 142

Q

q key, 188

QActions, 71

QColor type, 73

QColorDialog, 73

QColorDialog.getColor (), 73 QComboBox, 68, 69

QFileDialog, 72, 73

QGridLayout, 59

QHBoxLayout, 63

QLineEdit, 66

QLineEditEntry, 70

QMainWindow, 62, 64, 71, 74

QPlainTextEdit, 66

QPushButton, 70

QPushButtons, 65

QSlider, 70

Qt graphical toolkit, 57, 59, 62, 66, 67, 72,

73, 74

QTextEdit, 66

questions, asking of the program, 181–182 Quick2wire, 235

quote marks ("), 30, 31, 114, 205

QVBoxLayout, 63

QWebView, 63, 64, 65, 70, 71, 73

Qwidget, 59

raspi-config, 10, 107, 190 raspistill, 191, 192

reactions game, simple, 230

read () method, 236

realistic game physics, 103–108 record_sound () function, 179 Rect class, 78, 90

recv () method, 141

Red value, 111

red varying variable, 129

-regex flag, 205, 214

regression testing, 252–253, 255 regular expressions (regex), 203–207,

214, 217

relays, 237, 238

re.match (), 204, 207, 214

- RenderPlain, 85, 96
- re.search (), 204, 207, 214
- reset () method, 95
- resistors, 223, 224–225, 228, 229
- RGB values, 73, 87
- right () method, 15
- robotics, 190, 237, 238
- root directory, 8, 197–198, 213, 217
- round brackets (), 34
- RPiGL module, 110, 136
- RPi.GPIO module, 224–225, 226, 235, 239
- r flag, 205, 214
- Run button, 103

R

- randint (a , b) method, 95
- range (x, y), 39
- Raspberry Pi
 - as great device on which to learn programming, 2
 - model A, 5
 - model B, 5
 - online shop, 6
 - speaking to, 180–181
 - turning of into mirror, 185–186
 - Raspberry Pi Foundation camera module, 184, 190–193, 196
- Raspbian, 6, 7, 197, 199, 205

S

- sample_rate , 179
- scaling factor, 112
- scenery, adding of, 99–101
- scp (secure copy), 207, 208, 215, 217
- Screen class, 49
- screen_x variable, 95
- scripting
 - bringing it all together, 209–215
 - command-line flags, 202–203
 - getting started with Linux command line,

197–199

networking with, 207–209

regular expressions (regex), 203–207 using subprocess module, 200–202

working with files in Python, 216–217

SD card, 5, 6, 196, 219

search bar/box, in web browser, 69, 70

security, 155–159, 196

self.assertEqual () call, 250

self.image variable, 84

self.programl.uniform.transform_ matrix.value attribute, 116

self.rect variable, 84

self.render (), 152

self.url_entry.text () parameter, 67 self.verteces_buffer.draw (), 116

self.webview.load call, 69

self.write (), 152

semicolon (;), 113, 137

send () method, 141

sequences, 33

serial communications protocol, 236

Serial Peripheral Interface (SPI) communications protocol, 231–235, 236, 239

server socket, 141

servers, 141, 160

servos, 238

setBlock method, 165, 166, 175

setBlocks (), 175

setMaximumSize () method, 66, 67 setMinimumSize () method, 67

sets

grouping values in, 55

as mutable data type, 45

non-sequential values in, 37–38

operations on, 38

setup.py, 103

shader objects, 111

shaders, 128, 136. See also attribute shader; fragment shader; vertex shader

shields (expansion boards), 236

simple reactions game, 230

size variable, 187

skeleton of the program, 79

- Slave Select (SS), 231
- slaves, 231
- slider movements, 74
- small_gray_frame image, 189
- Snake (game), 169–175
- SoC (System on a Chip), 109
- socket module, 141
- sockets, 140, 141, 159
- software
 - installation of, 10–11
 - making sure it's usable, 254
 - soldering, guide on, 222
 - soldering irons, 222, 223
 - solderless breadboard, 220, 221, 223
 - sort (), 36
- sound
 - adding of to platform game, 98–99
 - calculating level of in spinning cube, 129–135
 - getting of into computer, 177–181
 - recording of, 179
- sox (audio Swiss Army knife), 184, 196
- Space module, 106
- speaking, to Pi, 180–181
- special characters, 203, 205
- speech recognition oracle (Pyri), 182
- speech-to-text web service, 180, 196
- speed_y local variable, 84
- SPI (Serial Peripheral Interface) communications protocol, 231–235, 236, 239
- spinbox, 59, 60
- spinning cube
 - adding some texture, 136
 - addling light, 120–126
 - bringing it all together, 116–120
 - building the 3D model, 128–129
 - calculating sound level, 129–135
 - creation of, 110–116
 - making screen dance, 126–128
 - taking things further, 135–136
- Sprite class, 78, 94, 99

- sprites, 84–85, 86, 97
- square brackets ([]), 34
- square () function, 44
- SS (Slave Select), 231
- .ssh folder, 208
- statements. See specific statements
- stderr.Stdout (or standard out), 201 stdout.Stderr (or standard error), 201
- stdout=subprocess.PIPE parameter, 201 storage value, 157
- str (string, piece of text), 29
- str () function, 32
- stretching matrix, 128
- string operations, 31
- string type, 30
- strings
 - converting other data types to, 32
 - creation of, 30
 - defined, 34
 - each level of platform game as list of, 86
 - stripboards, 221, 222, 223
 - .strip (), 216
- structures, storing of values in, 33–37
- subclasses, 51
- subdirectories, 8
- subprocess module, 158, 200–201
- subprocess.call (), 184, 200, 208 subprocess.Popen (), 195, 200
- sudo prefix, 11, 226
- superclasses, 51, 58, 152
- superuser permissions, 226
- symbols_suite, 253
- sys module, 101
- sys.argv, 101, 167
- syslog value, 157
- SysStatusHandler, 157
- sysstatus-template.html, 157 /sys, 199
- system log file (syslog), 7, 205
- System on a Chip (SoC), 109
- system value, 157

T

- tags, for web pages, 149
- tarfile module, 214–215 tar.gz file, 212
- templates, 153, 154–155, 160 terminal command, 164–165 test_ method, 252
- test suites, 252–253, 255
- test your knowledge

- adding button to launch a

- QColorDialog , 73
- extending turtle controller program, 62 getting current weather, 149
- of Python statements, 32–33, 38
- on regular expressions, 206–207

- test-drive development, 252

- testing

- defined, 255

- getting more assertive, 250–252

- how much should you test?, 254–255

- purpose of, 254

- regression testing, 252–253

- unit tests, 245–250

- test.png, 191

- text, keeping of in strings, 30

- text editors, Leafpad, 11, 103

- text input/entry, in web browser, 66

- texture, adding of to spinning cube, 136 three slashes (///), 73

- 3D graphics, 109, 112, 119, 136, 163,

- 168, 169, 173

- three-dimensional float (3f) vectors, 114 time-lapse cartoon video, 192

- time.sleep (), 25

- timestamp, 214

- Tk graphical toolkit, 57

- tools, getting the best, 222–223

- tornado module, 151

- Tornado server, 193, 196

- Tornado web application, 152, 153, 160, 194 tornado.web.RequestHandler, 152

- tornado.web.StaticFileHandler, 194 touch screen, 57

- training images, 190

- transform_matrix variable, 114,

- 116, 123
- `transforms.compose ()`, 112, 128 `transforms.rotation_degrees()`, 113
- `transforms.scaling ()`, 113
- `transforms.stretching ()`, 113 translation matrix, 128
- triangular brackets (<>), 149
- True value, 31–32, 39, 42, 88, 91, 92, 226, 229, 245, 252
- tuples, 33–34, 35, 36, 37, 55
- Turtle class/turtle class, 49, 51 turtle module, 13–21, 49
- TurtleControl class, 59, 60 `turtle.Screen ()` statement, 49 `turtle.Turtle ()` statement, 49 tweeting/tweets, 144–146
- Twitter, 145, 159
- 2D graphics, 109, 168, 169, 173 two-dimensional list, 35
- `type ()` function, 28
- type IDs, 166
- type page argument, 157
- `typeID` type, 165
- types
 - converting between data types, 32
 - values as associated with, 28–29

U

- uniform variables, 137
- unit tests, 245–250, 252, 253, 255
- `unittest.main ()`, 247
- universal character set Transformation Format
- 8-bit (UTF-8), 141, 246, 248
- Universal Resource Locator (URL), 67
- Unix systems, 214
- `update ()` method, 87, 94
- `upper ()` method, 250
- URL entry bar/box/control, 67, 70, 71 `urllib.request.Request ()`, 180 usability problems, 255
- USB devices, 205
- USB keyboard, 5, 6
- USB memory stick, 6, 199, 208, 229
- USB microphone, 177

- USB mouse, 5, 6
- USB webcam, 5, 6, 184–187, 190–191, 196
- USB WiFi dongle, 5, 6
- use () method, 111
- user-info.html, 154
- user_input, 242, 244
- usernames, 8, 21, 22, 33, 160
- user-template.html, 154
- UTF-8 (universal character set Transformation

- Format 8-bit), 141, 246, 248

V

- valueChanged action, 71

- ValueError, 43

- values

- associated with types, 28–29, 55 as being numbers or text, 28
- storing of in structures, 33–37

- Van Loo, Gert (Raspberry Pi Foundation’s hardware supremo), 238

- variable_name_getRgb () [:3], 73

- variables. See also specific variables

- defined, 24, 27, 55

- naming convention, 28, 49

- for storing information, 25

- understanding, 24

- /var/log folder, 205

- varying keyword, 121

- varying variables, 120, 129

- varying vectors, 137

- varying.uniform variable, 120

- vector-matrix algebra, 112

- vectors, 112, 120, 121, 122, 137

- vertex shader, 111, 113, 115, 116, 120, 122, 137

- vertex_attrib attribute, 114

- vertex_glsl variable, 111, 113

- vertical boxes, 63

- vertices list, 111

- VGA input, 6
- video, making, 184–196
- video encoder, 192
- virtual filesystems, 199
- visualiser, 130
- voice control, 180
- voice-driven menu, 184
- voltage, 228
- v flag, 250

W

- w2schools.org (HTML resource), 150
- WAV files, 126, 177, 179, 184, 196
- wave module, 127, 179
- weather forecasts, with JSON, 147–149
- Web, getting on, 149–155
- web browser
 - adding of window menus to, 71–73
 - adding speech recognition oracle to, 184
 - creation of, 62–71
- web pages, 63, 149
- web server, 150
- webcam viewer, 188
- webcams, 6
- websites, making them dynamic, 151–152
- webView's load method, 67
- w:gz parameter, 215
- while loops, 25, 38, 42, 55, 173, 187, 189
- widgets, 59, 62, 63, 66, 71, 74
- window menus, adding of, 71–73
- windowing system, 7
- wire cutters/strippers, 222, 223
- Wireless Inventor's Kit, 238
- with block, 216
- Wolfram Alpha, 181, 182
- Wolfram Alpha app ID, 182
- wolframalpha.com, 181
- word processors, LibreOffice's Writer, 11
- World class, 87, 92
- world_cube_net.png, 136
- write () method, 236

X

x and y attributes, 84, 90

x coordinate (horizontal position), 112, 165 XML, 182, 188

xml module, 182

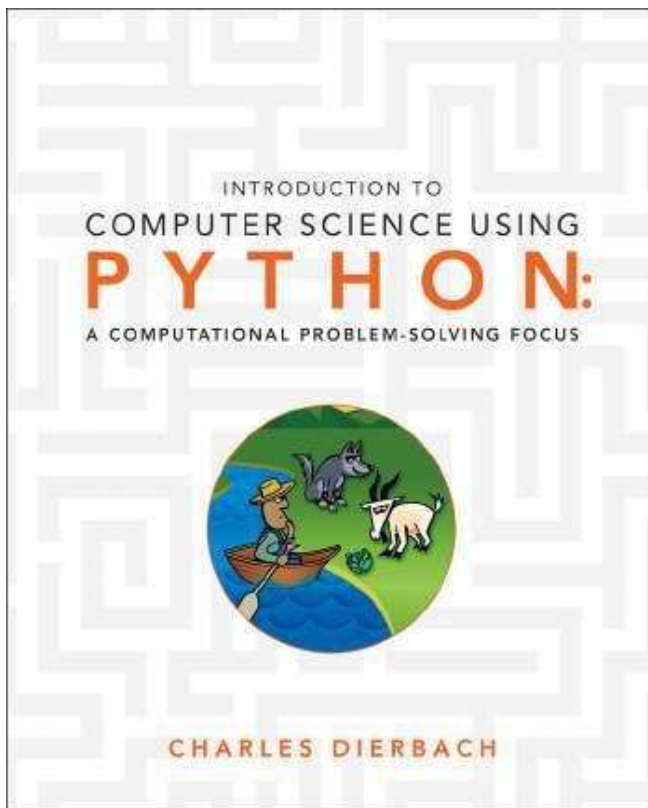
Y

y coordinate (vertical position), 112, 165

Z

z coordinate (depth), 112, 165 zoom slider bar, in web browser, 69

zoom_changed () method, 71



Science Using Python: A Computational Problem-Solving Focus

This page is intentionally left blank

Science Using Python: A Computational Problem-Solving Focus

Charles Dierbach

VP & Executive Publisher: Executive Editor:

Assistant Editor:

Marketing Manager:

Marketing Assistant:

Photo Editor:

Cover Designer:

Associate Production Manager: Production Editor:

Cover Illustration:

Don Fowley

Beth Lang Golub Samantha Mandel Christopher Ruel Ashley Tomeck Hilary

Newman Thomas Nery

Joyce Poh

Jolene Ling

Norm Christiansen

This book was set in 10/12 Times LT Std by Aptara. Text and cover were printed and bound by Courier Kendallville.

This book is printed on acid free paper.

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: www.wiley.com/go/citizenship.

Copyright © 2013 John Wiley & Sons, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc.,

111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008,
website <http://www.wiley.com/go/permissions>.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free of charge return mailing label are available at www.wiley.com/go/returnlabel. If you have chosen to adopt this textbook for use in your course, please accept this book as your complimentary desk copy. Outside of the United States, please contact your local sales representative.

Library of Congress Cataloging-in-Publication Data
Dierbach, Charles, 1953–

Introduction to Computer Science Using Python: A Computational Problem-Solving Focus/Charles Dierbach. p. cm.

Includes index.

ISBN 978-0-470-55515-6 (pbk.)

1. Python (Computer program language) I. Title.

QA76.73.P98D547 2012

005.13'3—dc23

2012027172

Printed in the United States of America 10 9 8 7 6 5 4 3 2 1

DEDICATION

To my wife Chen Jin, and our sons Jayden and Bryson.

Brief Contents

Preface xxi Acknowledgments xxv About the Author xxvii

1 Introduction 1

2 Data and Expressions 38

3 Control Structures 79

4 Lists 125

5 Functions 168

6 Objects and Their Use 206

7 Modular Design 247

8 Text Files	289
9 Dictionaries and Sets	337
10 Object-Oriented Programming	383
11 Recursion	460
12 Computing and Its Developments	491
Appendix	525
Index	569

vii

Contents

Preface	xxi
Acknowledgments	xxv
About the Author	xxvii

1 Introduction	1
----------------	---

MOTIVATION 2

FUNDAMENTALS 2

1.1 What Is Computer Science? 2

1.1.1 The Essence of Computational Problem Solving 3

1.1.2 Limits of Computational Problem Solving 5

Self-Test Questions 6

1.2 Computer Algorithms 6

1.2.1 What Is an Algorithm? 6

1.2.2 Algorithms and Computers: A Perfect Match 7 Self-Test Questions 8

1.3 Computer Hardware 9

1.3.1 Digital Computing: It's All about Switches 9

1.3.2 The Binary Number System 10

1.3.3 Fundamental Hardware Components 11

1.3.4 Operating Systems—Bridging Software and Hardware 11

1.3.5 Limits of Integrated Circuits Technology: Moore's Law 12 Self-Test Questions 13

1.4 Computer Software 14

1.4.1 What Is Computer Software? 14

1.4.2 Syntax, Semantics, and Program Translation 14

1.4.3 Procedural vs. Object-Oriented Programming 17 Self-Test Questions 17

COMPUTATIONAL PROBLEM SOLVING 17

1.5 The Process of Computational Problem Solving	17
1.5.1 Problem Analysis	18
1.5.2 Program Design	19
1.5.3 Program Implementation	21
1.5.4 Program Testing	21

ix x Contents

1.6 The Python Programming Language	22
1.6.1 About Python	22
1.6.2 The IDLE Python Development Environment	22
1.6.3 The Python Standard Library	23
1.6.4 A Bit of Python	24
1.6.5 Learning How to Use IDLE	26
1.7 A First Program—Calculating the Drake Equation	29
1.7.1 The Problem	30
1.7.2 Problem Analysis	30
1.7.3 Program Design	30
1.7.4 Program Implementation	30
1.7.5 Program Testing	32

Chapter Summary	33
Chapter Exercises	34
Python Programming Exercises	36
Program Modification Problems	37
Program Development Problems	37

2 Data and Expressions 38

MOTIVATION	39
FUNDAMENTAL CONCEPTS	40
2.1 Literals	40

2.1.1 What Is a Literal?	40
2.1.2 Numeric Literals	40
2.1.3 String Literals	44
2.1.4 Control Characters	46
2.1.5 String Formatting	47
2.1.6 Implicit and Explicit Line Joining	48
2.1.7 Let's Apply It—"Hello World Unicode Encoding"	48
Self-Test Questions	

49

2.2 Variables and Identifiers 50

2.2.1 What Is a Variable? 50

2.2.2 Variable Assignment and Keyboard Input 52

2.2.3 What Is an Identifier? 53

2.2.4 Keywords and Other Predefined Identifiers in Python 54

2.2.5 Let's Apply It—"Restaurant Tab Calculation" 55 Self-Test Questions 56

2.3 Operators 57

2.3.1 What Is an Operator? 57

2.3.2 Arithmetic Operators 57

2.3.3 Let's Apply It—"Your Place in the Universe" 59 Self-Test Questions 60

2.4 Expressions and Data Types 61

2.4.1 What Is an Expression? 61

2.4.2 Operator Precedence 61

2.4.3 Operator Associativity 63

Contents xi

2.4.4 What Is a Data Type? 64

2.4.5 Mixed-Type Expressions 64

2.4.6 Let's Apply It—"Temperature Conversion Program" 65 Self-Test Questions 66

COMPUTATIONAL PROBLEM SOLVING 67

2.5 Age in Seconds Program 67

2.5.1 The Problem 67

2.5.2 Problem Analysis 67

2.5.3 Program Design 67

2.5.4 Program Implementation and Testing 69

Chapter Summary 74

Chapter Exercises 74

Python Programming Exercises 76

Program Modification Problems 76

Program Development Problems 77

3 Control Structures 79

MOTIVATION 80

FUNDAMENTAL CONCEPTS 80

3.1 What Is a Control Structure? 80

3.2 Boolean Expressions (Conditions) 81

3.2.1 Relational Operators 81

3.2.2 Membership Operators 82

3.2.3 Boolean Operators 83

3.2.4 Operator Precedence and Boolean Expressions 85

3.2.5 Short-Circuit (Lazy) Evaluation 86

3.2.6 Logically Equivalent Boolean Expressions 87 Self-Test Questions 88

3.3 Selection Control 89

3.3.1 If Statement 89

3.3.2 Indentation in Python 90

3.3.3 Multi-Way Selection 91

3.3.4 Let's Apply It—Number of Days in Month Program 94 Self-Test Questions 96

3.4 Iterative Control 96

3.4.1 While Statement 97

3.4.2 Input Error Checking 98

3.4.3 Infinite loops 99

3.4.4 Definite vs. Indefinite Loops 100

3.4.5 Boolean Flags and Indefinite Loops 100

3.4.6 Let's Apply It—Coin Change Exercise Program 101 Self-Test Questions 104

COMPUTATIONAL PROBLEM SOLVING 104

3.5 Calendar Month Program 104

3.5.1 The Problem 104

xii Contents

3.5.2 Problem Analysis 104

3.5.3 Program Design 105

3.5.4 Program Implementation and Testing 107 Chapter Summary 117

Chapter Exercises 118

Python Programming Exercises 120

Program Modification Problems 121

Program Development Problems 123

4 Lists 125

MOTIVATION	126
FUNDAMENTAL CONCEPTS	127
4.1 List Structures	127
4.1.1 What Is a List?	127
4.1.2 Common List Operations	127
4.1.3 List Traversal	128
Self-Test Questions	129
4.2 Lists (Sequences) in Python	130
4.2.1 Python List Type	130
4.2.2 Tuples	131
4.2.3 Sequences	132
4.2.4 Nested Lists	134
4.2.5 Let's Apply It—A Chinese Zodiac Program	135
Self-Test Questions	137
4.3 Iterating Over Lists (Sequences) in Python	137
4.3.1 For Loops	137
4.3.2 The Built-in range Function	138
4.3.3 Iterating Over List Elements vs. List Index Values	139
4.3.4 While Loops and Lists (Sequences)	140
4.3.5 Let's Apply It—Password Encryption/Decryption Program	141
Self-Test Questions	144
4.4 More on Python Lists	144
4.4.1 Assigning and Copying Lists	144
4.4.2 List Comprehensions	146
COMPUTATIONAL PROBLEM SOLVING	147
4.5 Calendar Year Program	147
4.5.1 The Problem	147
4.5.2 Problem Analysis	147
4.5.3 Program Design	148
4.5.4 Program Implementation and Testing	149
Chapter Summary	161
Chapter Exercises	162
Python Programming Exercises	164
Program Modification Problems	164

5 Functions 168

MOTIVATION 169

FUNDAMENTAL CONCEPTS 169

5.1 Program Routines 169

5.1.1 What Is a Function Routine? 169

5.1.2 Defining Functions 170

5.1.3 Let's Apply It—Temperature Conversion Program (Function Version) 173

Self-Test Questions 175

5.2 More on Functions 176

5.2.1 Calling Value-Returning Functions 176

5.2.2 Calling Non-Value-Returning Functions 177

5.2.3 Parameter Passing 178

5.2.4 Keyword Arguments in Python 181

5.2.5 Default Arguments in Python 183

5.2.6 Variable Scope 183

5.2.7 Let's Apply It—GPA Calculation Program 186

Self-Test Questions 189

COMPUTATIONAL PROBLEM SOLVING 189

5.3 Credit Card Calculation Program 189

5.3.1 The Problem 189

5.3.2 Problem Analysis 190

5.3.3 Program Design 190

5.3.4 Program Implementation and Testing 191

Chapter Summary 202

Chapter Exercises 202

Python Programming Exercises 203

Program Modification Problems 204

Program Development Problems 204

6 Objects and Their Use 206

MOTIVATION 207

FUNDAMENTAL CONCEPTS 207

6.1 Software Objects 207

6.1.1 What Is an Object? 208

6.1.2 Object References	209
Self-Test Questions	216
6.2 Turtle Graphics	216
6.2.1 Creating a Turtle Graphics Window	216
6.2.2 The “Default” Turtle	218
6.2.3 Fundamental Turtle Attributes and Behavior	219
6.2.4 Additional Turtle Attributes	222
6.2.5 Creating Multiple Turtles	225
6.2.6 Let’s Apply It—Bouncing Balls Program	226
Self-Test Questions	229
COMPUTATIONAL PROBLEM SOLVING	229
6.3 Horse Race Simulation Program	229
6.3.1 The Problem	230
6.3.2 Problem Analysis	230
6.3.3 Program Design	231
6.3.4 Program Implementation and Testing	231
Chapter Summary	243
Chapter Exercises	243
Python Programming Exercises	244
Program Modification Problems	245
Program Development Problems	246
7 Modular Design	247
MOTIVATION	248
FUNDAMENTAL CONCEPTS	248
7.1 Modules	248
7.1.1 What Is a Module?	248
7.1.2 Module Specification	249
Self-Test Questions	251
7.2 Top-Down Design	251
7.2.1 Developing a Modular Design of the Calendar Year Program	251
7.2.2 Specification of the Calendar Year Program Modules	252
Self-Test Questions	255
7.3 Python Modules	255
7.3.1 What Is a Python Module?	255
7.3.2 Modules and Namespaces	256
7.3.3 Importing Modules	257

7.3.4 Module Loading and Execution	260
7.3.5 Local, Global, and Built-in Namespaces in Python	262
7.3.6 A Programmer-Defined Stack Module	264
7.3.7 Let's Apply It—A Palindrome Checker Program	267
Self-Test Questions	268

COMPUTATIONAL PROBLEM SOLVING 269

7.4 Calendar Year Program (function version)	269
7.4.1 The Problem	269
7.4.2 Problem Analysis	269
7.4.3 Program Design	269
7.4.4 Program Implementation and Testing	269
Chapter Summary	284
Chapter Exercises	284
Python Programming Exercises	286
Program Modification Problems	287

Contents xv

8 Text Files 289

MOTIVATION 290

FUNDAMENTAL CONCEPTS 290

8.1 What Is a Text File? 290

8.2 Using Text Files 291

8.2.1 Opening Text Files 291

8.2.2 Reading Text Files 293

8.2.3 Writing Text Files 294

Self-Test Questions 295

8.3 String Processing 296

8.3.1 String Traversal 296

8.3.2 String-Applicable Sequence Operations 296

8.3.3 String Methods 297

8.3.4 Let's Apply It—Sparse Text Program 300

Self-Test Questions 303

8.4 Exception Handling 303

8.4.1 What Is an Exception? 303

8.4.2 The Propagation of Raised Exceptions 304

8.4.3 Catching and Handling Exceptions 305

- 8.4.4 Exception Handling and User Input 307
- 8.4.5 Exception Handling and File Processing 309
- 8.4.6 Let's Apply It—Word Frequency Count Program 310 Self-Test Questions 314

COMPUTATIONAL PROBLEM SOLVING 314

- 8.5 Cigarette Use/Lung Cancer Correlation Program 314
 - 8.5.1 The Problem 315
 - 8.5.2 Problem Analysis 315
 - 8.5.3 Program Design 316
 - 8.5.4 Program Implementation and Testing 318
 - 8.5.5 Determining the Correlation Between Smoking and Lung Cancer 331
- Chapter Summary 331
- Chapter Exercises 332
- Python Programming Exercises 333
- Program Modification Problems 333
- Program Development Problems 334

9 Dictionaries and Sets 337

MOTIVATION 338

FUNDAMENTAL CONCEPTS 338

9.1 Dictionary Type in Python 338

9.1.1 What Is a Dictionary? 339

9.1.2 Let's Apply It—Phone Number Spelling Program 342 Self-Test Questions 346

9.2 Set Data Type 346

9.2.1 The Set Data Type in Python 346

9.2.2 Let's Apply It—Kitchen Tile Visualization Program 348 Self-Test Questions 356

COMPUTATIONAL PROBLEM SOLVING 356

9.3 A Food Co-op's Worker Scheduling Simulation 356

9.3.1 The Problem 357

9.3.2 Problem Analysis 357

9.3.3 Program Design 358

9.3.4 Program Implementation and Testing 360

9.3.5 Analyzing a Scheduled vs. Unscheduled Co-op Worker Approach 375

Chapter Summary 379

Chapter Exercises 379

Python Programming Exercises 380

Program Modification Problems 380

Program Development Problems 381

10 Object-Oriented Programming 383

MOTIVATION 384

FUNDAMENTAL CONCEPTS 384

10.1 What Is Object-Oriented Programming? 384

10.1.1 What Is a Class? 385

10.1.2 Three Fundamental Features of Object-Oriented Programming 385

10.2 Encapsulation 386

10.2.1 What Is Encapsulation? 386

10.2.2 Defining Classes in Python 387

10.2.3 Let's Apply It—A Recipe Conversion Program 394 Self-Test Questions 399

10.3 Inheritance 400

10.3.1 What Is Inheritance? 400

10.3.2 Subtypes 401

10.3.3 Defining Subclasses in Python 402

10.3.4 Let's Apply It—A Mixed Fraction Class 407

Self-Test Questions 411

10.4 Polymorphism 411

10.4.1 What Is Polymorphism? 411

10.4.2 The Use of Polymorphism 414

Self-Test Questions 417

10.5 Object-Oriented Design Using UML 417

10.5.1 What Is UML? 417

10.5.2 UML Class Diagrams 418

Self-Test Questions 422

COMPUTATIONAL PROBLEM SOLVING 423

10.6 Vehicle Rental Agency Program 423

10.6.1 The Problem 423

Contents xvii

10.6.2 Problem Analysis 423

10.6.3 Program Design 423

10.6.4 Program Implementation and Testing 429 Chapter Summary 453

Chapter Exercises 454

Python Programming Exercises 455

Program Modification Problems 456

Program Development Problems 457

11 Recursion 460

MOTIVATION 461

FUNDAMENTAL CONCEPTS 461

11.1 Recursive Functions 461

11.1.1 What Is a Recursive Function? 461

11.1.2 The Factorial Function 464

11.1.3 Let's Apply It—Fractals (Sierpinski Triangle) 467 Self-Test Questions 471

11.2 Recursive Problem Solving 472

11.2.1 Thinking Recursively 472

11.2.2 MergeSort Recursive Algorithm 472

11.2.3 Let's Apply It—MergeSort Implementation 474 Self-Test Questions 476

11.3 Iteration vs. Recursion 476

COMPUTATIONAL PROBLEM SOLVING 477

11.4 Towers of Hanoi 477

11.4.1 The Problem 477

11.4.2 Problem Analysis 477

11.4.3 Program Design and Implementation 481

Chapter Summary 487

Chapter Exercises 487

Python Programming Exercises 488

Program Modification Problems 489

Program Development Problems 490

12 Computing and Its Developments 491

CONTRIBUTIONS TO THE MODERN COMPUTER 492

12.1 The Concept of a Programmable Computer 492

12.1.1 “Father of the Modern Computer”—Charles Babbage (1800s) 492 12.1.2

“The First Computer Programmer”—Ada Lovelace (1800s) 493

12.2 Developments Leading to Electronic Computing 493

12.2.1 The Development of Boolean Algebra (mid-1800s) 493 12.2.2 The

Development of the Vacuum Tube (1883) 494 12.2.3 The Development of

Digital Electronic Logic Gates (1903) 494

xviii Contents

12.2.4 The Development of Memory Electronic Circuits (1919) 495

12.2.5 The Development of Electronic Digital Logic Circuits (1937) 495

12.2.6 “The Father of Information Theory”—Claude Shannon (1948) 496

FIRST-GENERATION COMPUTERS (1940s–mid-1950s) 496

12.3 The Early Groundbreakers 496

12.3.1 The Z3—The First Programmable Computer (1941) 496

12.3.2 The Mark I—First Computer Project in the United States (1937–1943)

497 12.3.3 The ABC—The First Fully Electronic Computing Device (1942) 498

12.3.4 Colossus—A Special-Purpose Electronic Computer (1943) 499 12.3.5

ENIAC—The First Fully Electronic Programmable Computer 500 12.3.6

EDVAC/ACE—The First Stored Program Computers (1950) 501 12.3.7

Whirlwind—The First Real-Time Computer (1951) 502

12.4 The First Commercially Available Computers 503

12.4.1 The Struggles of the Eckert-Mauchly Computer Corporation (1950) 503

12.4.2 The LEO Computer of the J. Lyons and Company (1951) 504

SECOND-GENERATION COMPUTERS (mid-1950s to mid-1960s) 505 12.5

Transistorized Computers 505

12.5.1 The Development of the Transistor (1947) 505

12.5.2 The First Transistor Computer (1953) 506

12.6 The Development of High-Level Programming Languages 506

12.6.1 The Development of Assembly Language (early 1950s) 506 12.6.2 The

First High-Level Programming Languages (mid-1950s) 507 12.6.3 The First

“Program Bug” (1947) 508

THIRD-GENERATION COMPUTERS (mid-1960s to early 1970s) 508 12.7

The Development of the Integrated Circuit (1958) 508

12.7.1 The Catalyst for Integrated Circuit Advancements (1960s)	509	12.7.2 The Development of the Microprocessor (1971)	511
12.8 Mainframes, Minicomputers, and Supercomputers	512		
12.8.1 The Establishment of the Mainframe Computer (1962)	512	12.8.2 The Development of the Minicomputer (1963)	513
12.8.3 The Development of the UNIX Operating System (1969)	513	12.8.4 The Development of Graphical User Interfaces (early 1960s)	514
12.8.5 The Development of the Supercomputer (1972)	515		

FOURTH-GENERATION COMPUTERS (early 1970s to the Present)	515	12.9 The Rise of the Microprocessor	515
12.9.1 The First Commercially Available Microprocessor (1971)	515	12.9.2 The First Commercially Available Microcomputer Kit (1975)	516
12.10 The Dawn of Personal Computing	516		
12.10.1 The Beginnings of Microsoft (1975)	516		
12.10.2 The Apple II (1977)	517		
12.10.3 IBM's Entry into the Microcomputer Market (1981)	517	12.10.4 Society Embraces the Personal Computer (1983)	518
12.10.5 The Development of Graphical User Interfaces (GUIs)	518	12.10.6 The Development of the C11 Programming Language	519

Contents xix

THE DEVELOPMENT OF COMPUTER NETWORKS	520
12.11 The Development of Wide Area Networks	520
12.11.1 The Idea of Packet-Switched Networks (early 1960s)	520
12.11.2 The First Packet-Switched Network: ARPANET (1969)	520
12.12 The Development of Local Area Networks (LANs)	521
12.12.1 The Need for Local Area Networks	521
12.12.2 The Development of Ethernet (1980)	521
12.13 The Development of the Internet and World Wide Web	522
12.13.1 The Realization of the Need for "Internetworking"	522
12.13.2 The Development of the TCP/IP Internetworking Protocol (1973)	522
12.13.3 The Development of the World Wide Web (1990)	522
12.13.4 The Development of the Java Programming Language (1995)	523

Appendix 525 Index 569

This page is intentionally left blank

Preface

Book Concept

This text introduces students to programming and computational problem solving using the Python 3 programming language. It is intended primarily for a first-semester computer science (CS1) course, but is also appropriate for use in any course providing an introduction to computer programming and/or computational problem solving. The book provides a step-by-step, “hands on” pedagogical approach which, together with Python’s clear and simple syntax, makes this book easy to teach and learn from.

The primary goal in the development of this text was to create a pedagogically sound and accessible textbook that emphasizes fundamental programming and computational problem- solving concepts over the minutiae of a particular programming language. Python’s ease in the creation and use of both indexed and associative data structures (in the form of lists/tuples and dictionaries), as well as sets, allows for programming concepts to be demonstrated without the need for detailed discussion of programming language specifics.

Taking advantage of Python’s support of both the imperative (i.e., procedural) and objectoriented paradigms, a “back to basics,” “objects-late” approach is taken to computer programming. It follows the belief that solid grounding in imperative programming should precede the larger number of (and more abstract) concepts of the object-oriented paradigm. Therefore, objects are not covered until Chapter 5, and object-oriented programming is not introduced until Chapter 10. For those who do not wish to introduce object-oriented programming, Chapter 10 can easily be skipped.

How This Book Is Different

This text has a number of unique pedagogical features including:

- ◆ *A short motivation section* at the beginning of each chapter which provides a larger perspective on the chapter material to be covered.
- ◆ *Hands-on exercises* throughout each chapter which take advantage of the interactive capabilities of Python.
- ◆ A fully-developed computational problem solving example at the end of each chapter that places an emphasis on *program testing and program debugging*.

♦ A richly illustrated, final chapter on “*Computing and Its Developments*” that provides a storyline of notable individuals, accomplishments, and developments in computing, from Charles Babbage through modern times.

♦ A *Python 3 Programmers’ Reference* in the back of the text which allows the book to serve as both a pedagogical resource *and* as a convenient Python reference.

xxi xxii Preface

Pedagogical Features

The book takes a step-by-step pedagogical approach. Each new concept is immediately followed by a pedagogical element that elucidates the material covered and/or challenges students’ understanding. The specific pedagogical features of the book are listed below.

Summary Boxes

At the end of each subsection, a clearly outlined summary box is provided containing the most salient information of the material just presented. These concise summaries also serve as useful reference points as students review the chapter material.

Let’s Try It Sections

Also at the end of each subsection, a short Let’s Try It section is given in which students are asked to type in Python code (into the Python shell) and observe the results. These “hands on” exercises help to immediately reinforce material as students progress through the chapter.

Let’s Apply It Sections

At the end of each major section, a complete program example is provided with detailed line-by-line discussion. These examples serve to demonstrate the programming concepts just learned in the context of an actual program.

Self-Test Questions

Also at the end of each major section, a set of multiple-choice/short-answer questions is given. The answers are included so that students may perform a comprehension self check in answering these.

A variety of exercises and program assignments are also included at the end of every chapter. These are designed to gradually ease students from review of general concepts, to code-writing exercises, to modification of significant-sized

programs, to developing their own programs, as outlined below.

Chapter Exercises

At the end of each chapter, a set of simple, short-answer questions are provided.

Python Programming Exercises

Also at the end of each chapter, a set of simple, short Python programming exercises are given.

Program Modification Problems

Additionally, at the end of each chapter is a set of programming problems in which students are asked to make various modifications to program examples in the chapter. Thus, these exercises do not require students to develop a program from scratch. They also serve as a means to encourage students to think through the chapter program examples.

Program Development Problems

Finally, at the end of each chapter is a set of computational problems which students are to develop programs for from scratch. These problems are generally similar to the program examples given in the chapters.

Emphasis on Computational Problem Solving

The capstone programs at the end of each chapter show students how to go through the process of computational problem solving. This includes problem analysis, design, implementation, and testing, as outlined in Chapter 1. As a program is developed and tested, errors are intentionally placed in the code, leading to discussion and demonstration of program testing and debugging. Programming errors, therefore, are presented as a normal part of software development. This helps students develop

Preface xxiii

their own program debugging skills, and reinforces the idea that program debugging is an inevitable part of program development—an area of coverage that is crucial for beginning programmers, and yet often lacking in introductory computer science books.

Given the rigor with which these problems are presented, the sections are somewhat lengthy. Since the capstones do not introduce any new concepts, they may be skipped if the instructor does not have time to cover them.

Guided Book Tour

Chapter 1 addresses the question “What is computer science?” Computational problem solving is introduced, including discussions on the limits of computation, computer algorithms, computer hardware, computer software, and a brief introduction to programming in Python. The end of the chapter lays out a step-by-step computational problem solving process for students consisting of problem analysis, program design, program implementation, and program testing.

Chapter 2 covers data and expressions, including arithmetic operators, discussion of limits of precision, output formatting, character encoding schemes, control characters, keyboard input, operator precedence and associativity, data types, and coercion vs. type conversion.

Chapter 3 introduces control structures, including relational, membership and Boolean operators, short-circuit (lazy) evaluation, selection control (if statements) and indentation in Python, iterative control (while statements), and input error checking. (For statements are not covered until Chapter 4 on Lists.) Break and continue statements are not introduced in this book. It is felt that these statements, which violate the principles of structured programming, are best not introduced to the beginning programmer.

Chapter 4 presents lists and for statements. The chapter opens with a general discussion of lists and list operations. This is followed by lists and tuples in Python, including nested lists, for loops, the built-in range function, and list comprehensions. Since all values in Python are (object) references, and lists are the first mutable type to which students are introduced, a discussion of shallow vs. deep copying is provided without explicit mention of references. The details of object representation in Python is covered in Chapter 6.

Chapter 5 introduces the notion of a program routine, including discussions of parameter passing (actual argument vs. formal parameters), value vs. non-value returning functions, mutable vs. immutable arguments, keyword and default arguments in Python, and local vs. global scope.

Chapter 6 introduces students to the concept of objects in programming. Here students see how objects are represented as references, and therefore are able to fully understand the behavior of assignment and copying of lists (initially

introduced in Chapter 4), as well as other types. Turtle graphics is introduced (by use of the turtle module of the Python Standard Library) and is used to provide an intuitive, visual means of understanding the concept of object instances. This also allows students to write fun, graphical programs, while at the same time reinforcing the notion and behavior of objects.

Chapter 7 covers modules and modular design. It starts off by explaining the general notion of a module and module specification, including docstrings in Python. It is followed by a discussion of topdown design. It then introduces modules in Python, including namespaces, the importing of modules, module private variables, module loading and execution, and local, global, and built-in namespaces. The notion of a stack is introduced here via the development of a programmer-defined stack module.

xxiv Preface

Chapter 8 introduces text files and string processing. It starts with how to open/close, and read/ write files in Python. Then the string-applicable sequence operations from Chapter 4 are revisited, and additional string methods are covered. Exception handling is introduced in the context of file handling, and some of the more commonly occurring Python Standard Exceptions are introduced.

Chapter 9 presents dictionaries (Python’s associative data structure) and sets.

Chapter 10 introduces object-oriented programming. It begins with a discussion of classes, and the notion of encapsulation. Then, how classes are defined is presented, including discussion of special methods in Python. Inheritance and subtypes are discussed next, followed by a discussion of the use of polymorphism. Finally, the chapter ends with a brief introduction to class diagrams in UML.

Chapter 11 covers recursion and recursive problem solving, including discussion of recursion vs. iteration, and when recursion is appropriately used.

Chapter 12 concludes the book by providing an overview of the people, achievements and developments in computing. This chapter serves to “humanize” the field and educate students on the history of the discipline.

Online Textbook Supplements

All supplements are available via the book's companion website at www.wiley.com/college/dierbach. Below is the list of supplements that accompany this text:

- ◆ Instructor's manual, with answers to all exercises and program assignments
- ◆ PowerPoint slides, summarizing the key points of each chapter
- ◆ Program code for all programs in the book
- ◆ Test bank of exam questions for each chapter

A separate student companion site is available at the above web site which grants students access to the program code and additional files needed to execute and/or modify programs in the book. All other program code is available to instructors only.

Acknowledgments

I would first like to thank the people at Wiley & Sons. To Dan Sayre, for getting this project going; to my editor Beth Golub, for all her patience and guidance during the evolution of the book; and to Samantha Mandel, Assistant Editor, for her invaluable help. I would also like to thank Harry Nolan, Design Director, who took the time to ensure that the book design turned out as I envisioned; to Jolene Ling, Production Editor, who so graciously worked on the production of the book and the ensuing changes, and for seeing that everything came together.

There are many others to thank who have in some way contributed to this project. First, thanks to Harry Hochheiser, for all the motivating and informative discussions that eventually led me to Python, and ultimately the development of this book. Many thanks to my colleague Josh Dehlinger, who lent his extremely critical eye to the project (and took up the slack on many of my department duties!). Thanks to my department chair, Chao Lu, for his support, friendship, and for creating such a collegial and productive environment to work (and for funneling some of my duties to Josh!). And thanks to Shiva Azadegan, who first planted the idea of writing a book in my head, and for being such a supportive friend, as well as a wonderful colleague.

I would also like to acknowledge a couple of my outstanding graduate TAs in the Python course for all their help and enthusiasm on the project. I thank Crystal McKinney, for so freely offering her time to review chapters and offer her suggestions. I owe a great debt of thanks to Leela Sedaghat, who contributed to

the project in so many ways—her insightful review of chapters, the enormous amount of time spent on verifying and obtaining image permissions, and her design of and contribution to the Python Programmers’ Reference manual, which without her help, would never have been completed on time. Previous graduate students Ahbi Grover and Lanlan Wang also read earlier drafts of the book.

Finally, I thank the reviewers. Without them, the book could never be what it is now. First, special thanks to Claude Anderson of Rose-Hulman Institute of Technology. His meticulous review for technical errors, and his suggestions on pedagogy, have significantly contributed to the book. In addition, I thank each of the following individuals who served as reviewers on this project: James Atlas, University of Delaware; Richard Borie, University of Alabama; Tim Bower, Kansas State University Salina; Darin Brezeale, University of Texas at Arlington; Diana Cukierman, Simon Fraser University; Chris Heiden, St. Clair County Community College; Ric Heishman, George Mason University; Jennifer Kay, Rowan University; Debby Keen, University of Kentucky; Clayton Lewis, University of Colorado; Alan McLeod, Queen’s University at Kingston; Ethan Miller, University of California, Santa Cruz; Joe Oldham, Centre College; Susan Mary Rosselet, Bemidji State University; Terry A. Scott, University of Northern Colorado; and Leon Tietz, Minnesota State University Mankato.

xxv

About the Author

Charles Dierbach is an Associate Professor of computer science at Towson University, and has regularly taught introductory undergraduate computer science courses for the past thirty-five years. He received his Ph.D. in Computer Science from the University of Delaware. While a lecturer there, he received the Outstanding Teaching Award from the undergraduate chapter of the ACM. At Towson, he served as Director of the Undergraduate Computer Science program for over ten years. In addition to teaching introductory computer science courses, Dr. Dierbach also teaches undergraduate and graduate courses in object-oriented design and programming.

xxvii

Introduction CHAPTER 1

This chapter addresses the question “What is computer science?” We begin by introducing the essence of computational problem solving via some classic examples. Next, computer algorithms, the heart of computational problem solving, are discussed. This is followed by a look at computer hardware (and the related issues of binary representation and operating systems) and computer software (and the related issues of syntax, semantics, and program translation). The chapter finishes by presenting the process of computational problem solving, with an introduction to the Python programming language.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦

Explain the essence of computational problem solving ♦ Explain what a computer algorithm is

♦ Explain the fundamental components of digital hardware ♦ Explain the role of binary representation in digital computing ♦ Explain what an operating systems is

♦ Explain the fundamental concepts of computer software ♦ Explain the fundamental features of IDLE in Python

♦ Modify and execute a simple Python program

CHAPTER CONTENTS

Motivation

Fundamentals

1.1 What Is Computer Science?

1.2 Computer Algorithms

1.3 Computer Hardware

1.4 Computer Software

Computational Problem Solving

1.5 The Process of Computational Problem Solving 1.6 The Python

Programming Language

1.7 A First Program—Calculating the Drake Equation 1

MOTIVATION

Computing technology has changed, and is continuing to change the world. Essentially every aspect of life has been impacted by computing. Just-in-time inventory allows companies to significantly reduce costs. Universal digital medical records promise to save the lives of many of the estimated 100,000 people who die each year from medical errors. Vast information resources, such

as Wikipedia, now provide easy, quick access to a breadth of knowledge as never before. Information sharing via Facebook and Twitter has not only brought family and friends together in new ways, but has also helped spur political change around the world. New interdisciplinary fields combining computing and science will lead to breakthroughs previously unimaginable. Computing-related fields in almost all areas of study are emerging (see Figure 1-1).

In the study of computer science, there are fundamental principles of computation to be learned that will never change. In addition to these principles, of course, there is always changing technology. That is what makes the field of computer science so exciting. There is constant change and advancement, but also a foundation of principles to draw from. What can be done with computation is limited only by our imagination. With that said, we begin our journey into the world of computing. I have found it an unending fascination—I hope that you do too. Bon voyage!



Various Computational-Related Fields		
Computational Biology	Computational Medicine	Computational Journalism
Computational Chemistry	Computational Pharmacology	Digital Humanities
Computational Physics	Computational Economics	Computational Creativity
Computational Mathematics	Computational Textiles	Computational Music
Computational Materials Science	Computational Architecture	Computational Photography
Computer-Aided Design	Computational Social Science	Computational Advertising
Computer-Aided Manufacturing	Computational Psychology	Computational Intelligence

FIGURE 1-1 Computing-Related Specialized Fields
FUNDAMENTALS

1.1 What Is Computer Science?

Many people, if asked to define the field of computer science, would likely say that it is about programming computers. Although programming is certainly a primary activity of computer science, programming languages and computers are only *tools*. What computer science is fundamentally about is **computational problem solving**—that is, solving problems by the use of computation (Figure 1-2).

This description of computer science provides a succinct definition of the field. However, it does not convey its tremendous *breadth and diversity*. There are various areas of study in computer science including software engineering (the design and implementation of large software systems), database management, computer networks, computer graphics, computer simulation, data mining, information security, programming language design, systems programming, computer architecture, human–computer interaction, robotics, and artificial intelligence, among others.

The definition of computer science as computational problem solving begs the question: *What is computation?* One characterization of computation is given by the notion of an *algorithm*. The definition of an algorithm is given in section 1.2. For now, consider an algorithm to be a series of steps that can be systematically followed for producing the answer to a certain type of problem. We look at fundamental issues of computational problem solving next.



FIGURE 1-2 Computational

Problem Solving

Computer science is fundamentally about computational problem solving.

1.1.1 The Essence of Computational Problem Solving

In order to solve a problem computationally, two things are needed: a *representation* that captures all the relevant aspects of the problem, and an *algorithm* that solves the problem by use of the representation. Let's consider a problem known as the **Man, Cabbage, Goat, Wolf problem**(Figure 1-3).

A man lives on the east side of a river. He wishes to bring a cabbage, a goat, and a wolf to a village on the west side of the river to sell. However, his boat is only big enough to hold himself, and either the cabbage, goat, or wolf. In addition, the man cannot leave the goat alone with the cabbage because the goat will eat the cabbage, and he cannot leave the wolf alone with the goat because the wolf will eat the goat. How does the man solve his problem?

There is a simple algorithmic approach for solving this problem by simply trying all possible combinations of items that may be rowed back and forth across the river. Trying all possible solutions to a given problem is referred to as a *brute*

force approach. What would be an appropriate



FIGURE 1-3 Man,

Cabbage, Goat, Wolf Problem

representation for this problem? Since only the relevant aspects of the problem need to be represented, all the irrelevant details can be omitted. A representation that leaves out details of what is being represented is a form of **abstraction**.

The use of abstraction is prevalent in computer science. In this case, is the color of the boat relevant? The width of the river? The name of the man? No, the only relevant information is *where* each item is at each step. The collective location of each item, in this case, refers to the *state* of the problem. Thus, the *start state* of the problem can be represented as follows.

man cabbage goat wolf [E, E, E, E]

In this representation, the symbol E denotes that each corresponding object is on the east side of the river. If the man were to row the goat across with him, for

example, then the representation of the new problem state would be

man cabbage goat wolf [W, E, W, E]

in which the symbol W indicates that the corresponding object is on the west side of the river—in this case, the man and goat. (The locations of the cabbage and wolf are left unchanged.) A solution to this problem is a sequence of steps that converts the initial state,

[E, E, E, E]

in which all objects are on the east side of the river, to the *goal state*,
[W, W, W, W]

in which all objects are on the west side of the river. Each step corresponds to the man rowing a particular object across the river (or the man rowing alone). As you will see, the Python programming language provides an easy means of representing sequences of values. The remaining task is to develop or find an existing algorithm for computationally solving the problem using this representation. The solution to this problem is left as a chapter exercise.

As another example computational problem, suppose that you needed to write a program that displays a calendar month for any given month and year, as shown in Figure 1-4. The representation of this problem is rather straightforward. Only a few values need to be maintained—the month and year, the number of days in each month, the names of the days of the week, and the day of the week that the first day of the month falls on. Most of these values are either provided by the user (such as the month and year) or easily determined (such as the number of days in a given month).

The less obvious part of this problem is how to determine the day of the week that a given date falls on. You would need an algorithm that can compute this. Thus, no matter how well you may know a given programming language or how good a programmer you may be, without such an algo

rithm you could not solve this problem.

MAY 2012						
Sun	Mon	Tues	Wed	Thur	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

FIGURE 1-4 Calendar Month

In order to solve a problem computationally, two things are needed: a *representation* that captures all the relevant aspects of the problem, and an *algorithm* that solves the problem by use of the representation.

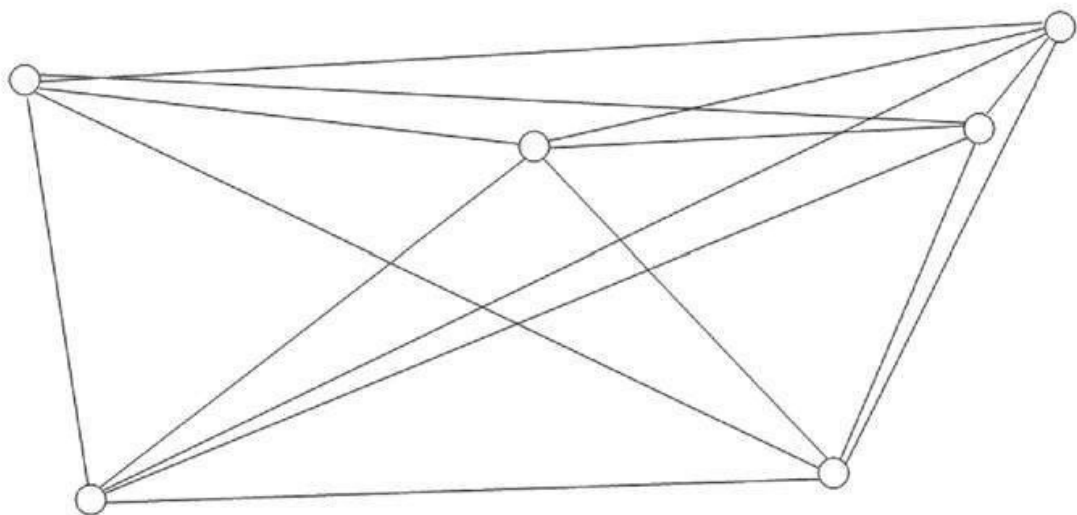
1.1.2 Limits of Computational Problem Solving

Once an algorithm for solving a given problem is developed or found, an important question is, “Can a solution to the problem be found in a reasonable amount of time?” If not, then the particular algorithm is of limited practical use.

San Francisco Boston 2708 mi.

New York Chicago

344 mi.



748 mi. 1749 mi.

Los AngelesAtlanta

FIGURE 1-5 Traveling Salesman Problem

The **Traveling Salesman problem** (Figure 1-5) is a classic computational

problem in computer science. The problem is to find the shortest route of travel for a salesman needing to visit a given set of cities. In a brute force approach, the lengths of all possible routes would be calculated and compared to find the shortest one. For ten cities, the number of possible routes is $10!$ (10 factorial), or over three and a half million (3,628,800). For twenty cities, the number of possible routes is $20!$, or over two and a half quintillion (2,432,902,008,176,640,000). If we assume that a computer could compute the lengths of one million routes per second, it would take over 77,000 years to find the shortest route for twenty cities by this approach. For 50 cities, the number of possible routes is over 10^{64} . In this case, it would take more time to solve than the age of the universe!

A similar problem exists for the game of chess (Figure 1-6). A brute force approach for a chess-playing program would be to “look ahead” to *all* the eventual outcomes of every move that

can be made in deciding each next move. There



FIGURE 1-6 Game of Chess

are approximately 10^{120} possible chess games that can be played. This is related to the average number of look-ahead steps needed for deciding each move. How big is this number? There are approximately 10^{80} atoms in the observable universe, and an estimated 3.3×10^{90} grains of sand to fill the universe solid.

Thus, *there are more possible chess games that can be played than grains of sand to fill the universe solid!* For problems such as this and the Traveling Salesman problem in which a brute-force approach is impractical to use, more efficient problem-solving methods must be discovered that find either an exact or an approximate solution to the problem.

Any algorithm that correctly solves a given problem must solve the problem in a reasonable amount of time, otherwise it is of limited practical use.

Self-Test Questions

1. A good definition of computer science is “the science of programming computers.” (TRUE/FALSE)
2. Which of the following areas of study are included within the field of computer science? **(a)** Software engineering
(b) Database management
(c) Information security
(d) All of the above
3. In order to computationally solve a problem, two things are needed: a representation of the problem, and an _____ that solves it.
4. Leaving out detail in a given representation is a form of _____.
5. A “brute-force” approach for solving a given problem is to:
(a) Try all possible algorithms for solving the problem.
(b) Try all possible solutions for solving the problem.
(c) Try various representations of the problem.
(d) All of the above
6. For which of the following problems is a brute-force approach practical to use? **(a)** Man, Cabbage, Goat, Wolf problem
(b) Traveling Salesman problem
(c) Chess-playing program
(d) All of the above

ANSWERS: 1. False, 2. (d), 3. algorithm, 4. abstraction, 5. (b), 6. (a)

1.2 Computer Algorithms

This section provides a more complete description of an algorithm than given above, as well as an example algorithm for determining the day of the week for a given date.

1.2.1 What Is an Algorithm?

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time (that is, it

1.2 Computer Algorithms 7

eventually terminates). Algorithms solve *general* problems (determining whether any given number is a prime number), and not specific ones (determining whether 30753 is a prime number). Algorithms, therefore, are general computational methods used for solving particular problem instances.

The word “algorithm” is derived from the ninth-century Arab mathematician, Al-Khwarizmi (Figure 1-7), who worked on “written processes to achieve some goal.” (The term “algebra” also derives from the term “al-jabr,” which he introduced.)

Computer algorithms are central to computer science. They provide step-by-step methods of computation that a machine can carry out. Having high-speed machines (computers) that can consistently follow and execute a given set of instructions provides a reliable and effective means of realizing computation. However, *the computation that a given computer performs is only as good as the underlying algorithm used*. Understanding what can be effectively programmed and executed by computers, therefore, relies on the understanding of computer algorithms.



FIGURE 1-7 Al-Khwarizmi

(Ninth Century A.D.)

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time.

1.2.2 Algorithms and Computers: A Perfect Match

Much of what has been learned about algorithms and computation since the beginnings of modern computing in the 1930s–1940s could have been studied centuries ago, since the study of algorithms does not depend on the existence of computers. The algorithm for performing long division is such an example. However, most algorithms are not as simple or practical to apply manually. Most require the use of computers either because they would require too much time for a person to apply, or involve so much detail as to make human error likely. Because *computers can execute instructions very quickly and reliably without error*, algorithms and computers are a perfect match! Figure 1-8 gives an example algorithm for determining the day of the week for any date between

January 1, 1800 and December 31, 2099.

Because computers can execute instructions very quickly and reliably without error, algorithms and computers are a perfect match.

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
if **century_digits** equals 20, then add 6 to **value**.
5. If the **month** is equal to January and **year** is not a leap year,
then add 1 to **value**, else,
if the **month** is equal to February and the **year** is a leap year, then
add 3 to **value**; if not a leap year, then add 4 to **value**, else,
if the **month** is equal to March or November, then add 4 to **value**, else,
if the **month** is equal to May, then add 2 to **value**, else,
if the **month** is equal to June, then add 5 to **value**, else,
if the **month** is equal to August, then add 3 to **value**, else,
if the **month** is equal to October, then add 1 to **value**, else,
if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else
if **value** is equal to 2, day of the week is Monday; else
if **value** is equal to 3, day of the week is Tuesday; else
if **value** is equal to 4, day of the week is Wednesday; else
if **value** is equal to 5, day of the week is Thursday; else
if **value** is equal to 6, day of the week is Friday; else
if **value** is equal to 0, day of the week is Saturday

FIGURE 1-8 Day of the Week Algorithm

Note that there is no value to add for the months of April and July.

Self-Test Questions

1. Which of the following are true of an algorithm?

(a) Has a finite number of steps

(b) Produces a result in a finite amount of time

- (c) Solves a general problem
- (d) All of the above

2. Algorithms were first developed in the 1930–1940s when the first computing machines appeared. (TRUE/FALSE)

3. Algorithms and computers are a “perfect match” because: (Select all that apply.) (a) Computers can execute a large number of instructions very quickly. (b) Computers can execute instructions reliably without error. (c) Computers can determine which algorithms are the best to use for a given problem.

4. Given that the year 2016 is a leap year, what day of the week does April 15th of that year fall on? Use the algorithm in Figure 1-8 for this.

5. Which of the following is an example of an algorithm? (Select all that apply.) (a) A means of sorting any list of numbers (b) Directions for getting from your home to a friend’s house (c) A means of finding the shortest route from your house to a friend’s house.

ANSWERS: 1. (d), 2. False, 3. (a,b) 4. Friday, 5. (a,c)

1.3 Computer Hardware

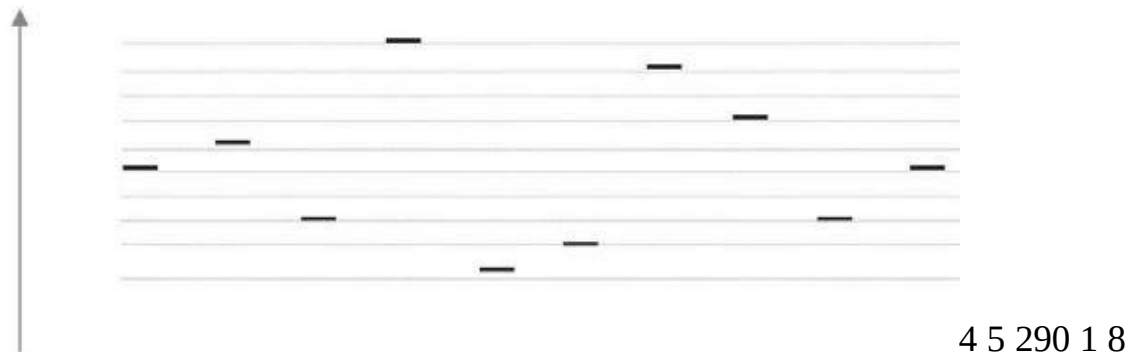
Computer hardware comprises the physical part of a computer system. It includes the all-important components of the *central processing unit* (CPU) and *main memory*. It also includes *peripheral components* such as a keyboard, monitor, mouse, and printer. In this section, computer hardware and the intrinsic use of binary representation in computers is discussed.

1.3.1 Digital Computing: It’s All about Switches

It is essential that computer hardware be reliable and error free. If the hardware gives incorrect results, then any program run on that hardware is unreliable. A rare occurrence of a hardware error was discovered in 1994. The widely used Intel processor was found to give incorrect results only when certain numbers were divided, estimated as likely to occur once every 9 billion divisions. Still, the discovery of the error was very big news, and Intel promised to replace the processor for any one that requested it.

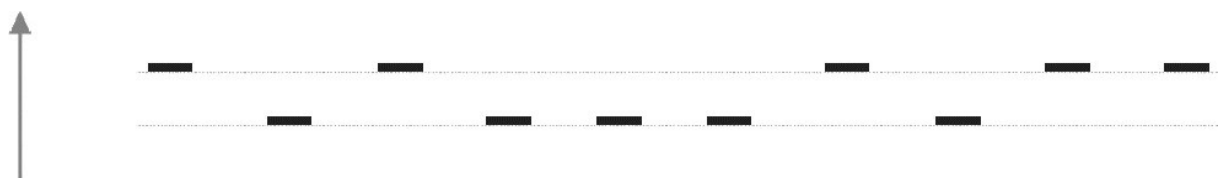
The key to developing reliable systems is to keep the design as simple as

possible. In digital computing, all information is represented as a series of digits. We are used to representing numbers using base 10 with digits 0–9. Consider if information were represented within a computer system this way, as shown in Figure 1-9.



62 4 FIGURE 1-9 Decimal Digitalization

In current electronic computing, each digit is represented by a different voltage level. The more voltage levels (digits) that the hardware must utilize and distinguish, the more complex the hardware design becomes. This results in greater chance of hardware design errors. It is a fact of information theory, however, that any information can be represented using only *two* symbols. Because of this, *all information within a computer system is represented by the use of only two digits, 0 and 1*, called **binary representation**, shown in Figure 1-10.



10 1 0 0 0 1 0 1 1 FIGURE 1-10 Binary Digitalization

In this representation, each digit can be one of only two possible values, similar to a light switch that can be either on or off. Computer hardware, therefore, is based on the use of simple electronic “on/off” switches called **transistors** that switch at very high speed. **Integrated circuits** (“chips”), the building blocks of computer hardware, are comprised of millions or even billions of transistors. The development of the transistor and integrated circuits is discussed in Chapter 12. We discuss binary representation next.

All information within a computer system is represented using only two digits, 0

and 1, called **binary representation**.

1.3.2 The Binary Number System

For representing numbers, any base (radix) can be used. For example, in base 10, there are ten possible digits (0, 1, . . . , 9), in which each column value is a power of ten as shown in Figure 1-11.

10,000,000	1,000,000	100,000	10,000	1,000	100	10	1
10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

9 9 = 99 **FIGURE**

1-11 Base 10 Representation

Other radix systems work in a similar manner. **Base 2** has digits 0 and 1, with place values that are powers of two, as depicted in Figure 1-12.

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

0	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

0 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = 99 **FIGURE**

1-12 Base 2 Representation

As shown in this figure, converting from base 2 to base 10 is simply a matter of adding up the column values that have a 1.

The term **bit** stands for **b**inary **d**igi **t**. Therefore, every bit has the value 0 or 1. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits. Although values represented in base 2 are significantly longer than those represented in base 10, binary representation is used in digital computing because of the resulting simplicity of hardware design.

The algorithm for the conversion from base 10 to base 2 is to successively divide a number by two until the remainder becomes 0. The remainder of each division provides the next higher-order (binary) digit, as shown in Figure 1-13.

99/2	=	49,	with remainder 1
49/2	=	24,	with remainder 1
24/2	=	12,	with remainder 0
12/2	=	6,	with remainder 0
6/2	=	3,	with remainder 0
3/2	=	1,	with remainder 1
1/2	=	0,	with remainder 1

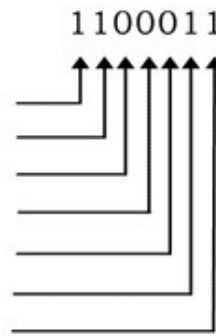


FIGURE 1-13

Converting from Base 10 to Base 2

Thus, we get the binary representation of 99 to be 1100011. This is the same as in Figure 1-12 above, except that we had an extra leading insignificant digit of 0, since we used an eight-bit representation there.

The term **bit** stands for binary digit. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits.

1.3.3 Fundamental Hardware Components

The **central processing unit (CPU)** is the “brain” of a computer system, containing digital logic circuitry able to interpret and execute instructions. **Main memory** is where currently executing programs reside, which the CPU can directly and very quickly access. Main memory is volatile; that is, the contents are lost when the power is turned off. In contrast, **secondary memory** is nonvolatile, and therefore provides long-term storage of programs and data. This kind of storage, for example, can be magnetic (hard drive), optical (CD or DVD), or nonvolatile flash memory (such as in a USB drive). **Input/output devices** include anything that allows for input (such as the mouse and keyboard) or output (such as a monitor or printer). Finally, **buses** transfer data between components within a computer system, such as between the CPU and main memory. The relationship of these devices is depicted in Figure 1-14 below.

The **central processing unit (CPU)** is the “brain” of a computer, containing digital logic circuitry able to interpret and execute instructions.

1.3.4 Operating Systems—Bridging Software and Hardware

An **operating system** is software that has the job of managing and interacting with the hardware resources of a computer. Because an operating system is intrinsic to the operation a computer, it is referred to as **system software**.

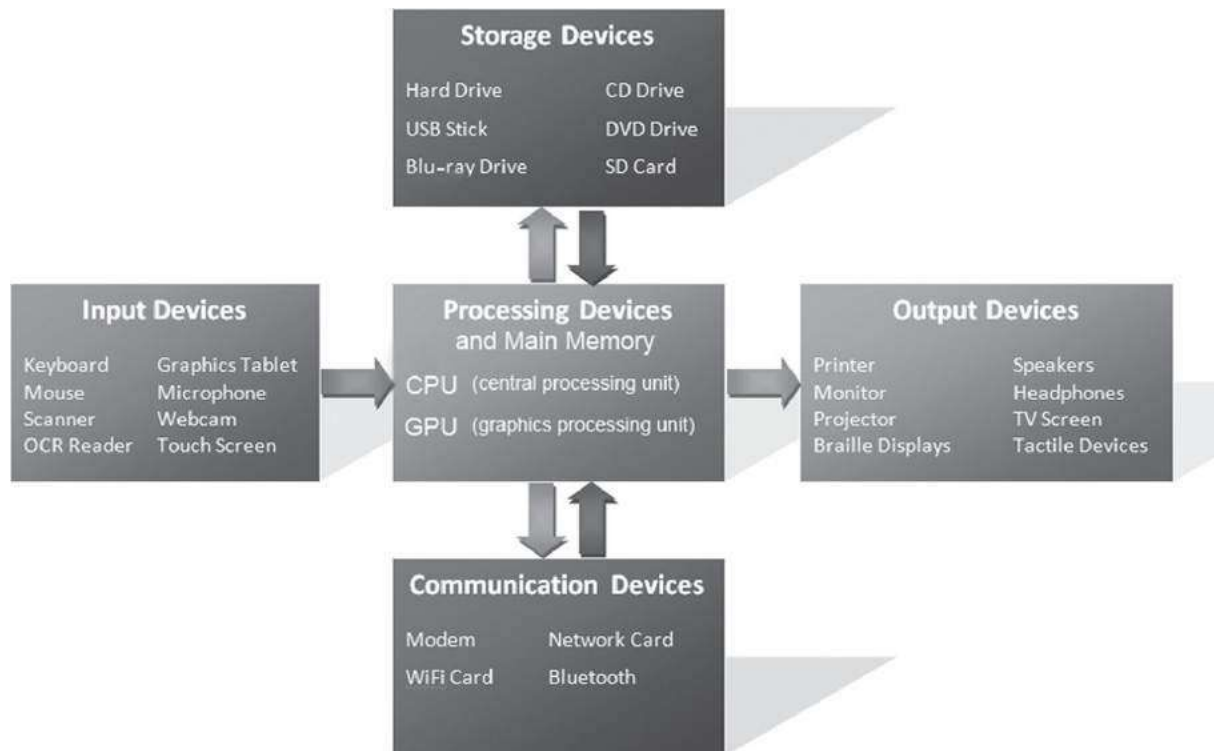


FIGURE 1-14 Fundamental Hardware Components

An operating system acts as the “middle man” between the hardware and executing application programs (see Figure 1-15). For example, it controls the allocation of memory for the various programs that may be executing on a computer. Operating systems also provide a particular user interface. Thus, it is the operating system installed on a given computer that determines the “look and feel” of the user interface and how the user interacts with the system, and not the particular model computer.

An **operating system** is software that has the job of managing the hardware resources of a given computer and providing a particular user interface.

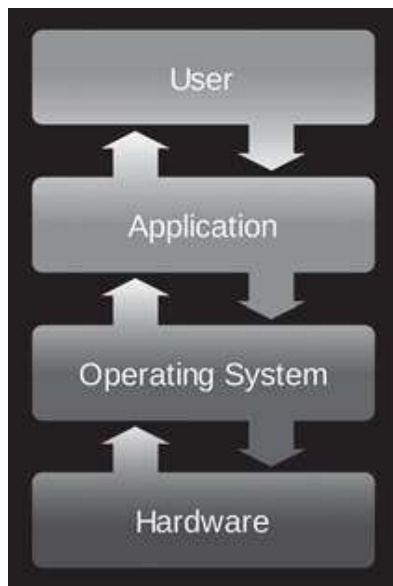


FIGURE 1-15 Operating System

1.3.5 Limits of Integrated Circuits Technology: Moore's Law

In 1965, Gordon E. Moore (Figure 1-16), one of the pioneers in the development of integrated circuits and cofounder of Intel Corporation, predicted that as a result of continuing engineering developments, the number of transistors that would be able to be put on a silicon chip would double roughly every two years, allowing the complexity and therefore the capabilities of integrated circuits to grow exponentially. This prediction became known as **Moore's Law**. Amazingly, to this day that prediction has held true. While this doubling of performance cannot go on indefinitely, it has not yet reached its limit.



FIGURE 1-16 Gordon E. Moore

Moore's Law states that the number of transistors that can be placed on a single silicon chip doubles roughly every two years.

Self-Test Questions

1. All information in a computer system is in binary representation. (TRUE/FALSE)
2. Computer hardware is based on the use of electronic switches called _____.
3. How many of these electronic switches can be placed on a single integrated circuit, or "chip"? (a) Thousands
(b) Millions
(c) Billions
4. The term "bit" stands for _____.
5. A bit is generally a group of eight bytes. (TRUE/FALSE)
6. What is the value of the binary representation 0110.
(a) 12
(b) 3
(c) 6
7. The _____ interprets and executes instructions in a computer system.
8. An operating system manages the hardware resources of a computer

system, as well as provides a particular user interface. (TRUE/FALSE)

9. Moore's Law predicts that the number of transistors that can fit on a chip doubles about every ten years. (TRUE/FALSE)

ANSWERS: 1. True, 2. transistors, 3. (c), 4. binary digit, 5. False, 6. (c), 7. CPU, 8. True, 9. False

1.4 Computer Software

The first computer programs ever written were for a mechanical computer designed by Charles Babbage in the mid-1800s. (Babbage's Analytical Engine is discussed in Chapter 12). The person who wrote these programs was a woman, Ada Lovelace (Figure 1-17), who was a talented mathematician. Thus, she is referred to as "the first computer programmer." This section discusses fundamental issues of computer software.

1.4.1 What Is Computer Software?

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer. This can be in the form of instructions on paper, or in digital form. While system software is intrinsic to a computer system, **application software** fulfills users' needs, such as a photo-editing program. We discuss the important concepts of syntax, semantics, and program translation next.



FIGURE 1-17 Ada Lovelace

“The First Computer Programmer”

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer.

1.4.2 Syntax, Semantics, and Program Translation

What Are Syntax and Semantics?

Programming languages (called “artificial languages”) are languages just as “natural languages” such as English and Mandarin (Chinese). *Syntax* and *semantics* are important concepts that apply to all languages.

The **syntax** of a language is a set of characters and the acceptable arrangements (sequences) of those characters. English, for example, includes the letters of the alphabet, punctuation, and properly spelled words and properly punctuated sentences. The following is a syntactically correct sentence in English,

“Hello there, how are you?” The following, however, is not syntactically correct, “Hello there, hao are you?”

In this sentence, the sequence of letters “hao” is not a word in the English language. Now consider the following sentence, “Colorless green ideas sleep furiously.” This sentence is syntactically correct, but is *semantically* incorrect, and thus has no meaning.

1.4 Computer Software 15

The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters. In Mandarin, “Hao” is syntactically correct meaning “good.” (“Hao” is from a system called pinyin, which uses the Roman alphabet rather than Chinese characters for writing Mandarin.) Thus, every language has its own syntax and semantics, as demonstrated in Figure 1-18.

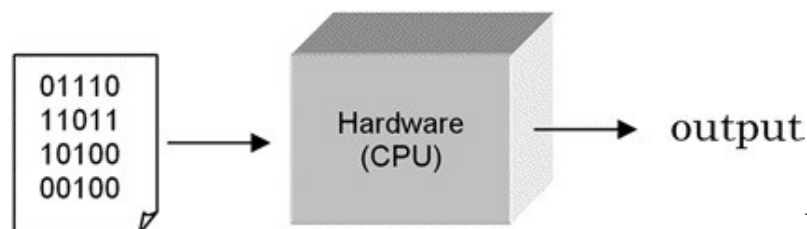
<u>ENGLISH</u>	<u>MANDARIN</u> (pinyin)	<u>MANDARIN</u> (Chinese Characters)
Syntax Hao	Syntax Hao	Syntax 好
Semantics No meaning (<i>syntactically incorrect</i>)	Semantics “Good”	Semantics “Good”

FIGURE 1-18 Syntax and Semantics of Languages

The **syntax** of a language is a set of characters and the acceptable sequences of those characters. The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters.

Program Translation

A central processing unit (CPU) is designed to interpret and execute a specific set of instructions represented in binary form (i.e., 1s and 0s) called **machine code**. Only programs in machine code can be executed by a CPU, depicted in Figure 1-19.



Machine Code

FIGURE 1-19 Execution of

Writing programs at this “low level” is tedious and error-prone. Therefore, most programs are written in a “high-level” programming language such as Python. Since the instructions of such programs are not in machine code that a CPU can execute, a translator program must be used. There are two fundamental types of translators. One, called a **compiler**, translates programs directly into machine code to be executed by the CPU, denoted in Figure 1-20.

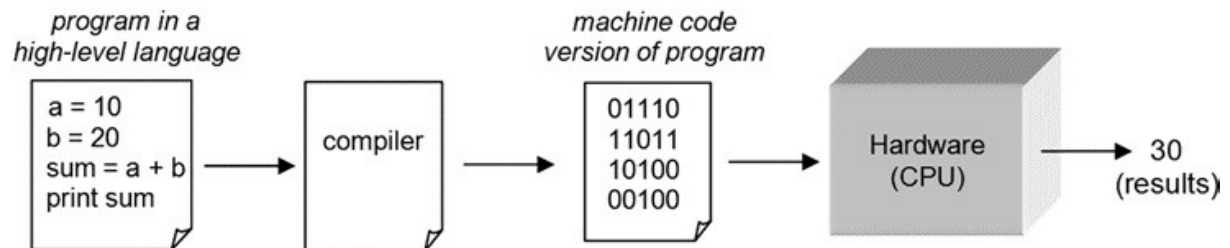


FIGURE 1-20 Program Execution by Use of a Compiler

The other type of translator is called an **interpreter**, which executes program instructions in place of (“running on top of”) the CPU, denoted in Figure 1-21.

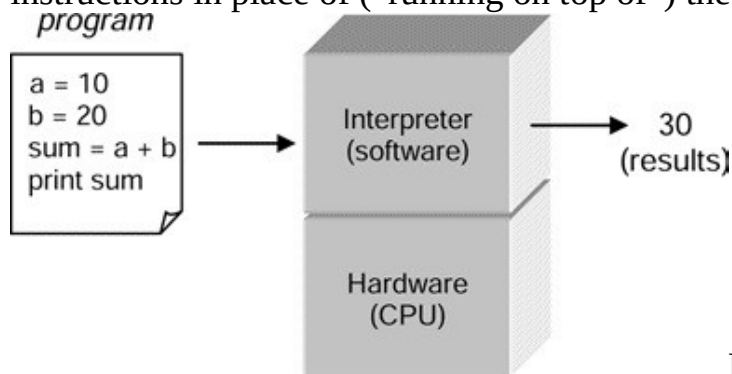


FIGURE 1-21 Program

Execution by Use of a
Interpreter

Thus, an interpreter can immediately execute instructions as they are entered. This is referred to as **interactive mode**. This is a very useful feature for program development. Python, as we shall see, is executed by an interpreter. On the other hand, compiled programs generally execute faster than interpreted programs. Any program can be executed by either a compiler or an interpreter, as long there exists the corresponding translator program for the programming language that it is written in.

A **compiler** is a translator program that translates programs directly into machine code to be executed by the CPU. An **interpreter** executes program instructions in place of (“running on top of”) the CPU.

Program Debugging: Syntax Errors vs. Semantic Errors

Program debugging is the process of finding and correcting errors (“**bugs**”) in a computer program. Programming errors are inevitable during program development. **Syntax errors** are caused by invalid syntax (for example, entering `prnt` instead of `print`). Since a translator cannot understand instructions containing syntax errors, translators terminate when encountering such errors indicating where in the program the problem occurred.

In contrast, **semantic errors** (generally called **logic errors**) are errors in program logic. Such errors cannot be automatically detected, since translators cannot understand the intent of a given computation. For example, if a program computed the average of three numbers as follows,

```
(num1 + num2 + num3) / 2.0
```

a translator would have no means of determining that the divisor should be 3 and not 2. *Computers do not understand what a program is meant to do, they only follow the instructions given.* It is up to the programmer to detect such errors. Program debugging is not a trivial task, and constitutes much of the time of program development.

Syntax errors are caused by invalid syntax. **Semantic (logic) errors** are caused by errors in program logic.

1.4.3 Procedural vs. Object-Oriented Programming

Programming languages fall into a number of *programming paradigms*. The two major programming paradigms in use today are *procedural (imperative) programming* and *object-oriented programming*. Each provides a different way of thinking about computation. While most programming languages only support one paradigm, Python supports both procedural and object-oriented programming. We will start with the procedural aspects of Python. We then introduce objects in Chapter 6, and delay complete discussion of object-oriented programming until Chapter 10.

Procedural programming and **object-oriented programming** are two major programming paradigms in use today.

Self-Test Questions

1. Two general types of software are system software and _____

software.

2. The syntax of a given language is,

- (a) the set of symbols in the language.
- (b) the acceptable arrangement of symbols.
- (c) both of the above

3. The semantics of a given language is the meaning associated with any arrangement of symbols in the language. (TRUE/FALSE)

4. CPUs can only execute instructions that are in binary form called _____.

5. The two fundamental types of translation programs for the execution of computer programs are _____ and _____.

6. The process of finding and correcting errors in a computer program is called _____.

7. Which kinds of errors can a translator program detect?

- (a) Syntax errors
- (b) Semantic errors
- (c) Neither of the above

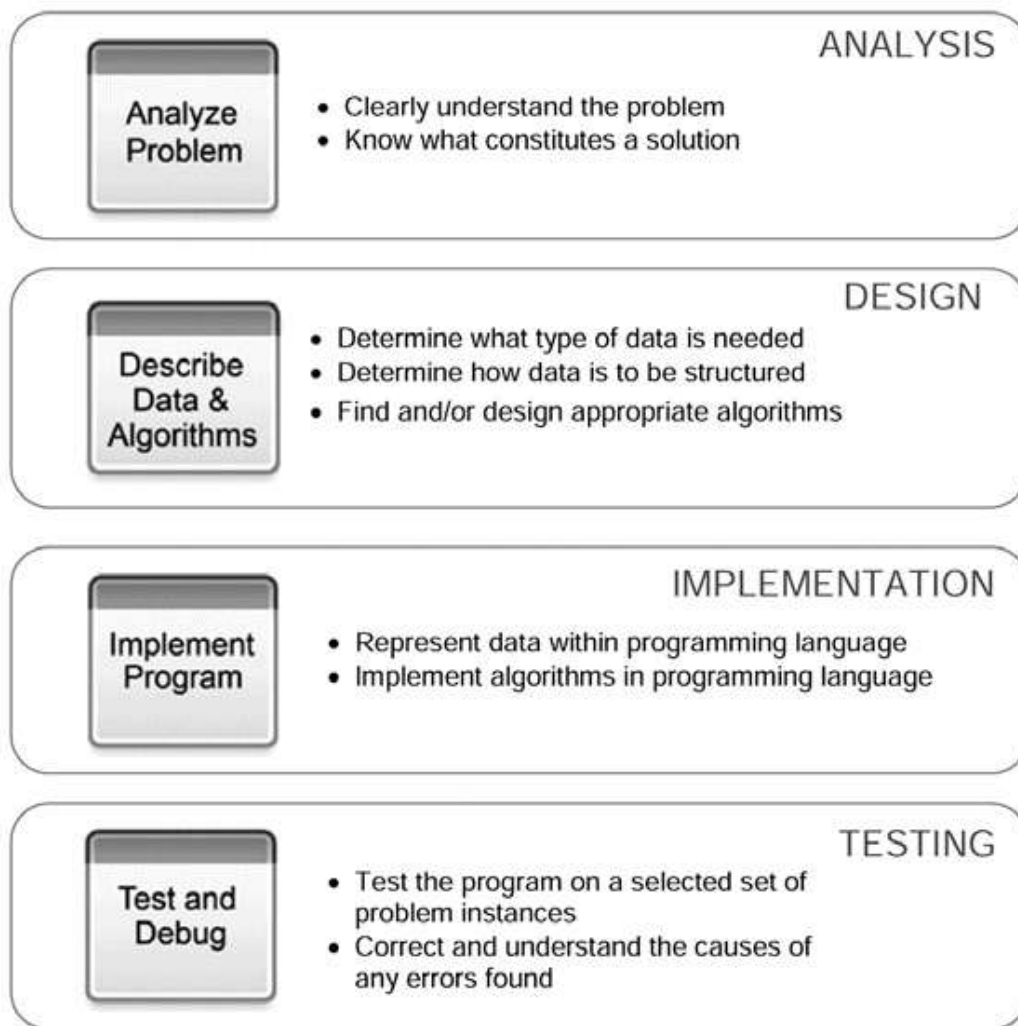
8. Two major programming paradigms in use today are _____ programming and _____ programming.

ANSWERS: 1. application, 2. (c), 3. False, 4. machine code, 5. compilers, interpreters, 6. program debugging, 7. (a), 8. procedural, object-oriented

COMPUTATIONAL PROBLEM SOLVING

1.5 The Process of Computational Problem Solving

Computational problem solving does not simply involve the act of computer programming. It is a *process*, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined in Figure 1-22.



FIGURE

1-22 Process of Computational Problem Solving

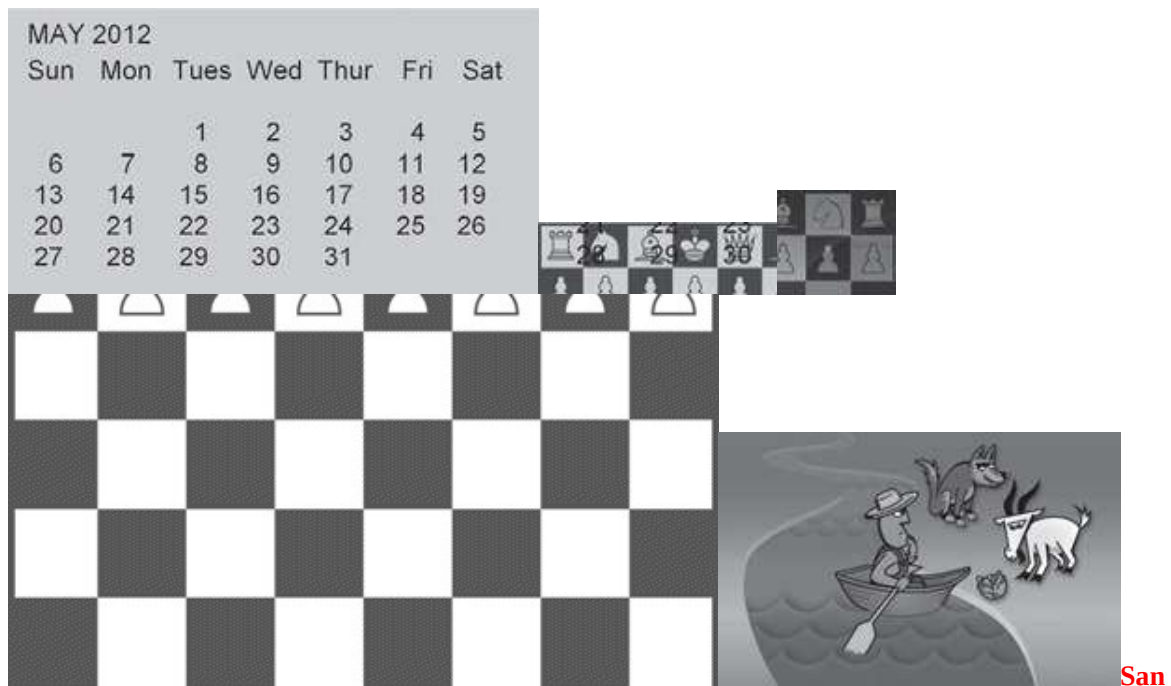
1.5.1 Problem Analysis

Understanding the Problem

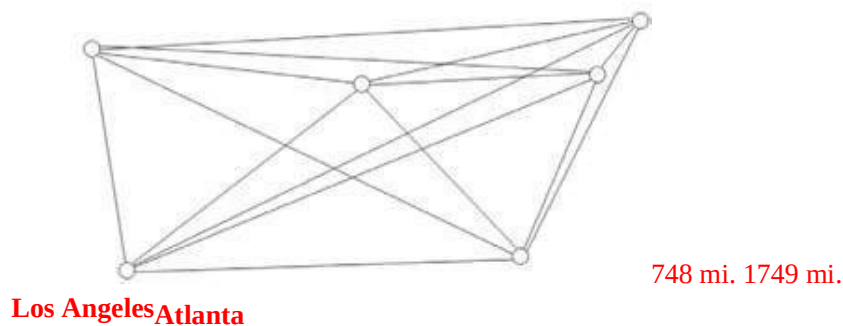
Once a problem is clearly understood, the fundamental computational issues for solving it can be determined. For each of the problems discussed earlier, the representation is straightforward. For the calendar month problem, there are two algorithmic tasks—determining the first day of a given month, and displaying the calendar month in the proper format. The first day of the month can be obtained by *direct calculation* by use of the algorithm provided in Figure 1-8.

For the Man, Cabbage, Goat, Wolf (MCGW) problem, a brute-force algorithmic approach of trying all possible solutions works very well, since there are a small

number of actions that can be taken at each step, and only a relatively small number of steps for reaching a solution. For both the Traveling Salesman problem and the game of chess, the brute-force approach is infeasible. Thus, the computational issue for these problems is to find other, more efficient algorithmic approaches for their solution. (In fact, methods have been developed for solving Traveling Salesman problems involving tens of thousands of cities. And current chess-playing programs can beat top-ranked chess masters.)



New York Chicago
344 mi.



Knowing What Constitutes a Solution

Besides clearly understanding a computational problem, one must know what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding,

- ◆ A solution
- ◆ An *approximate* solution
- ◆ A *best* solution
- ◆ *All* solutions

For the MCGW problem, there are an infinite number of solutions since the man could pointlessly row back and forth across the river an arbitrary number of times. A *best solution* here is one with the shortest number of steps. (There may be more than one “best” solution for any given problem.) In the Traveling Salesman problem there is only one solution (unless there exists more than one shortest route). Finally, for the game of chess, the goal (solution) is to win the game. Thus, since the number of chess games that can be played is on the order of 10^{120} (with each game ending in a win, a loss, or a stalemate), there are a comparable number of possible solutions to this problem.

1.5.2 Program Design

Describing the Data Needed

For the Man, Cabbage, Goat, Wolf problem, a list can be used to represent the correct location (east and west) of the man, cabbage, goat, and wolf as discussed earlier, reproduced below, man cabbage goat wolf
[W, E, W, E]

For the Calendar Month problem, the data include the month and year (entered by the user), the number of days in each month, and the names of the days of the week. A useful structuring of the data is given below,

[month, year]
[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']

The month and year are grouped in a single list since they are naturally associated. Similarly, the names of the days of the week and the number of days

in each month are grouped. (The advantages of list representations will be made clear in Chapter 4.) Finally, the first day of the month, as determined by the algorithm in Figure 1-8, can be represented by a single integer,

0 – Sunday, 1 – Monday, . . . , 6 – Saturday

For the Traveling Salesman problem, the distance between each pair of cities must be represented. One possible way of structuring the data is as a table, depicted in Figure 1-23.

For example, the distance from Atlanta to Los Angeles is 2175 miles. There is duplication of information in this representation, however. For each distance from x to y , the distance from y to x is

	Atlanta	Boston	Chicago	Los Angeles	New York City	San Francisco
Atlanta	-	1110	718	2175	888	2473
Boston	1110	-	992	2991	215	3106
Chicago	718	992	-	2015	791	2131
Los Angeles	2175	2991	2015	-	2790	381
New York City	888	215	791	2790	-	2901
San Francisco	2473	3106	2131	381	2901	-

FIGURE 1-23 Table Representation of Data

also represented. If the size of the table is small, as here, this is not much of an issue. However, for a significantly larger table, significantly more memory would be wasted during program execution. Since only half of the table is really needed (for example, the shaded area in the figure), the data could be represented as a list of lists instead,

```
[ ['Atlanta', ['Boston', 1110], ['Chicago', 718], ['Los Angeles', 2175], ['New
York', 888], ['San Francisco', 2473] ],
['Boston', ['Chicago', 992], ['Los Angeles', 2991], ['New York', 215], ['San
Francisco', 3106] ],
['Chicago', ['Los Angeles', 2015], ['New York', 791], ['San Francisco', 2131] ],
['Los Angeles', ['New York', 2790], ['San Francisco', 381] ],
['New York', ['San Francisco', 2901] ] ]
```

Finally, for a chess-playing program, the location and identification of each chess piece needs to be represented (Figure 1-24). An obvious way to do this is shown on the left below, in which each piece is represented by a single letter

(‘K’ for the king, ‘Q’ for the queen, ‘N’ for the knight, etc.),



FIGURE 1-24 Representations of Pieces on a Chess Board

There is a problem with this choice of symbols, however—there is no way to distinguish the white pieces from the black ones. The letters could be modified, for example, PB for a black pawn and PW for a white pawn. While that may be an intuitive representation, it is not the best representation for a program. A better way would be to represent pieces using positive and negative integers as shown on the right of the figure: 1 for a white pawn and 21 for a black pawn; 2 for a white bishop and 22 for a black bishop, and so forth. Various ways of representing chess boards have been developed, each with certain advantages and disadvantages. The appropriate representation of data is a fundamental aspect of computer science.

Describing the Needed Algorithms

When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. For the MCGW problem, there are standard search algorithms that can be used. For the calendar month problem, a day of the week algorithm already exists. For the Traveling Salesman problem, there are various (nontrivial) algorithms that can be utilized, as mentioned, for solving problems with tens of thousands of cities. Finally, for the game of chess, since it is infeasible to look ahead at the final outcomes of every possible move, there are algorithms that make a best guess at which moves to make. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called *heuristic algorithms*.

1.5.3 Program Implementation

Design decisions provide *general* details of the data representation and the

algorithmic approaches for solving a problem. The details, however, do not specify which programming language to use, or how to implement the program. That is a decision for the implementation phase. Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

1.5.4 Program Testing

As humans, we have abilities that far exceed the capabilities of any machine, such as using commonsense reasoning, or reading the expressions of another person. However, one thing that we are not very good at is dealing with detail, which computer programming demands. Therefore, while we are enticed by the existence of increasingly capable computing devices that unfailingly and speedily execute whatever instructions we give them, writing computer programs is difficult and challenging. As a result, *programming errors are pervasive, persistent and inevitable*. However, the sense of accomplishment of developing software that can be of benefit to hundreds, thousands, or even millions of people can be extremely gratifying. If it were easy to do, the satisfaction would not be as great.

Given this fact, software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated. In subsequent chapters, program testing and debugging will be discussed and expanded upon. For now, we provide the following general truisms of software development in Figure 1-25.

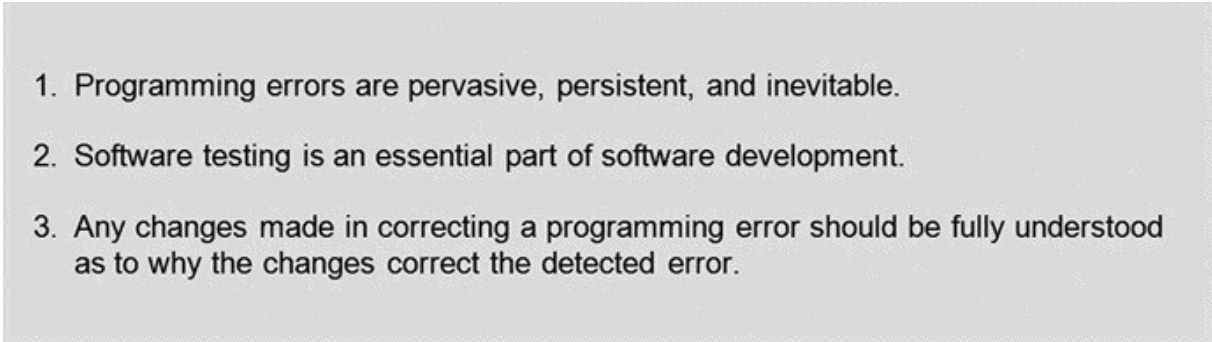
- 
1. Programming errors are pervasive, persistent, and inevitable.
 2. Software testing is an essential part of software development.
 3. Any changes made in correcting a programming error should be fully understood as to why the changes correct the detected error.

FIGURE 1-25 Truisms of Software Development

Truism 1 reflects the fact that programming errors are inevitable and that we must accept it. As a result of truism 1, truism 2 states the essential role of

software testing. Given the inevitability of programming errors, it is important to test a piece of software in a thorough and systematic manner. Finally, truism 3 states the importance of understanding *why* a given change (or set of changes) in a program fixes a specific error. If you make a change to a program that fixes a given problem but you don't know why it did, then you have lost control of the program logic. As a result, you may have corrected one problem, but inadvertently caused other, potentially more serious ones.

Accountants are committed to reconciling balances to the penny. They do not disregard a discrepancy of one cent, for example, even though the difference between the calculated and expected balances is so small. They understand that a small, seemingly insignificant difference can be the result of two (or more) very big discrepancies. For example, there may be an erroneous credit of \$1,500.01 and an erroneous debit of \$1,500. (The author has experienced such a situation in which the people involved worked all night to find the source of the error.) Determining the source of errors in a program is very much the same. We next look at the Python programming language.

1.6 The Python Programming Language

Now that computational problem solving and computer programming have been discussed, we turn to the Python programming language and associated tools to begin putting this knowledge into practice.

1.6.1 About Python

Guido van Rossum (Figure 1-26) is the creator of the Python programming language, first released in the early 1990s. Its name comes from a 1970s British comedy sketch television show called *Monty Python's Flying Circus*. (Check them out on YouTube!) The development environment IDLE provided with Python (discussed below) comes from the name of a member of the comic group.

Python has a simple syntax. Python programs are clear and easy to read. At the same time, Python provides powerful programming features, and is widely used. Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA. Python is well supported and freely available at www.python.org. (See the Python 3 Programmers' Reference at the end of the text for how to download and install Python.)

1.6.2 The IDLE Python Development Environment

IDLE is an **integrated development environment (IDE)**. An IDE is a bundled set of software tools

for program development. This typically includes



FIGURE 1-26 Guido van Rossum

an **editor** for creating and modifying programs, a **translator** for executing programs, and a **program debugger**. A debugger provides a means of taking control of the execution of a program to aid in finding program errors.

Python is most commonly translated by use of an interpreter. Thus, Python provides the very useful ability to execute in interactive mode. The window that

provides this interaction is referred to as the **Python shell**. Interacting with the shell is much like using a calculator, except that, instead of being limited to the operations built into a calculator (addition, subtraction, etc.), it allows the entry and creation of any Python code. Example use of the Python shell is demonstrated in Figure 1-27.

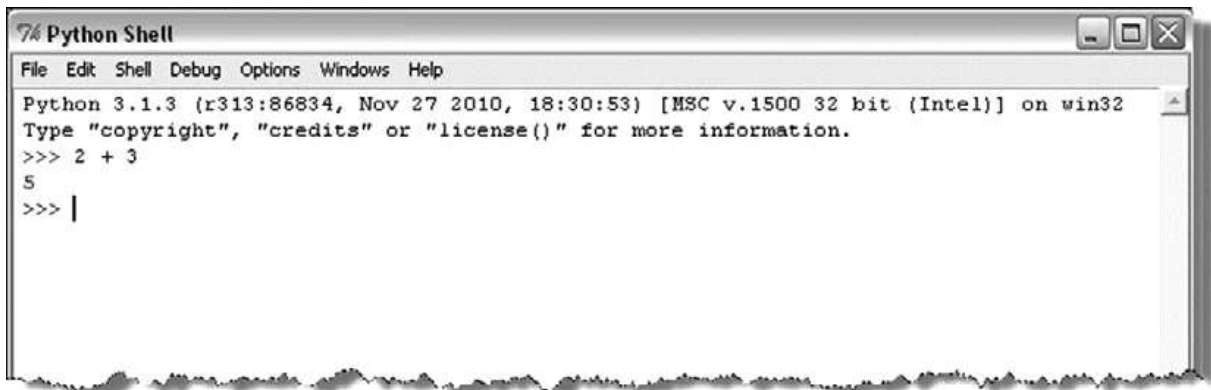


FIGURE 1-27 Python Shell

Here, the expression `2 + 3` is entered at the **shell prompt** (`>>>`), which immediately responds with the result 5.

Although working in the Python shell is convenient, the entered code is not saved. Thus, for program development, a means of entering, editing, and saving Python programs is provided by the program editor in IDLE. Details are given below.

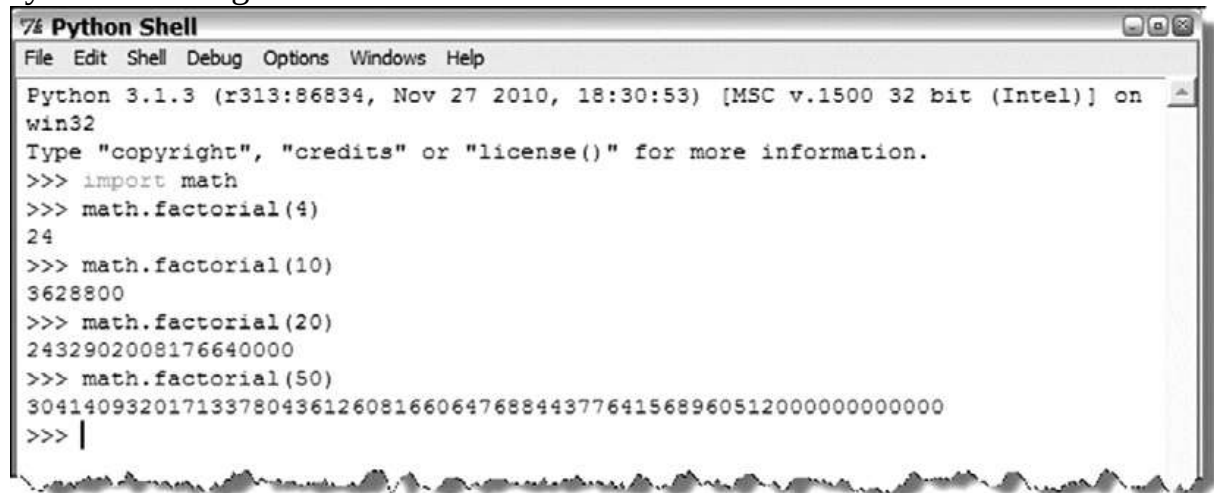
An **Integrated Development Environment (IDE)** is a bundled set of software tools for program development.

1.6.3 The Python Standard Library

The **Python Standard Library** is a collection of *built-in modules*, each providing specific functionality beyond what is included in the “core” part of Python. (We discuss the creation of Python modules in Chapter 7.) For example, the `math` module provides additional mathematical functions. The `random` module provides the ability to generate random numbers, useful in programming, as we shall see. (Other Python modules are described in the Python 3 Programmers’ Reference.) In order to utilize the capabilities of a given module in a specific program, an **import** statement is used as shown in Figure 1-28.

The example in the figure shows the use of the `import math` statement to gain

access to a particular function in the math module, the factorial function. The syntax for using the factorial

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following text:

```
Python 3.1.3 (r313:86834, Nov 27 2010, 18:30:53) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> math.factorial(4)
24
>>> math.factorial(10)
3628800
>>> math.factorial(20)
2432902008176640000
>>> math.factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
>>> |
```

FIGURE 1-28 Using an import statement

function is `math.factorial(n)`, for some positive integer `n`. We will make use of library modules in Chapter 2. In section 1.7, we see how to enter and execute a complete Python program.

The **Python Standard Library** is a collection of *modules*, each providing specific functionality beyond what is included in the core part of Python.

1.6.4 A Bit of Python

We introduce a bit of Python, just enough to begin writing some simple programs. Since all computer programs input data, process the data, and output results, we look at the notion of a variable, how to perform some simple arithmetic calculations, and how to do simple input and output.

Variables

One of the most fundamental concepts in programming is that of a *variable*. (Variables are discussed in detail in Chapter 2.) A simple description of a variable is “a name that is assigned to a value,” as shown below,

`n = 5` variable is assigned the value 5 Thus, whenever variable `n` appears in a calculation, it is the current value that `n` is assigned to that is used, as in the following,

`n = 120` (5 + 1 + 20) If variable `n` is assigned a new value, then the same expression will produce a different result,

`n = 510`

n 1 20 (10 1 20) We next look at some basic arithmetic operators of Python. A **variable** is “a name that is assigned to a value.”

Some Basic Arithmetic Operators

The common arithmetic operators in Python are 1 (addition), 2 (subtraction), * (multiplication), / (division), and ** (exponentiation). Addition, subtraction, and division use the same symbols as standard mathematical notation,

10 120 25 215 20 / 10

(There is also the symbol // for truncated division, discussed in Chapter 2.) For multiplication and exponentiation, the asterisk (*) is used.

5 * 10 (5 times 10) 2 ** 4 (2 to the 4th power) Multiplication is never denoted by the use of parentheses as in mathematics, as depicted below, 10 * (20 15)

CORRECT 10(20 15) INCORRECT Note that parentheses may be used to denote subexpressions. Finally, we see how to input information from the user, and display program results.

The common arithmetic operators in Python are 1 (addition), 2 (subtraction), * (multiplication), / (division), and ** (exponentiation).

Basic Input and Output

The programs that we will write request and get information from the user. In Python, the input function is used for this purpose,

name 5 input('What is your name?: ') Characters within quotes are called *strings*.

This particular use of a string, for requesting input from The input function displays the string on the screen to prompt the user the user, is called a *prompt*. for input,

What is your name?: Charles

The underline is used here to indicate the user's input.

The print function is used to display information on the screen in Python. This may be used to display a message,

```
>>> print('Welcome to My First Program!') Welcome to My First Program!
```

or used to output the value of a variable,

```
>>> n 5 10 >>> print(n) 10
```

or to display a combination of both strings and variables,

```
>>> name 5 input('What is your name?: ') What is your name?: Charles
```

```
>>> print('Hello', name)
```


Hello Charles

Note that a comma is used to separate the individual items being printed, causing a space to appear between each when displayed. Thus, the output of the print function in this case is Hello Charles, and not HelloCharles. There is more to say about variables, operators, and input/ output in Python. This will be covered in the chapters ahead.

In Python, input is used to request and get information from the user, and print is used to display information on the screen.

1.6.5 Learning How to Use IDLE

In order to become familiar with writing your own Python programs using IDLE, we will create a simple program that asks the user for their name and responds with a greeting. This program utilizes the following concepts:



- ◆ creating and executing Python programs
- ◆ input and print

First, to create a Python program file, select New Window from the File menu in the Python shell as shown in Figure 1-29:

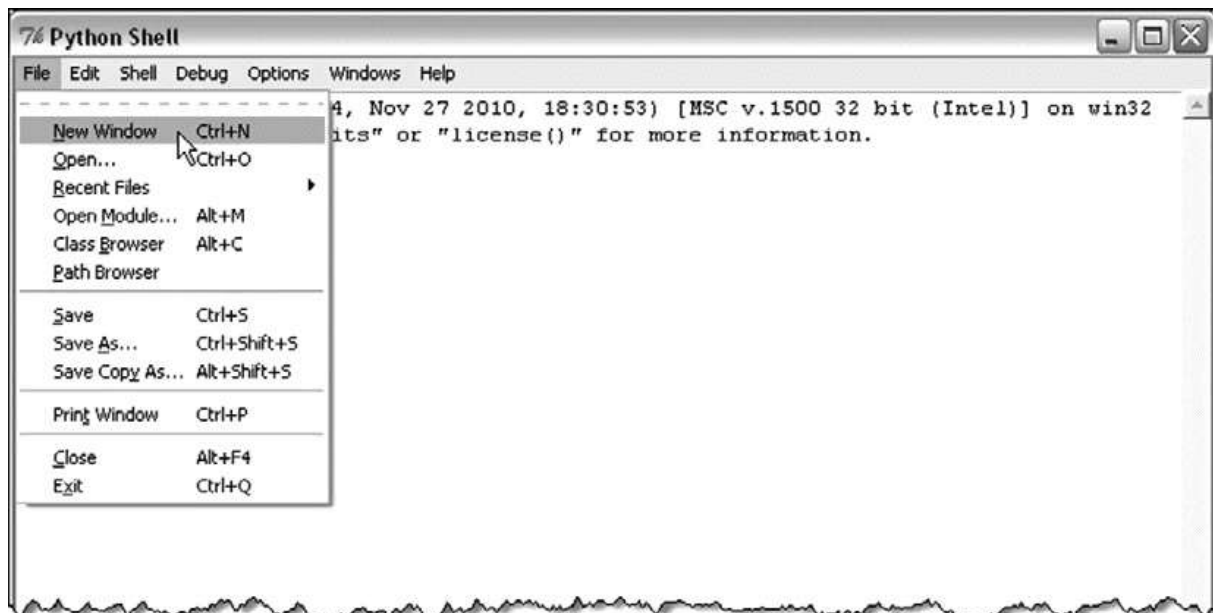
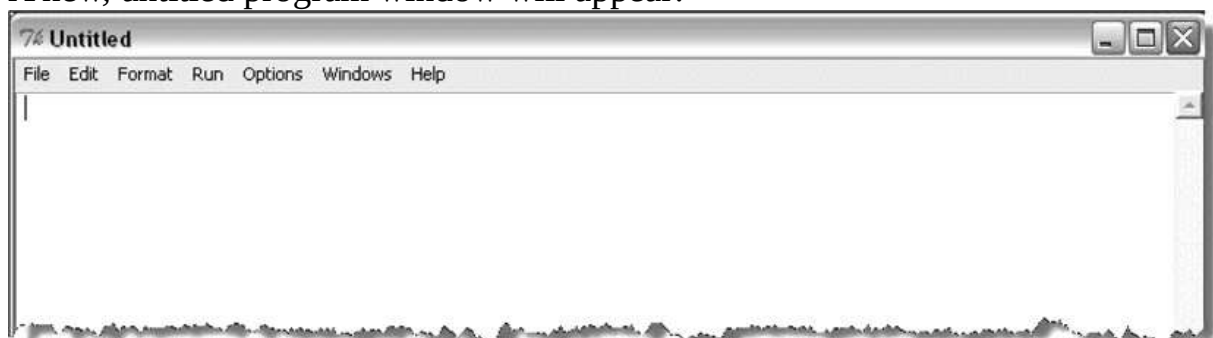
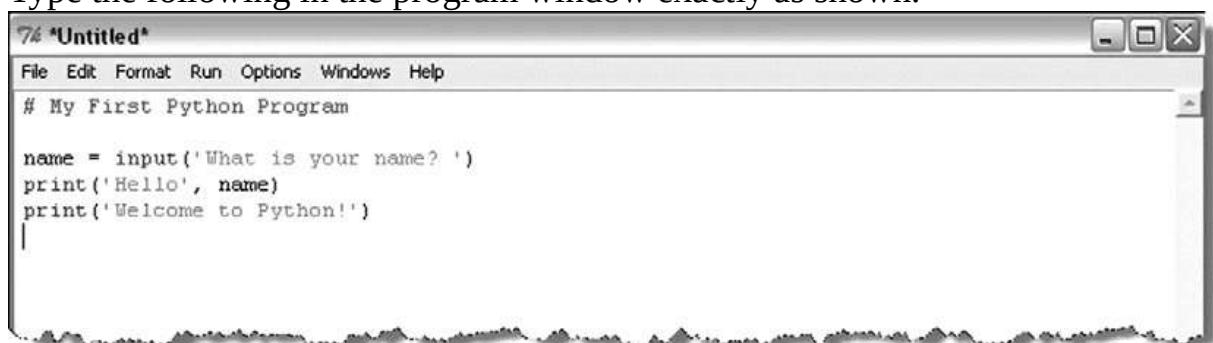


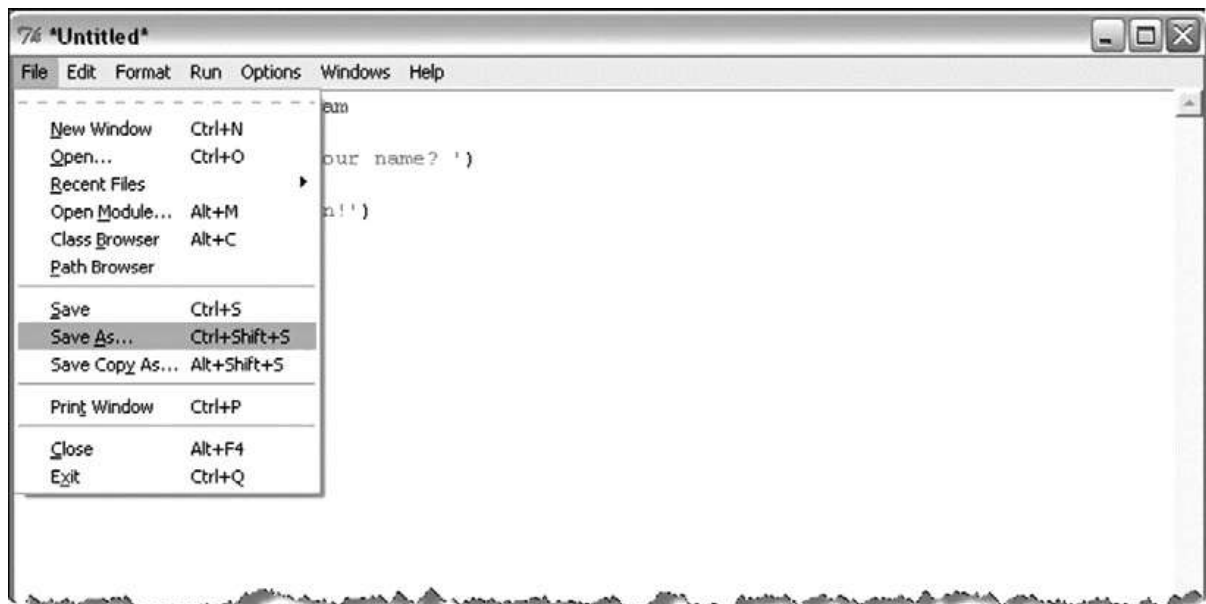
FIGURE 1-29 Creating a Python Program File
A new, untitled program window will appear:



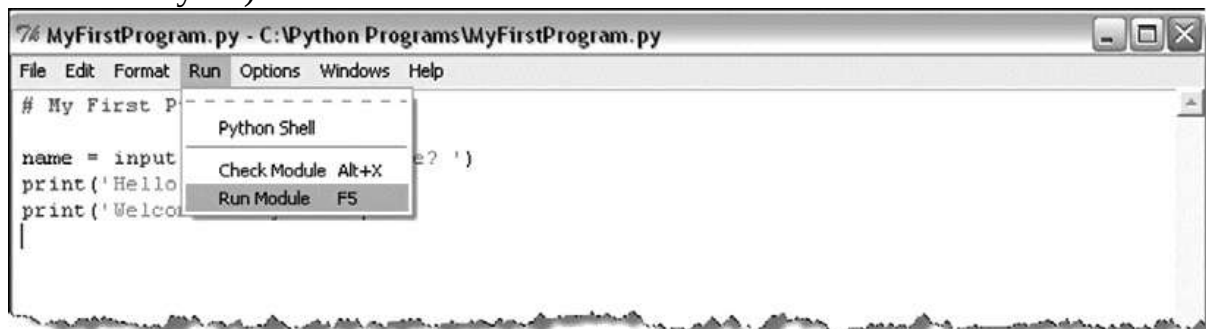
Type the following in the program window exactly as shown.



When finished, save the program file by selecting Save As under the File menu, and save in the appropriate folder with the name MyFirstProgram.py.



To run the program, select Run Module from the Run menu (or simply hit function key F5).



If you have entered the program code correctly, the program should execute as shown in Figure 1-30.

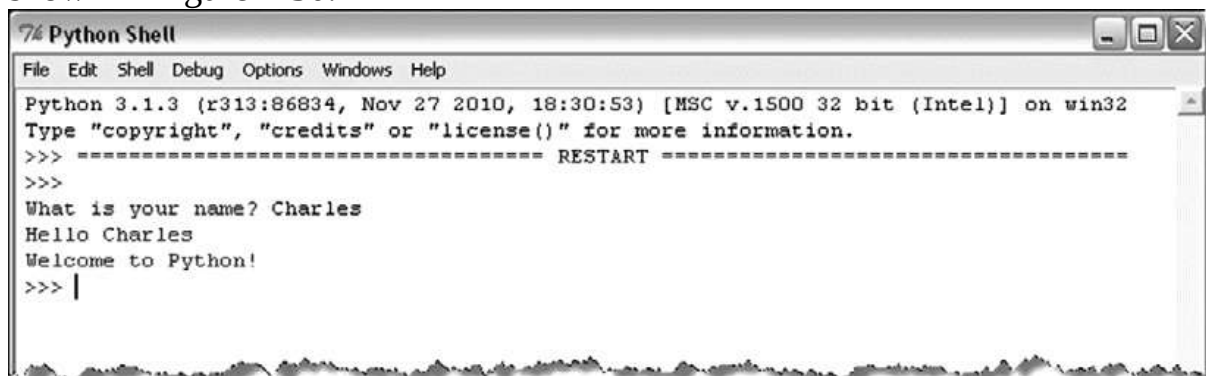
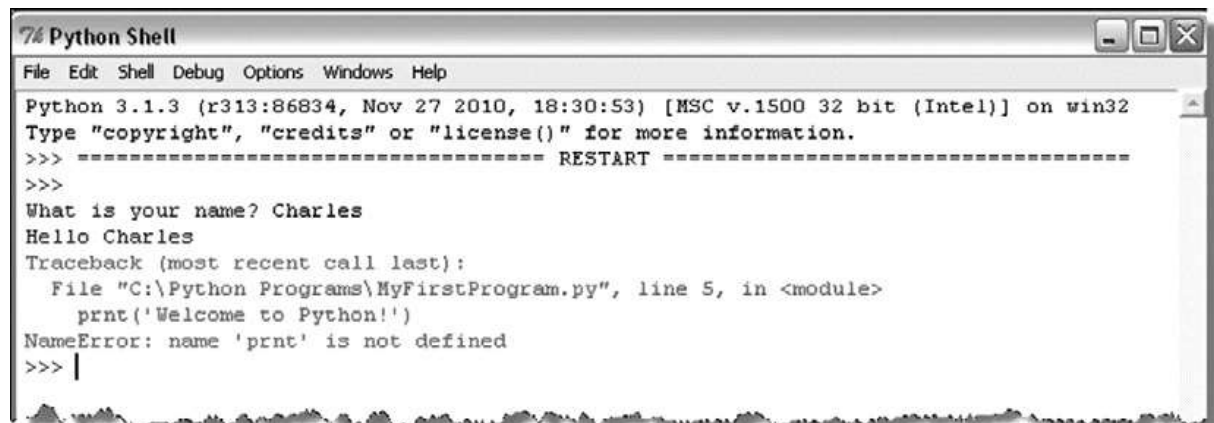


FIGURE 1-30 Sample Output of MyFirstProgram.py

If, however, you have mistyped part of the program resulting in a syntax error (such as mistyping print), you will get an error message similar to that in Figure

1-31.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.3 (r313:86834, Nov 27 2010, 18:30:53) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
What is your name? Charles
Hello Charles
Traceback (most recent call last):
  File "C:\Python Programs\MyFirstProgram.py", line 5, in <module>
    prnt('Welcome to Python!')
NameError: name 'prnt' is not defined
>>> |
```

FIGURE 1-31 Output Resulting from a Syntax Error

In that case, go back to the program window and make the needed corrections, then re-save and re-execute the program. You may need to go through this process a number of times until all the syntax errors have been corrected.

1.7 A First Program—Calculating the Drake Equation

Dr. Frank Drake conducted the first search for radio signals from extraterrestrial civilizations in 1960. This established SETI (Search for Extraterrestrial Intelligence), a new area of scientific inquiry. In order to estimate the number of civilizations that may exist in our galaxy that we may be able to communicate with, he developed what is now called the *Drake equation*.

The Drake equation accounts for a number of different factors. The values used for some of these are the result of scientific study, while others are only the result of an “intelligent guess.” The factors consist of *R*, the average rate of star creation per year in our galaxy; *p*, the percentage of those stars that have planets; *n*, the average number of planets that can potentially support life for each star with planets; *f*, the percentage of those planets that actually go on to develop life; *i*, the percentage of those planets that go on to develop intelligent life; *c*, the percentage of those that have the technology communicate with us; and *L*, the expected lifetime of civilizations (the period that they can communicate). The Drake equation is simply the multiplication of all these factors, giving *N*, the estimated number of detectable civilizations there are at any given time,



N 5 R ? p ? n ? f ? i ? c ? L

Figure 1-32 shows those parameters in the Drake equation that have some consensus as to their correct value.

Drake Equation Factor Values		Estimated Values
Rate of star creation	R	7 [†]
Percentage of stars with planets	p	40%
Average number of planets that can potentially support life for each star with planets	n	(no consensus)
Percentage of those that go on to develop life	f	13%
Percentage of those that go on to intelligent develop life	i	(no consensus)
Percentage of those willing and able to communicate	c	(no consensus)
Expected lifetime of civilizations	L	(no consensus)
[†] Estimate of NASA and the European Space Agency		

FIGURE

1-32 Proposed Values for the Drake Equation

1.7.1 The Problem

The value of 7 for R, the rate of star creation, is the least disputed value in the Drake equation today. Given the uncertainty of the remaining factors, you are to develop a program that allows a user to enter their own estimated values for the

remaining six factors (p, n, f, i, c, and L) and displays the calculated result.

1.7.2 Problem Analysis

This problem is very straightforward. We only need to understand the equation provided.

1.7.3 Program Design

The program design for this problem is also straightforward. The data to be represented consist of numerical values, with the Drake equation as the algorithm. The overall steps of the program are depicted in Figure 1-33.

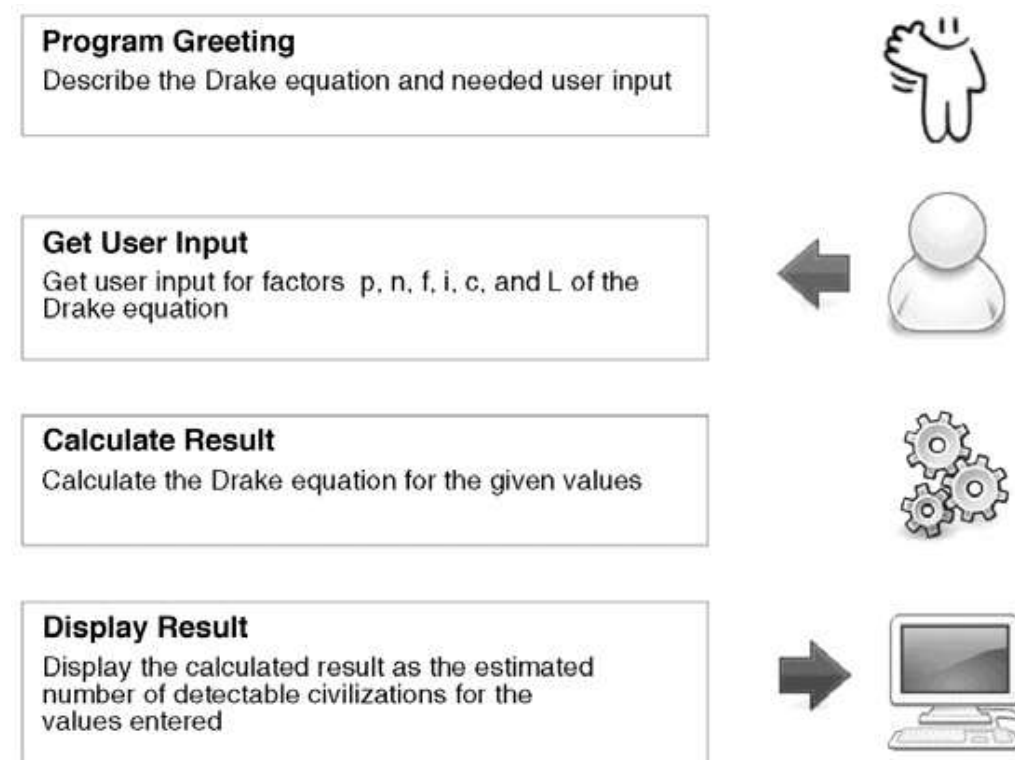


FIGURE 1-

33 The Overall Steps of the Drake Equation Program

1.7.4 Program Implementation

The implementation of this program is fairly simple. The only programming elements needed are input, assignment, and print, along with the use of arithmetic operators. An implementation is given in Figure 1-34. Example execution of the program is given in Figure 1-35.

First, note the program lines beginning with the hash sign, #. In Python, this symbol is used to denote a *comment statement*. A **comment statement** contains

information for persons reading the program. Comment statements are ignored during program execution—they have no effect on the program results. In this program, the initial series of comment statements (**lines 1–23**) explain the Drake equation and provide a brief summary of the purpose of the program.

```
1 # SETI Program
2 #
3 # The Drake equation, developed by Frank Drake in the 1960s, attempts to
4 # estimate how many extraterrestrial civilizations, N, may exist in our
5 # galaxy at any given time that we might come in contact with,
6 #
7 #     N = R * p * n * f * i * c * L
8 #
9 # where,
10 #
11 #     R ... estimated rate of star creation in our galaxy
12 #     p ... estimated percent of stars that have planets
13 #     n ... estimated average number of planets that can potentially support
14 #         life for each star with planets
15 #     f ... estimated percent of those planets that actually go on to develop life
16 #     i ... estimated percent of those planets go on to develop intelligent life
17 #     c ... estimated percent of those that are willing and able to communicate
18 #     L ... estimated expected lifetime of such civilizations
19 #
20 # Given that the value for R, 7 per year, is the least disputed of the values,
21 # the user will be prompted to enter estimated values for the remaining six
22 # factors. The estimated number of civilizations that may be detected in our
23 # galaxy will then be displayed.
24 #
25 # display program welcome
26 print('Welcome to the SETI program')
27 print('This program will allow you to enter specific values related to')
28 print('the likelihood of finding intelligent life in our galaxy. All')
29 print('percentages should be entered as integer values, e.g., 40 and not .40')
30 print()
31 #
32 # get user input
33 p = int(input('What percentage of stars do you think have planets?: '))
34 n = int(input('How many planets per star do you think can support life?: '))
35 f = int(input('What percentage do you think actually develop life?: '))
36 i = int(input('What percentage of those do you think have intelligent life?: '))
37 c = int(input('What percentage of those do you think can communicate with us?: '))
38 L = int(input('Number of years you think civilizations last?: '))
39 #
40 # calculate result
41 num_detectable_civilizations = 7 * (p/100) * n * (f/100) * (i/100) * (c/100) * L
42 #
43 # display result
44 print()
45 print('Based on the values entered ...')
46 print('there are an estimated', round(num_detectable_civilizations),
47       'potentially detectable civilizations in our galaxy')
```

FIGURE 1-34 Drake Equation Program

Comment statements are also used in the program to denote the beginning of each program section (**lines 25, 32, 40, and 43**). These *section headers* provide a

broad outline of the program following the program design depicted in Figure 1-33.

The program welcome section (**lines 25–30**) contains a series of print instructions displaying output to the screen. Each begins with the word print followed by a matching pair of parentheses. Within the parentheses are the items to be displayed. In this case, each contains a particular string of characters. The final “empty” print, print() on **line 30** (and **line 44**), does not display anything. It simply causes the screen cursor to move down to the next line, therefore creating a skipped line in the screen output. (Later we will see another way of creating the same result.)

```
Program Execution ...

Welcome to the SETI program
This program will allow you to enter specific values related to
the likelihood of finding intelligent life in our galaxy. All
percentages should be entered as integer values, e.g., 40 and not .40

What percentage of stars do you think have planets?: 40
How many planets per star do you think can support life?: 2
What percentage do you think actually develop life?: 5
What percentage of those do you think have intelligent life?: 3
What percentage of those do you think can communicate with us?: 5
Number of years you think civilizations last?: 10000

Based on the values entered...
there are an estimated 4.2 potentially detectable civilizations in
our galaxy
>>>
```

FIGURE 1-35 Execution of the Drake Equation Program

The following section (**lines 32–38**) contains the instructions for requesting the input from the user. Previously, we saw the input function used for inputting a name entered by the user. In that case, the instruction was of the form,

```
name = input('What is your name?:')
```

In this program, there is added syntax,

```
p = int(input('What percentage of stars do you think have planets?:'))
```

The input function always returns what the user enters as a string of characters. This is appropriate when a person’s name is being entered. However, in this program, numbers are being entered, not names. Thus, in the following,

What percentage of stars do you think have planets?: 40

40 is to be read as single number and not as the characters '4' and '0'. The addition of the `int (. . .)` syntax around the input instruction accomplishes this. This will be discussed more fully when numeric and string (character) data are discussed in Chapter 2.

On **line 41** the Drake equation is calculated and stored in variable `num_detectable_civilizations`. Note that since some of the input values were meant to be percentages (`p`, `f`, `i`, and `c`), those values in the equation are each divided by 100. Finally, **lines 44–47** display the results.

1.7.5 Program Testing

To test the program, we can calculate the Drake equation for various other values using a calculator, providing a set of *test cases*. A **test case** is a set of input values and expected output of a given program. A **test plan** consists of a number of test cases to verify that a program meets all requirements. A good strategy is to include “average,” as well as “extreme” or “special” cases in a test plan. Such a test plan is given in Figure 1-36.

Based on these results, we can be fairly confident that the program will give the correct results for all input values.

Chapter Summary 33

Input Values							Expected Results	Actual Results	Evaluation
p	n	f	i	c	L				
Extreme Cases (no chance of contacting intelligent life)									
Zero Planets per Star	0	2	100%	1%	1%	10,000	0	0	passed
Zero percent of Planets Support Life	50%	0	100%	1%	1%	10,000	0	0	passed
Average Cases									
Average Case 1	30%	3	75%	1%	5%	5,000	12	12	passed
Average Case 2	50%	6	80%	5%	10%	10,000	840	840	passed
Extreme Cases (great chance of contacting intelligent life)									
Extreme Case 1	100%	10	100%	100%	100%	10,000	700,000	700,000	passed
Extreme Case 2	100%	12	100%	100%	100%	100,000	8,400,000	8,400,000	passed

FIGURE 1-36 Test Plan Results for Drake's Equation Program

CHAPTER SUMMARY General Topics

Computational Problem Solving/Representation/ Abstraction
 Algorithms/Brute Force Approach
 Computer Hardware/Transistors/Integrated Circuits/ Moore's Law
 Binary Representation/Bit/Byte/
 Binary-Decimal Conversion
 Central Processing Unit (CPU)/Main and Secondary Memory
 Input/Output Devices/Buses
 Computer Software/System Software/
 Application Software

Operating Systems

Syntax and Semantics/Program Translation/ Compiler vs. Interpreter

Program Debugging/Syntax Errors vs. Logic (Semantic) Errors

Procedural Programming vs. Object-Oriented Programming

The Process of Computational Problem Solving/ Program Testing

Integrated Development Environments (IDE)/ Program Editors/Debuggers

Comment Statements as Program Documentation

Test Cases/Test Plans

Python-Specific Programming Topics

The Python Programming Language/ Guido van Rossum (creator of Python)

Comment Statements in Python

Introduction to Variables, Arithmetic Operators, input and print in Python

Introduction to Strings with the input Function in Python

Introduction to the Python Standard Library and the import Statement

CHAPTER EXERCISES Section 1.1

The Python Shell/The IDLE Integrated Development Environment

The Standard Python Library

1. Search online for two computing-related fields named “computational X” other than the ones listed in Figure 1-1.
2. Search online for two areas of computer science other than those given in the chapter.
3. For the Man, Cabbage, Goat, Wolf problem:
 - (a) List all the invalid states for this problem, that is, in which the goat is left alone with the cabbage, or the wolf is left alone with the goat.
 - (b) Give the shortest sequence of steps that solves the MCGW problem.
 - (c) Give the sequence of state representations that correspond to your solution starting with (E,E,E,E) and ending with (W,W,W,W).
 - (d) There is an alternate means of representing states. Rather than a sequence representation, a set representation can be used. In this representation, if an item is on the east side of the river, its symbol is in the set, and if on the west side, the symbol is not in the set as shown below,

{M,C,G,W}—all items on east side of river (start state)

{C,W}—cabbage and wolf on east side of river, man and goat on west side { }

—all items on the west side of the river (goal state)

Give the sequence of states for your solution to the problem using this new state representation. **(e)** How many shortest solutions are there for this problem?

4. For a simple game that starts with five stones, in which each player can pick up either one or two stones, the person picking up the last stone being the loser,

(a) Give a state representation appropriate for this problem.

(b) Give the start state and goal state for this problem.

(c) Give a sequence of states in which the first player wins the game.

Section 1.2

5. Using the algorithm in Figure 1-8, *show all steps* for determining the day of the week for January 24,

2018. (Note that 2018 is not a leap year.)

6. Using the algorithm in Figure 1-8, determine the day of the week that you were born on.

7. Suppose that an algorithm was needed for determining the day of the week for dates that only occur within the years 2000–2099. Simplify the day of the week algorithm in Figure 1-8 as much as possible by making the appropriate changes.

8. As precisely as possible, give a series of steps (an algorithm) for doing long addition.

Section 1.3

9. What is the number of bits in 8 bytes, assuming the usual number of bits in a byte? **10.** Convert the following values in binary representation to base 10. *Show all steps.* **(a)** 1010 **(b)** 1011 **(c)** 10000 **(d)** 1111

Chapter Exercises 35

11. Convert the following values into binary (base 2) representation. *Show all steps.*

(a) 5 **(b)** 7 **(c)** 16 **(d)** 15 **(e)** 32

(f) 33 **(g)** 64 **(h)** 63 **(i)** 128 **(j)** 127

12. What is in common within each of the following groups of binary numbers?

(a) values that end with a “0” digit (e.g., 1100)

(b) values that end with a “1” digit (e.g., 1101)

(c) values with a leftmost digit of “1,” followed by all “0s” (e.g., 1000)

(d) values consisting only of all “1” digits (e.g., 1111)

13. Assuming that Moore’s Law continues to hold true, where n is the number of transistors that can currently be placed on an integrated circuit (chip), and $k \cdot n$ is the number that can be placed on a chip in eight years, what is the value of k ?

Section 1.4

14. Give two specific examples of an application program besides those mentioned in the chapter.

15. For each of the following statements in English, indicate whether the statement contains a syntax error, a logic (semantic) error, or is a valid statement.

(a) Witch way did he go?

(b) I think he went over their.

(c) I didn’t see him go nowhere.

16. For each of the following arithmetic expressions for adding up the integers 1 to 5, indicate whether the expression contains a syntax error, a semantic error, or is a valid expression.

(a) 1 12 113 14 15

(b) 1 12 14 15

(c) 1 12 13 14 15

(d) 5 14 1 3 12 11

17. Give one benefit of the use of a compiler, and one benefit of the use of an interpreter.

Section 1.5

18. Use the Python Interactive Shell to calculate the number of routes that can be taken for the Traveling Salesman problem for:

(a) 6 cities **(b)** 12 cities **(c)** 18 cities **(d)** 36 cities

19. Enter the following statement into the interactive shell:

```
print("What is your favorite color?")
```

Record the output. Now enter the following statement exactly as given,

```
printt("What is your favorite color?")
```

Record the output. Is this a syntax error or a logic error?

20. For the Traveling Salesman problem,

(a) Update the list representation of the distances between cities in the table in Figure 1-23 to add the city of Seattle. The distances between Seattle and each of

the other cities is given below.

Atlanta to Seattle, 2641 miles, Boston to Seattle, 3032 miles, Chicago to Seattle, 2043 miles, LA to Seattle, 1208 miles, NYC to Seattle, 2832 miles, San Francisco to Seattle, 808 miles

(b) Determine a reasonably short route of travel for visiting each city once and only once, starting in Atlanta and ending in San Francisco.

Section 1.6

21. Which of the following capabilities does an integrated development environment (IDE) provide? **(a)** Creating and modifying programs

(b) Executing programs

(c) Debugging programs

(d) All of the above

22. The Python shell is a window in which Python instructions are immediately executed. (TRUE/FALSE)

23. Suppose that the math module of the Python Standard Library were imported. What would be the proper syntax for calling a function in the math module named sqrt to calculate the square root of four?

24. What is the value of variable n after the following instructions are executed?

j 5 5

k 5 10

n 5 j * k

25. Which of the following is a proper arithmetic expression in Python?

(a) 10(15 1 6)

(b) (10 * 2)(4 1 8)

(c) 5 * (6 - 2)

26. Exactly what is output by the following if the user enters 24 in response to the input prompt. age 5 input('How old are you?: ')

print('You are', age, 'years old')

PYTHON PROGRAMMING EXERCISES

P1. Write a simple Python program that displays the following powers of 2, one per line: 2 2 2 2 2 2 2 2

P2. Write a Python program that allows the user to enter any integer value, and displays the value of 2 raised to that power. Your program should function as

shown below.

What power of two? 10

Two to the power of 10 is 1024

P3. Write a Python program that allows the user to enter any integer base and integer exponent, and displays the value of the base raised to that exponent. Your program should function as shown below.

What base? 10

What power of 10 ? 4

10 to the power of 4 is 10000

P4. Write a Python program that allows the user to enter a four-digit binary number and displays its value in base 10. *Each binary digit should be entered one per line, starting with the leftmost digit*, as shown below.

Enter leftmost digit: 1

Enter the next digit: 0

Enter the next digit: 0

Enter the next digit: 1

The value is 9

Program Development Problems 37

P5. Write a simple Python program that prompts the user for a certain number of cities for the Traveling Salesman problem, and displays the total number of possible routes that can be taken. Your program should function as shown below.

How many cities? 10

For 10 cities, there are 3628800 possible routes

PROGRAM MODIFICATION PROBLEMS M1. Modify the sample “hello” Python program in section 1.6.5 to first request the user’s first name, and then request their last name. The program should then display,

Hello *firstname lastname*

Welcome to Python!

for the *firstname* and *lastname* entered.

M2. Modify the Drake’s Equation Program in section 1.7 so that it calculates results for a best case scenario, that is, so that factors p (percentage of stars that have planets), f (percentage of those planets that develop life), i (percentage of

those planets that develop intelligent life), and c (percentage of those planets that can communicate with us) are all hard-coded as 100%. The value of R should remain as 7. Design the program so that the only values that the user is prompted for are how many planets per star can support life, n , and the estimated number of years civilizations last, L . Develop a set of test cases for your program with the included test results.

PROGRAM DEVELOPMENT PROBLEMS

D1. Develop and test a program that allows the user to enter an integer value indicating the number of cities to solve for the Traveling Salesman problem. The program should then output the number of years it would take to solve using a brute force-approach. Make use of the factorial function of the math module as shown in Figure 1-28. Estimate the total amount of time it takes by using the assumptions given in section 1.1.2.

D2. Based on the information provided about the game of chess in section 1.1.2, develop and test a program that determines how many years it would take for all possible chess games to be played if everyone in the world (regardless of age) played one (unique) chess game a day. Assume the current world population to be 7 billion.

CHAPTER 2 Data and Expressions

With this chapter, we begin a detailed discussion of the concepts and techniques of computer programming. We start by looking at issues related to the representation, manipulation, and input/ output of data—fundamental to all computing.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to: ♦

Explain and use numeric and string literal values

♦ Explain the limitations in the representation of floating-point values ♦ Explain what a character-encoding scheme is

♦ Explain what a control character is

♦ Explain and use variables, identifiers, and keywords

♦ Describe and perform variable assignment

♦ Describe and use operators and expressions

♦ Describe and use operator precedence and operator associativity ♦ Define a

data type, and explain type coercion vs. type conversion ♦ Explain the difference between static and dynamic typing ♦ Effectively use arithmetic expressions in Python

♦ Write a simple straight-line Python program

♦ Explain the importance and use of test cases in program testing

CHAPTER CONTENTS Motivation

Fundamental Concepts 2.1 Literals

2.2 Variables and Identifiers 2.3 Operators

38

2.4 Expressions and Data Types Computational Problem Solving

2.5 Age in Seconds Program

MOTIVATION

The generation, collection, and analysis of data is a driving force in today's world. The sheer amount of data being created is staggering. Chain stores generate *terabytes* (see Figure 2-1) of customer information, looking for shopping patterns of individuals. Facebook users have created 40 billion photos requiring more than a *petabyte* of storage. A certain radio telescope is expected to generate an *exabyte* of information every four hours. All told, the current amount of data created each year is estimated to be almost two *zettabytes*, more than doubling every two years. In this chapter, we look at how data is represented and operated on in Python.

Term†	Size (bytes)		Equivalent Storage
Kilobyte (KB) Kibibyte	10^3 2^{10}	1,000 1,024	Typewritten page†† (2 KB)
Megabyte (MB) Mebibyte	10^6 2^{20}	1,000,000 1,048,576	A small novel†† (1 MB)
Gigabyte (GB) Gibibyte	10^9 2^{30}	1,000,000,000 1,073,741,824	A pickup truck load of books†† (1 GB)
Terabyte (TB) Tibibyte	10^{12} 2^{40}	1,000,000,000,000 1,099,511,627,776	An academic research library†† (2 TB)
Petabyte (PB) Pebibyte	10^{15} 2^{50}	1,000,000,000,000,000 1,125,899,906,842,624	All U.S. academic libraries†† (2 PB)
Exabyte (EB) Exbibyte	10^{18} 2^{60}	1,000,000,000,000,000,000 1,152,921,504,606,846,976	All words ever spoken†† (5 EB)
Zettabyte (ZB) Zebibyte	10^{21} 2^{70}	1,000,000,000,000,000,000,000 1,180,591,620,717,411,303,424	Amount of data produced each year†††
† Because of inconsistencies in the definition of Megabyte, Gigabyte, etc., the International Organization for Standards (ISO) has recommended the use of the terms Kilobyte, Megabyte, Gigabyte, etc. for 10^3 , 10^6 , 10^9 etc., and Kibibyte, Mebibyte and Gibibyte for 2^{10} , 2^{20} , 2^{30} , etc. †† School of Information Management and Systems, University of California Berkeley http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/ ††† The fifth annual IDC Digital Universe study, 2011 http://bit.ly/lbCBCJ			

FIGURE 2-1 Measurements of Data Size (bytes)

FUNDAMENTAL CONCEPTS

2.1 Literals

2.1.1 What Is a Literal?

To take something literally is to take it at “face value.” The same is true of *literals* in programming. A **literal** is a sequence of one or more characters that stands for itself, such as the literal 12. We look at numeric literals in Python next.

A **literal** is a sequence of one or more characters that stands for itself.

2.1.2 Numeric Literals

A **numeric literal** is a literal containing only the digits 0–9, an optional sign character (1 or 2), and a possible decimal point. (The letter *e* is also used in exponential notation, shown in the next subsection). If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or “**float**” (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10). *Commas are never used in*

numeric literals. Figure 2-2 gives additional examples of numeric literals in Python.

Numeric Literals						
integer values	floating-point values					incorrect
5	5.	5.0	5.125	0.0005	5000.125	5,000.125
2500	2500.	2500.0	2500.125			2,500 2,500.125
+2500	+2500.	+2500.0	+2500.125			+2,500 +2,500.125
-2500	-2500.	-2500.0	-2500.125			-2,500 -2,500.125

FIGURE 2-2 Numeric Literals in Python

Since numeric literals without a provided sign character denote positive values, an explicit positive sign character is rarely used. Next we look at how numeric values are represented in a computer system.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
... 1024 ... 21024 ... .1024 ??? ??? ???
... 1,024 ... 0.1024 ... 1,024.46 ??? ??? ???
```

A **numeric literal** is a literal containing only the digits 0–9, a sign character (1 or 2) and a possible decimal point. Commas are never used in numeric literals.

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating-point values, however, have both a limited *range* and a limited *precision*. Python uses a double-precision standard format (IEEE 754) providing a range of 10^{2308} to 10^{308} with 16 to 17 digits of precision. To denote such a range of values, floating-points can be represented in scientific notation,

```
9.0045602e 15 (9.0045602 , 8 digits of precision)
1.006249505236801e8 (1.006249505236801 , 16 digits of precision)
4.239e216 (4.239 , 4 digits of precision)
```

It is important to understand the limitations of floating-point representation. For example, the multiplication of two values may result in **arithmetic overflow**, a condition that occurs when a calculated result is too large in magnitude (size) to

be represented,

```
... 1.5e200 * 2.0e210
... inf
```

This results in the special value `inf` (“infinity”) rather than the arithmetically correct result `3.0e410`, indicating that arithmetic overflow has occurred. Similarly, the division of two numbers may result in **arithmetic underflow**, a condition that occurs when a calculated result is too small in magnitude to be represented,

```
... 1.0e2300 / 1.0e100 0.0
```

This results in `0.0` rather than the arithmetically correct result `1.0e2400`, indicating that arithmetic underflow has occurred. We next look at possible effects resulting from the limited precision in floating-point representation.

LET’S TRY IT

From the Python Shell, enter the following and observe the results.

```
... 1.2e200 * 2.4e100 ... 1.2e200 / 2.4e100 ??? ???
... 1.2e200 * 2.4e200 ... 1.2e2200 / 2.4e200 ??? ???
```

Arithmetic overflow occurs when a calculated result is too large in magnitude to be represented. **Arithmetic underflow** occurs when a calculated result is too small in magnitude to be represented.

Limits of Precision in Floating-Point Representation

Arithmetic overflow and arithmetic underflow are relatively easily detected. The loss of precision that can result in a calculated result, however, is a much more subtle issue. For example, $1/3$ is equal to the infinitely repeating decimal `.33333333 . . .`, which also has repeating digits in base two, `.010101010. . .`. Since any floating-point representation necessarily contains only a finite number of digits, what is stored for many floating-point values is only an *approximation* of the true value, as can be demonstrated in Python,

```
... 1/3
... .3333333333333333
```

Here, the repeating decimal ends after the 16th digit. Consider, therefore, the following,

```
... 3 * (1/3) 1.0
```

Given the value of $1/3$ above, we would expect the result to be .9999999999999999, so what is happening here? The answer is that Python displays a *rounded* result to keep the number of digits displayed manageable. However, the representation of $1/3$ as .3333333333333333 remains the same, as demonstrated by the following,

```
... 1/3 1 1/3 1 1/3 1 1/3 1 1/3 1 1/3 1.9999999999999998
```

In this case we get a result that reflects the representation of $1/3$ as an approximation, since the last digit is 8, and not 9. However, if we use multiplication instead, we again get the rounded value displayed,

```
... 6 * (1/3) 2.0
```

The bottom line, therefore, is that no matter how Python chooses to display calculated results, the value stored is limited in both the range of numbers that can be represented and the degree of precision. For most everyday applications, this slight loss in accuracy is of no practical concern. However, in scientific computing and other applications in which precise calculations are required, this is something that the programmer must be keenly aware of.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... 1/10 ... 6 * (1/10) ??? ???  
... 1/10 1 1/10 1 1/10 ... 6 * 1/10 ??? ???  
... 10 * (1/10)  
???
```

Since any floating-point representation contains only a finite number of digits, what is stored for many floating-point values is only an *approximation* of the true value.

Built-in **format** Function

Because floating-point values may contain an arbitrary number of decimal places, the built-in **format** function can be used to produce a numeric string version of the value containing a specific number of decimal places,

```
... 12/5 ... 5/7  
2.4 0.7142857142857143 ... format(12/5, '.2f') ... format(5/7, '.2f')  
'2.40' '0.71'
```

In these examples, *format specifier* '.2f' rounds the result to two decimal places of accuracy in the string produced. For very large (or very small) values 'e' can be used as a format specifier,

```
... format(2 ** 100, '.6e') '1.267651e 130'
```

In this case, the value is formatted in scientific notation, with six decimal places of precision. Formatted numeric string values are useful when displaying results in which only a certain number of decimal places need to be displayed,

without use of format specifier

```
... tax 5 0.08
... print('Your cost: $', (1 + tax) * 12.99) Your cost: $ 14.029200000000001
```

with use of
format specifier

```
... print('Your cost: $', format((1 + tax) * 12.99, '.2f')) Your cost: $ 14.03
Finally, a comma in the format specifier adds comma separators to the result,
... format(13402.25, ',.2f') '13,402.24'
```

We will next see the use of format specifiers for formatting string values as well. LET'S TRY IT From the Python Shell, enter the following and observe the results. ... format(11/12, '.2f') ... format(11/12, '.2e') ??? ???

```
... format(11/12, '.3f') ... format(11/12, '.3e') ??? ???
```

The built-in **format** function can be used to produce a numeric string of a given floating-point value *rounded* to a specific number of decimal places.

2.1.3 String Literals

Numerical values are not the only literal values in programming. **String literals**, or “**strings**,” represent a sequence of characters, 'Hello' 'Smith, John' "Baltimore, Maryland 21210"

In Python, string literals may be *delimited* (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line (except when delimited by triple quotes, discussed in Chapter 7). We have already seen the use of strings in Chapter 1 for displaying screen output,

```
... print('Welcome to Python!') Welcome to Python!
Additional examples of string literals are given in Figure 2-3.
```

<code>'A'</code>	- a string consisting of a single character
<code>'jsmith16@mycollege.edu'</code>	- a string containing non-letter characters
<code>"Jennifer Smith's Friend"</code>	- a string containing a single quote character
<code>' '</code>	- a string containing a single blank character
<code>''</code>	- the empty string

FIGURE 2-3 String Literal Values

As shown in the figure, a string may contain zero or more characters, including letters, digits, special characters, and blanks. A string consisting of only a pair of matching quotes (with nothing in between) is called the **empty string**, which is different from a string containing only blank characters. Both blank strings and the empty string have their uses, as we will see. Strings may also contain quote characters as long as different quotes are used to delimit the string,

`"Jennifer Smith's Friend"`

If this string were delimited with single quotes, the apostrophe (single quote) would be considered the matching closing quote of the opening quote, leaving the last final quote unmatched, `'Jennifer Smith's Friend'` ... matching quote?

Thus, Python allows the use of more than one type of quote for such situations. (The convention used in the text will be to use single quotes for delimiting strings, and only use double quotes when needed.)

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... print('Hello') ... print('Hello') ... print('Let's Go') ??? ??? ???
```

```
... print("Hello") ... print("Let's Go!") ... print("Let's go!") ??? ??? ???
```

A **string literal**, or **string**, is a sequence of characters denoted by a pair of matching single or double (and sometimes triple) quotes in Python.

The Representation of Character Values

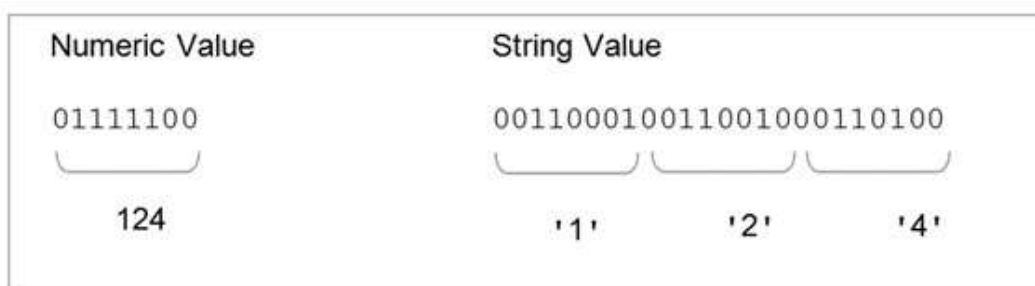
There needs to be a way to encode (represent) characters within a computer. Although various encoding schemes have been developed, the **Unicode** encoding scheme is intended to be a universal encoding scheme. Unicode is actually a collection of different encoding schemes utilizing between 8 and 32 bits for each character. The default encoding in Python uses **UTF-8**, an 8-bit encoding compatible with ASCII, an older, still widely used encoding scheme.

Currently, there are over 100,000 Unicode-defined characters for many of the languages around the world. Unicode is capable of defining more than 4 billion characters. Thus, all the world's languages, both past and present, can potentially be encoded within Unicode. A partial listing of the ASCII-compatible UTF-8 encoding scheme is given in Figure 2-4.

Space	00100000	32	A	01000001	65
!	00100001	33	B	01000010	66
"	00100010	34	C	01000011	67
#	00100011	35	.		
.			.		
.			Z	01011010	90
0	00110000	48	a	01100001	97
1	00110001	49	b	01100010	98
2	00110010	50	c	01100011	99
.			.		
.			.		
9	00111001	57	z	01111010	122

4 Partial UTF-8 (ASCII) Code Table

UTF-8 encodes characters that have an ordering with sequential numerical values. For example, 'A' is encoded as 01000001 (65), 'B' is encoded as 01000010 (66), and so on. This is true for character digits as well, '0' is encoded as 00110000 (48) and '1' is encoded as 00110001 (49). This underscores the difference between a numeric representation (that can be used in arithmetic calculations) vs. a number represented as a string of digit characters (that cannot), as demonstrated in Figure 2-5.



FIGURE

2-5 Numeric vs. String Representation of Digits

Python has means for converting between a character and its encoding. The `ord` function gives the UTF-8 (ASCII) encoding of a given character. For example,

`ord('A')` is 65. The `chr` function gives the character for a given encoding value, thus `chr(65)` is 'A'. (Functions are discussed in Chapter 5.) While in general there is no need to know the specific encoding of a given character, there are times when such knowledge can be useful.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
... ord('1') ... chr(65) ... chr(97) ??? ??? ??? ... ord('2') ... chr(90) ... chr(122) ???  
??? ???
```

Unicode is capable of representing over 4 billion different characters, enough to represent the characters of all languages, past and present. Python's (default) character encoding uses **UTF-8**, an eight-bit encoding that is part of the Unicode standard.

2.1.4 Control Characters

Control characters are special characters that are not displayed on the screen. Rather, they *control* the display of output (among other things). Control characters do not have a corresponding keyboard character. Therefore, they are represented by a combination of characters called an *escape sequence*.

An **escape sequence** begins with an **escape character** that causes the sequence of characters following it to “escape” their normal meaning. The backslash (`\`) serves as the escape character in Python. For example, the escape sequence `'\n'`, represents the *newline control character*, used to begin a new screen line. An example of its use is given below,

```
print('Hello\nJennifer Smith')
```

which is displayed as follows,

Hello

Jennifer Smith

Further discussion of control characters is given in the Python 3 Programmers' Reference.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... print('Hello World') ... print('Hello\nWorld') ??? ???  
... print('Hello World\n') ... print('Hello\n\nWorld') ??? ???  
... print('Hello World\n\n') ... print(1, '\n', 2, '\n', 3) ??? ???
```

```
... print('\nHello World') ... print('\n', 1, '\n', 2, '\n', 3) ??? ???
```

Control characters are nonprinting characters used to *control* the display of output (among other things). An **escape sequence** is a string of one or more characters used to denote control characters.

2.1.5 String Formatting

We saw above the use of built-in function `format` for controlling how numerical values are displayed. We now look at how the `format` function can be used to control how strings are displayed. As given above, the `format` function has the form,

```
format(value, format_specifier)
```

where *value* is the value to be displayed, and *format_specifier* can contain a combination of formatting options. For example, to produce the string 'Hello' left-justified in a field width of 20 characters would be done as follows,

```
format('Hello', '<20') → 'Hello '
```

To right-justify the string, the following would be used,

```
format('Hello', '>20') → ' Hello'
```

Formatted strings are left-justified by default. To center the string the '^' character is used: `format('Hello', '^20')`. Another use of the `format` function is to create strings of blank characters, which is sometimes useful,

```
format(' ', '30') → ' '
```

Finally blanks, by default, are the *fill character* for formatted strings. However, a specific fill character can be specified as shown below,

```
... print('Hello World', format('.', '.<30'), 'Have a Nice Day!') Hello World  
..... Have a Nice Day!
```

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... print(format('Hello World', '^40'))
```

```
???
```

```
... print(format('2', '2<20'), 'Hello World', format('2', '2>20')) ???
```

Built-in function **format** can be used to control how strings are displayed.

2.1.6 Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters. There are two ways in Python to deal with such situations—implicit and explicit line joining. We discuss this next.

Implicit Line Joining

There are certain delimiting characters that allow a *logical* program line to span more than one physical line. This includes matching parentheses, square brackets, curly braces, and triple quotes. For example, the following two program lines are treated as one logical line,

```
print('Name:', student_name, 'Address:', student_address, 'Number of Credits:',  
total_credits, 'GPA:', current_gpa)
```

Matching quotes (except for triple quotes, covered later) must be on the same physical line. For example, the following will generate an error,

```
print("This program will calculate a restaurant tab for a couple with a gift  
certificate, and a restaurant tax of 3%")
```

We will use this aspect of Python throughout the book.

Matching parentheses, square brackets, and curly braces can be used to span a *logical* program line on more than one physical line.

Explicit Line Joining

In addition to implicit line joining, program lines may be explicitly joined by use of the backslash (\) character. Program lines that end with a backslash that are not part of a literal string (that is, within quotes) continue on the following line,

```
numsecs_1900_dob 5 ((year_birth 2 1900) * avg_numsecs_year) 1 \  
((month_birth 2 1) * avg_numsecs_month) 1 \ (day_birth * numsecs_day)
```

Program lines may be explicitly joined by use of the backslash (\).

2.1.7 Let's Apply It—"Hello World Unicode Encoding"

It is a long tradition in computer science to demonstrate a program that simply displays "Hello World!" as an example of the simplest program possible in a particular programming language. In Python, the complete Hello World program is comprised of one program line,

```
print('Hello World!')
```

We take a twist on this tradition and give a Python program that displays the

Unicode encoding for each of the characters in the string “Hello World!” instead. This program utilizes the following programming features:

➤ string literals ➤ print ➤ ord function

The program and program execution are given in Figure 2-6.

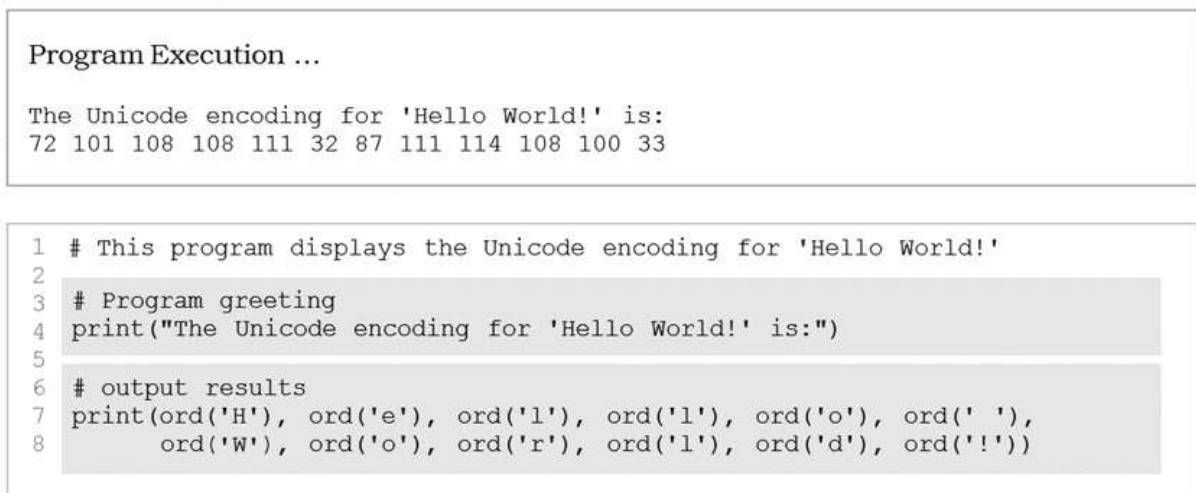


FIGURE 2-6 Hello World Unicode Encoding Program

The statements on **lines 1** , **3**, and **6** are comment statements, introduced in Chapter 1. They are ignored during program execution, used to provide information to those reading the program. The print function on **line 4** displays the message ‘Hello World!’. Double quotes are used to delimit the corresponding string, since the single quotes within it are to be taken literally. The use of print on **line 7** prints out the Unicode encoding, one-by-one, for each of the characters in the “Hello World!” string. Note from the program execution that there is a Unicode encoding for the blank character (32), as well as the exclamation mark (33).

Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.

(a)1024 (b)1,024 (c)1024.0 (d)0.25 (e).45 (f)0.25 110

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.

(a)1.89345348392e 1301 (c)2.0424e2320

(b)1.62123432632322e1300 (d)1.323232435342327896452e2140

3. Which of the following would result in either overflow or underflow for the

floating-point representation scheme mentioned in the chapter.

(a) $6.25 \times 10^{1240} * 1.24 \times 10^{110}$ (c) $6.25 \times 10^{1240} / 1.24 \times 10^{110}$

(b) $2.24 \times 10^{1240} * 1.45 \times 10^{1300}$ (d) $2.24 \times 10^{2240} / 1.45 \times 10^{1300}$

4. Exactly what is output by `print(format(24.893952, '.3f'))`

(a) 24.894 (b) 24.893 (c) 2.48e1

5. Which of the following are valid string literals in Python.

(a) "Hello" (b) 'hello' (c) "Hello' (d) 'Hello there' (e) "

6. Which of the following results of the `ord` and `chr` functions are correct? (a) `ord('1') → 49` (b) `chr(68) → 'd'`

(c) `chr(99) → 'c'` 7. How many lines of screen output is displayed by the following,

`print('apple\nbanana\ncherry\npeach')` (a) 1 (b) 2 (c) 3 (d) 4

ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow), 4. (a) 5. (a,b,d,e), 6. (a,c), 7. (d)

2.2 Variables and Identifiers

So far, we have only looked at literal values in programs. However, the true usefulness of a computer program is the ability to operate on *different* values each time the program is executed. This is provided by the notion of a *variable*. We look at variables and identifiers next.

2.2.1 What Is a Variable?

A **variable** is a name (identifier) that is associated with a value, as for variable `num` depicted in Figure 2-7.

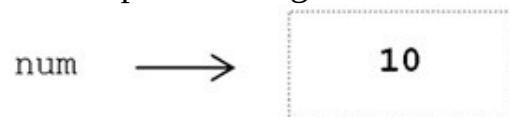


FIGURE 2-7 Program Variable

A variable can be assigned different values during a program's execution—hence, the name “variable.” Wherever a variable appears in a program (except on the left-hand side of an assignment statement), *it is the value associated with the variable that is used*, and not the variable's name,

`num 1 1 → 10 1 1 → 11`

Variables are assigned values by use of the **assignment operator**, `=`,
`num = 10` `num = 5` `num = 1`

Assignment statements often look wrong to novice programmers.

Mathematically, `num = 5` `num = 1` `num = 1` does not make sense. In computing, however, it is used to increment the value of a given variable by one. It is more appropriate,

therefore, to think of the = symbol as an arrow symbol, as shown in Figure 2-8.

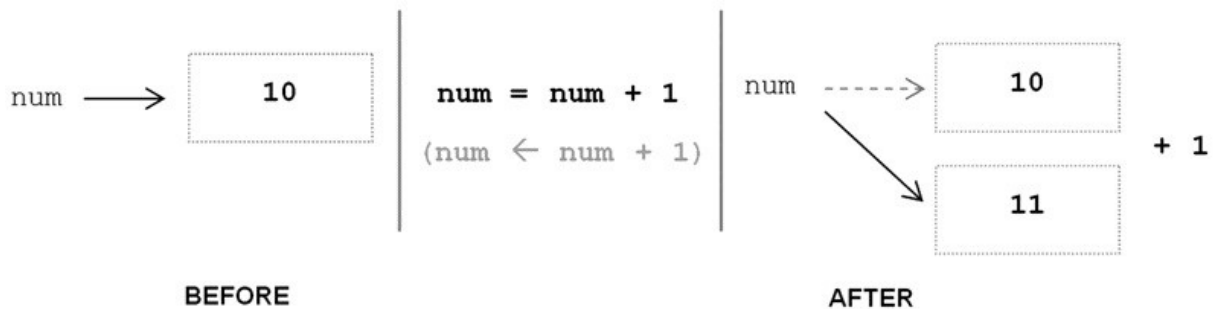


FIGURE 2-8 Variable Update

When thought of this way, it makes clear that *the right side of an assignment is evaluated first, then the result is assigned to the variable on the left*. An arrow symbol is not used simply because there is no such character on a standard computer keyboard. Variables may also be assigned to the value of another variable (or expression, discussed below) as depicted in Figure 2-9.

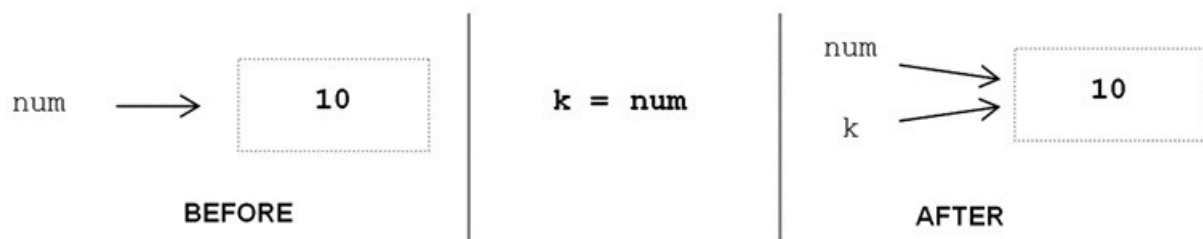


FIGURE 2-9 Variable Assignment (to another variable)

Variables `num` and `k` are both associated with the same literal value 10 in memory. One way to see this is by use of built-in function `id`,
`... id(num) ... id(k)`
 505494040 505494040

The `id` function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems. Specifically, if the value of `num` changed, would variable `k` change along with it? This cannot happen in this case because the variables refer to integer values, and integer values are *immutable*. An **immutable value** is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned, as depicted in Figure 2-10.



FIGURE 2-10 Variable Reassignment

If no other variable references the memory location of the original value, the memory location is *deallocated* (that is, it is made available for reuse).

Finally, in Python the same variable can be associated with values of different type during program execution, as indicated below.

```
var 5 12 integer
var 5 12.45 float
var 5 'Hello' string
```

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... num 510 ...k 530 ...num ...k
??? ???
...id(num) ...num ??? ???

... id(k) ...num 520 ???
...num ...id(num) ??? ???
...id(num)
??? ... k 5 k 1 1

... k ... k 5 num ???
... k ... id(num) ??? ???
... id(k) ... id(k) ??? ???
... id(num)
???
```

A **variable** is a name that is associated with a value. The **assignment operator**, `=`, is used to assign values to variables. An **immutable value** is a value that cannot be changed.

2.2.2 Variable Assignment and Keyboard Input

The value that is assigned to a given variable does not have to be specified in the program, as demonstrated in previous examples. The value can come from the user by use of the input function introduced in Chapter 1,

```
... name = input('What is your first name?')
What is your first name? John
```

In this case, the variable name is assigned the string 'John'. If the user hit return without entering any value, name would be assigned to the empty string ('').

All input is returned by the input function as a string type. *For the input of numeric values, the response must be converted to the appropriate type.* Python provides built-in **type conversion functions** **int()** and **float()** for this purpose, as shown below for a gpa calculation program,

```
line 5 input('How many credits do you have?') num_credits = int(line)
line 5 input('What is your grade point average?') gpa = float(line)
```

Here, the entered number of credits, say '24', is converted to the equivalent integer value, 24, before being assigned to variable num_credits. For input of the gpa, the entered value, say '3.2', is converted to the equivalent floating-point value, 3.2. Note that the program lines above could be combined as follows,

```
num_credits = int(input('How many credits do you have? '))
gpa = float(input('What is your grade point average? '))
```

LET'S TRY IT From the Python Shell, enter the following and observe the results. ... num = input('Enter number: ')

```
Enter number: 5
... num = input('Enter name: ')
Enter name: John
```

```
??? ???
```

```
... num = int(input('Enter number: '))
... num = int(input('Enter name: '))
Enter number: 5
Enter name: John
??? ???
```

All input is returned by the input function as a string type. Built-in functions int() and float() can be used to convert a string to a numeric type.

2.2.3 What Is an Identifier?

An **identifier** is a sequence of one or more characters used to provide a name for

a given program element. Variable names `line`, `num_credits`, and `gpa` are each identifiers. Python is *case sensitive*, thus, `Line` is different from `line`. Identifiers may contain letters and digits, but cannot begin with a digit. The underscore character, `_`, is also allowed to aid in the readability of long identifier names. It should not be used as the *fi r s t* character, however, as identifiers beginning with an underscore have special meaning in Python.

Spaces are not allowed as part of an identifier. This is a common error since some operating systems allow spaces within file names. In programming languages, however, spaces are used to delimit (separate) distinct syntactic entities. Thus, any identifier containing a space character would be considered two separate identifiers. Examples of valid and invalid identifiers in Python are given in Figure 2-11.

Valid Identifiers	Invalid Identifiers	Reason Invalid
<code>totalSales</code>	<code>'totalSales'</code>	quotes not allowed
<code>totalsales</code>	<code>total sales</code>	spaces not allowed
<code>salesFor2010</code>	<code>2010Sales</code>	cannot begin with a digit
<code>sales_for_2010</code>	<code>_2010Sales</code>	should not begin with an underscore

FIGURE 2-11 Identifier Naming

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... spring2014SemCredits 5 15 ... spring2014-sem-credits 5 15 ??? ???
... spring2014_sem_credits 5 15 ... 2014SpringSemesterCredits 5 15 ??? ???
```

An **identifier** is a sequence of one or more characters used to name a given program element. In Python, an identifier may contain letters and digits, but cannot begin with a digit. The special underscore character can also be used.

2.2.4 Keywords and Other Predefined Identifiers in Python

A **keyword** is an identifier that has predefined meaning in a programming language. Therefore, keywords cannot be used as “regular” identifiers. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword and below,

... and 5 10

SyntaxError: invalid syntax

The keywords in Python are listed in Figure 2-12. To display the keywords, type `help()` in the Python shell, and then type keywords (type 'q' to quit).

and	as	assert	break	class	continue	def
del	elif	else	except	finally	for	from
global	if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try	while
with	yield	false	none	true		

FIGURE 2-12 Keywords in Python

There are other predefined identifiers that *can* be used as regular identifiers, but should not be. This includes `float`, `int`, `print`, `exit`, and `quit`, for example. A simple way to check whether a given identifier is a keyword in Python is given below,

```
... 'exit' in dir(__builtins__) ... 'exit_program' in dir(__builtins__) True False
LET'S TRY IT
```

From the Python Shell, enter the following and observe the results.

```
... yield 5 1000 ... print('Hello')
```

```
??? ???
```

```
... Yield 5 1000 ... print 5 10
```

```
??? ... print('Hello')
```

```
???
```

A **keyword** is an identifier that has predefined meaning in a programming language and therefore cannot be used as a “regular” identifier. Doing so will result in a syntax error.

2.2.5 Let's Apply It—“Restaurant Tab Calculation”

The program below calculates a restaurant tab for a couple based on the use of a gift certificate and the items ordered. This program utilizes the following programming features:

- variables
- keyboard input
- built-in format function ➤ type conversion functions

An example execution of the program is given in Figure 2-13.

Program Execution ...

```
This program will calculate a restaurant tab for a couple with a gift
certificate, with a restaurant tax of 8.0 %
Enter the amount of the gift certificate: 200
Enter ordered items for person 1
Appetizer: 5.50
Entree: 21.50
Drinks: 4.25
Dessert: 6.00

Enter ordered items for person 2
Appetizer: 6.25
Entree: 18.50
Drinks: 6.50
Dessert: 5.50

Ordered items: $ 74.00
Restaurant tax: $ 5.92
Tab: $ -120.08
(negative amount indicates unused amount of gift certificate)
```

FIGURE 2-13 Execution of the Restaurant Tab Calculation Program

The program is given in Figure 2-14. **Lines 1–2** contain comment lines describing what the program does. The remaining comment lines provide an outline of the basic program sections. **Line 5** provides the required initialization of variables in the program, with variable tax assigned to 8% (.08). Variable tax is used throughout the program (in **lines 9 , 35 , and 39**). Thus, if the restaurant tax needs to be altered, only this line of the program needs to be changed. (Recall that * is used to denote multiplication in Python, introduced in Chapter 1.)

Lines 8–9 display to the user what the program does. The control character \n as the last character of the print function causes a screen line to be skipped before the next line is displayed. The cost of the menu items ordered is obtained from the user in **lines 15–27**.

Lines 30 and 31 total the cost of the orders for each person, assigned to variables amt_person1 and amt_person2. **Lines 34 and 35** compute the tab, including tax (stored in variable tab). Finally, **lines 38–41** display the cost of the ordered items, followed by the added restaurant tax and the amount due after deducting the amount of the gift certificate. The customers owe any remaining amount.

```

1 # Restaurant Tab Calculation Program
2 # This program will calculate a restaurant tab with a gift certificate
3
4 # initialization
5 tax = 0.08
6
7 # program greeting
8 print('This program will calculate a restaurant tab for a couple with')
9 print('a gift certificate, with a restaurant tax of', tax * 100, '%\n')
10
11 # get amount of gift certificate
12 amt_certificate = float(input('Enter amount of the gift certificate: '))
13
14 # cost of ordered items
15 print('Enter ordered items for person 1')
16
17 appetizer_per1 = float(input('Appetizier: '))
18 entree_per1 = float(input('Entree: '))
19 drinks_per1 = float(input('Drinks: '))
20 dessert_per1 = float(input('Dessert: '))
21
22 print('\nEnter ordered items for person 2')
23
24 appetizer_per2 = float(input('Appetizier: '))
25 entree_per2 = float(input('Entree: '))
26 drinks_per2 = float(input('Drinks: '))
27 dessert_per2 = float(input('Dessert: '))
28
29 # total items
30 amt_person1 = appetizer_per1 + entree_per1 + drinks_per1 + dessert_per1
31 amt_person2 = appetizer_per2 + entree_per2 + drinks_per2 + dessert_per2
32
33 # compute tab with tax
34 items_cost = amt_person1 + amt_person2
35 tab = items_cost + items_cost * tax
36
37 # display amount owe
38 print('\nOrdered items: $', format(items_cost, '.2f'))
39 print('Restaurant tax: $', format(items_cost * tax, '.2f'))
40 print('Tab: $', format(tab - amt_certificate, '.2f'))
41 print('(negative amount indicates unused amount of gift certificate)')

```

FIGURE 2-14 Restaurant Tab Calculation Program

A negative amount indicates the amount left on the gift certificate. Built-in function format is used to limit the output to two decimal places.

Self-Test Questions

- Which of the following are valid assignment statements, in which only variable k has already been assigned a value?
(a) n 5k 11 **(b)** n 5n 11 **(c)** n 1k 510 **(d)** n 11 51
- What is the value of variable num after the following assignment statements are executed? num 5 0

```
num 5 num 1 1
num 5 num 1 5
```

3. Do variables num and k reference the same memory location after the following instructions are executed? (YES/NO)

```
num 5 10
k 5 num
num 5 num 1 1
```

4. Which of the following are valid identifiers in Python?

(a)errors (b)error_count (c) error-count

5. Which of the following are keywords in Python?

(a)and (b)As (c)while (d)until (e)NOT

6. Which one of the following is correct for reading and storing an integer value from the user?

(a)n 5int_input('Enter: ') (b)n 5int(input('Enter: '))

ANSWERS: 1. (a), 2. 6, 3. No, 4. (a,b), 5. (a,c), 6. (b)

2.3 Operators

Now that we have used numeric and string types in Python, we look at operations that may be performed on them.

2.3.1 What Is an Operator?

An **operator** is a symbol that represents an operation that may be performed on one or more *operands*. For example, the + symbol represents the operation of addition. An **operand** is a value that a given operator is applied to, such as operands 2 and 3 in the expression 2 + 3. A **unary operator** operates on only one operand, such as the negation operator in -12. A **binary operator** operates on two operands, as with the addition operator. Most operators in programming languages are binary operators. We look at the arithmetic operators in Python next.

An **operator** is a symbol that represents an operation that may be performed on one or more **operands**. Operators that take one operand are called **unary operators**. Operators that take two operands are called **binary operators**.

2.3.2 Arithmetic Operators

Python provides the arithmetic operators given in Figure 2-15.

The 1, 2, * (multiplication) and / (division) arithmetic operators perform the usual operations. Note that the 2 symbol is used both as a unary operator (for negation) and a binary operator (for subtraction).

20 2 5 → 15 (as binary operator) 210 * 2 → 220 (as unary operator)

Arithmetic Operators		Example	Result
-x	negation	-10	-10
x + y	addition	10 + 25	35
x - y	subtraction	10 - 25	-15
x * y	multiplication	10 * 5	50
x / y	division	25 / 10	2.5
x // y	truncating div	25 // 10	2
		25 // 10.0	2.0
x % y	modulus	25 % 10	5
x ** y	exponentiation	10 ** 2	100

FIGURE

2-15 Arithmetic Operators in Python

Python also includes an exponentiation (**) operator. Integer and floating-point values can be used in both the base and the exponent,

2**4 → 16

2.5 ** 4.5 → 61.76323555016366

Python provides two forms of division. **“true” division** is denoted by a single slash, /. Thus, 25 / 10 evaluates to 2.5. **Truncating division** is denoted by a double slash, //, providing a truncated result based on the type of operands applied to. When both operands are integer values, the result is a truncated integer referred to as **integer division**. When at least one of the operands is a float type, the result is a truncated floating point. Thus, 25 // 10 evaluates to 2, while 25.0 // 10 becomes 2.0. This is summarized in Figure 2-16.

	Operands	result type	example	result
/ Division operator	int, int	float	7 / 5	1.4
	int, float	float	7 / 5.0	1.4
	float, float	float	7.0 / 5.0	1.4
// Truncating division operator	int, int	truncated int ("integer division")	7 // 5	1
	int, float	truncated float	7 // 5.0	1.0
	float, float	truncated float	7.0 // 5.0	1.0

FIGURE 2-16 Division Operators in Python

An example of the use of integer division would be to determine the number of dozen doughnuts for a given number of doughnuts. If variable numDoughnuts had a current value of 29, the number of dozen doughnuts would be calculated by,

numDoughnuts // 12 → 29 // 12 → 2

Lastly, the **modulus operator**(%) gives the remainder of the division of its operands, resulting in a cycle of values. This is shown in Figure 2-17.

Modulo 7		Modulo 10		Modulo 100	
0 % 7	0	0 % 10	0	0 % 100	0
1 % 7	1	1 % 10	1	1 % 100	1
2 % 7	2	2 % 10	2	2 % 100	2
3 % 7	3	3 % 10	3	3 % 100	3
4 % 7	4	4 % 10	4	.	.
5 % 7	5	5 % 10	5	.	.
6 % 7	6	6 % 10	6	96 % 100	96
7 % 7	0	7 % 10	7	97 % 100	97
8 % 7	1	8 % 10	8	98 % 100	98
9 % 7	2	9 % 10	9	99 % 100	99
10 % 7	3	10 % 10	0	100 % 100	0
11 % 7	4	11 % 10	1	101 % 100	1
12 % 7	5	12 % 10	2	102 % 100	2

FIGURE 2-17 The Modulus Operator

The modulus and truncating (integer) division operators are complements of each other. For example, `29 // 12` gives the number of dozen doughnuts, while `29 % 12` gives the number of leftover doughnuts (5).

LET’S TRY IT From the Python Shell, enter the following and observe the results.

```
... 10 1 35 ... 4 ** 2 ... 45 // 10.0 ??? ??? ???  
... 210 1 35 ... 45 / 10 ... 2025 % 10 ??? ??? ???  
... 4 * 2 ... 45 // 10 ... 2025 // 10 ??? ??? ???
```

The **division operator**, `/`, produces “true division” regardless of its operand types. The **truncating division operator**, `//`, produces either an integer or float truncated result based on the type of operands applied to. The **modulus operator**(`%`) gives the remainder of the division of its operands.

2.3.3 Let’s Apply It—“Your Place in the Universe”

The following program (Figure 2-18) calculates the approximate number of atoms that the average person contains, and the percentage of the universe that they comprise. This program utilizes the following programming features:

➤ floating-point scientific notation ➤ built-in format function

Program Execution ...

This program will determine your place in the universe.
Enter your weight in pounds: 150
You contain approximately 3.30e+28 atoms
Therefore, you comprise 3.30e-51 % of the universe

```
1 # Your Place in the Universe Program
2
3 # This program will determine the approximate number of atoms that a
4 # person consists of and the percent of the universe that they comprise.
5
6 # initialization
7 num_atoms_universe = 10e80
8 weight_avg_person = 70 # 70 kg (154 lbs)
9 num_atoms_avg_person = 7e27
10
11 # program greeting
12 print('This program will determine your place in the universe.')
13
14 # prompt for user's weight
15 weight_lbs = int(input('Enter your weight in pounds: '))
16
17 # convert weight to kilograms
18 weight_kg = 2.2 * weight_lbs
19
20 # determine number atoms in person
21 num_atoms = (weight_kg / 70) * num_atoms_avg_person
22 percent_of_universe = (num_atoms / num_atoms_universe) * 100
23
24 # display results
25 print('You contain approximately', format(num_atoms, '.2e'), 'atoms')
26 print('Therefore, you comprise', format(percent_of_universe, '.2e'),
27       '% of the universe')
```

FIGURE 2-18 Your Place in the Universe Program

Lines 1–4 describe the program. Needed variables `num_atoms_universe`, `weight_avg_person`, and `num_atoms_avg_person` are initialized in **lines 7–9**. The program greeting is on **line 12**. **Line 15** inputs the person's weight. **Line 18** converts the weight to kilograms for use in the calculations on **lines 21–22** which compute the desired results. Finally, **lines 25–27** display the results.

Self-Test Questions

1. Give the results for each of the following.

(a) $22 * 3$ (b) $15 \% 4$ (c) $3 ** 2$

2. Give the exact results of each of the following division operations.

(a) $5 / 4$ (b) $5 // 4$ (c) $5.0 // 4$

3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)
5. How many operands are there in the following arithmetic expression?
 $2 * 24 + 160 + 2 + 10$
(a)4 (b)3 (c)7
6. How many binary operators are there in the following arithmetic expression?
 $210 + 1 + 25 / (16 + 1 + 12)$
(a)2 (b)3 (c)4

ANSWERS: 1. (a) 26, (b) 3 (c) 9, 2. (a) 1.25 (b) 1 (c) 1.0, 3. (b), 4. no, 5. (a), 6. (b)

2.4 Expressions and Data Types

Now that we have looked at arithmetic operators, we will see how operators and operands can be combined to form expressions. In particular, we will look at how arithmetic expressions are evaluated in Python. We also introduce the notion of a *data type*.

2.4.1 What Is an Expression?

An **expression** is a combination of symbols that evaluates to a value.

Expressions, most commonly, consist of a combination of operators and operands,

$4 + 1 + (3 * k)$

An expression can also consist of a single literal or variable. Thus, 4, 3, and k are each expressions. This expression has two *subexpressions*, 4 and $(3 * k)$.

Subexpression $(3 * k)$ itself has two subexpressions, 3 and k.

Expressions that evaluate to a numeric type are called **arithmetic expressions**. A **subexpression** is any expression that is part of a larger expression.

Subexpressions may be denoted by the use of parentheses, as shown above.

Thus, for the expression $4 + 1 + (3 * 2)$, the two operands of the addition operator are 4 and $(3 * 2)$, and thus the result is equal to 10. If the expression were instead written as $(4 + 1 + 3) * 2$, then it would evaluate to 14.

Since a subexpression is an expression, any subexpression may contain subexpressions of its own, $4 + 1 + (3 * (2 + 2 + 1)) \rightarrow 4 + 1 + (3 * 1) \rightarrow 4 + 1 + 3 \rightarrow 7$

If no parentheses are used, then an expression is evaluated according to the rules of operator precedence in Python, discussed in the next section.

LET'S TRY IT From the Python Shell, enter the following and observe the results. ... (2 1 3) * 4 ... 2 + ((3 * 4) 2 8) ??? ???

... 2 1 (3 * 4) ... 2 1 3 * (4 2 1) ??? ???

An **expression** is a combination of symbols (or single symbol) that evaluates to a value. A **subexpression** is any expression that is part of a larger expression.

2.4.2 Operator Precedence

The way we commonly represent expressions, in which operators appear between their operands, is referred to as **infix notation**. For example, the expression 4 13 is in infix notation since the 1 operator appears between its two operands, 4 and 3. There are other ways of representing expressions called *prefix* and *postfix* notation, in which operators are placed *before* and *after* their operands, respectively.

The expression 4 1(3 * 5) is also in infix notation. It contains two operators, 1 and *.

The parentheses denote that (3 * 5) is a subexpression. Therefore, 4 and (3 * 5) are the operands of the addition operator, and thus the overall expression evaluates to 19. What if the parentheses were omitted, as given below?

4 1 3 * 5 How would this be evaluated? These are two possibilities, 4 1 3 * 5 → 4 15 → 19

4 1 3 * 5 → 7 * 5 → 35

Some might say that the first version is the correct one by the conventions of mathematics. However, each programming language has its own rules for the order that operators are applied, called **operator precedence**, defined in an **operator precedence table**. This may or may not be the same as in mathematics, although it typically is. In Figure 2-19, we give the operator precedence table for the Python operators discussed so far. (We will discuss the issue of associativity indicated in Figure 2-19 in the next section.)

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

FIGURE 2-19 Operator Precedence of Arithmetic Operators in Python

In the table, higher-priority operators are placed above lower-priority ones. Thus, we see that multiplication is performed before addition when no parentheses are included,

$4 + 1 * 3 * 5 \rightarrow 4 + 15 \rightarrow 19$

In our example, therefore, if the addition is to be performed first, parentheses would be needed, $(4 + 1 * 3) * 5 \rightarrow 7 * 5 \rightarrow 35$

As another example, consider the expression below. Following Python's rules of operator precedence, the exponentiation operator is applied first, then the truncating division operator, and finally the addition operator,

$4 + 1 * 2 ** 5 // 10 \rightarrow 4 + 1 * 32 // 10 \rightarrow 4 + 1 * 3 \rightarrow 7$

Operator precedence guarantees a consistent interpretation of expressions. However, it is good programming practice to use parentheses even when not needed if it adds clarity and enhances readability, without overdoing it. Thus, the previous expression would be better written as,

$4 + 1 * (2 ** 5) // 10$

LET'S TRY IT From the Python Shell, enter the following and observe the results. ... $2 + 1 * 3 * 4$... $2 * 3 // 4$??? ???

... $2 * 3 + 1 * 4$... $5 + 1 * 42 \% 10$??? ???

... $2 * 3 / 4$... $2 * 2 ** 3$??? ???

Operator precedence is the relative order that operators are applied in the evaluation of expressions, defined by a given **operator precedence table**.

2.4.3 Operator Associativity

A question that you may have already had is, "What if two operators have the same level of precedence, which one is applied first?" For operators following

the associative law, the order of evaluation doesn't matter,

$$(2 + 1 + 3) * 4 \rightarrow 9 \quad 2 + (1 + 4) \rightarrow 9$$

In this case, we get the same results regardless of the order that the operators are applied. Division and subtraction, however, do not follow the associative law,

$$(a) (8 - 2 + 4) * 2 \rightarrow 4 * 2 \rightarrow 8 \quad 2 * (4 - 2) \rightarrow 2 * 2 \rightarrow 4 \quad (b) (8 / 4) / 2 \rightarrow 2 / 2 \rightarrow 1 \quad 8 / (4 / 2) \rightarrow 8 / 2 \rightarrow 4 \quad (c) 2 ** (3 ** 2) \rightarrow 512 \quad (2 ** 3) ** 2 \rightarrow 64$$

Here, the order of evaluation does matter. To resolve the ambiguity, each operator has a specified **operator associativity** that defines the order that it and other operators with the same level of precedence are applied (as given in Figure 2-19). All operators in the figure, except for exponentiation, have left-to-right associativity—exponentiation has right-to-left associativity.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

```
... 6 2 3 1 2 ... 2 * 3 / 4 ... (2 ** 2) ** 3 ??? ??? ??? ... (6 2 3) 1 2 ... 12 % (10 / 2)
... 2 ** (2 ** 3) ??? ??? ??? ... 6 2 (3 1 2) ... 2 ** 2 ** 3
??? ???
```

Operator associativity is the order that operators are applied when having the same level of precedence, specific to each operator.

2.4.4 What Is a Data Type?

A **data type** is a set of values, and a set of operators that may be applied to those values. For example, the integer data type consists of the set of integers, and operators for addition, subtraction, multiplication, and division, among others. Integers, floats, and strings are part of a set of predefined data types in Python called the **built-in types**.

Data types prevent the programmer from using values inappropriately. For example, it does not make sense to try to divide a string by two, 'Hello' / 2. The programmer knows this by common sense. Python knows it because 'Hello' belongs to the string data type, which does not include the division operation. The need for data types results from the fact that the same internal representation of data can be interpreted in various ways, as shown in Figure 2-20.

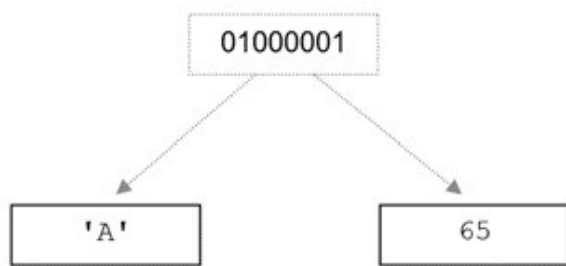


FIGURE 2-20 Multiple Interpretations

of a Sequence of Bits

The sequence of bits in the figure can be interpreted as a character ('A') or an integer (65). If a programming language did not keep track of the intended type of each value, then the programmer would have to. This would likely lead to undetected programming errors, and would provide even more work for the programmer. We discuss this further in the following section.

Finally, there are two approaches to data typing in programming languages. In **static typing**, a variable is declared as a certain type before it is used, and can only be assigned values of that type. Python, however, uses *dynamic typing*. In **dynamic typing**, the data type of a variable depends only on the type of value that the variable is currently holding. Thus, the same variable may be assigned values of different type during the execution of a program.

A **data type** is a set of values, and a set of operators that may be applied to those values.

2.4.5 Mixed-Type Expressions

A **mixed-type expression** is an expression containing operands of different type. The CPU can only perform operations on values with the same internal representation scheme, and thus only on operands of the same type. Operands of mixed-type expressions therefore must be converted to a common type. Values can be converted in one of two ways—by implicit (automatic) conversion, called *coercion*, or by explicit *type conversion*. We look at each of these next.

A **mixed-type expression** is an expression with operands of different type.

Coercion vs. Type Conversion

Coercion is the *implicit* (automatic) conversion of operands to a common type. Coercion is automatically performed on mixed-type expressions only if the operands can be safely converted, that is, if no loss of information will result. The conversion of integer 2 to floating-point 2.0 below is a safe conversion—the

conversion of 4.5 to integer 4 is not, since the decimal digit would be lost,

2 1 4.5 → 2.0 1 4.5 → 6.5 safe (automatic conversion of int to float)

Type conversion is the *explicit* conversion of operands to a specific type. Type conversion can be applied even if loss of information results. Python provides built-in **type conversion functions** `int()` and `float()`, with the `int()` function truncating results as given in Figure 2-21.

`float(2) 1 4.5 → 2.0 1 4.5 → 6.5 2 1 int(4.5) → 2 1 4 → 6`

Conversion Function		Converted Result	Conversion Function		Converted Result
<code>int()</code>	<code>int(10.8)</code>	10	<code>float()</code>	<code>float(10)</code>	10.0
	<code>int('10')</code>	10		<code>float('10')</code>	10.0
	<code>int('10.8')</code>	ERROR		<code>float('10.8')</code>	10.8

FIGURE 2-21 Conversion Functions `int()` and `float()` in Python

Note that numeric strings can also be converted to a numeric type. In fact, we have already been doing this when using `int` or `float` with the `input` function, `num_credits 5 int(input('How many credits do you have? '))`

Coercion is the *implicit* (automatic) conversion of operands to a common type.

Type conversion is the *explicit* conversion of operands to a specific type.

2.4.6 Let's Apply It—"Temperature Conversion Program"

The following Python program (Figure 2-22) requests from the user a temperature in degrees Fahrenheit, and displays the equivalent temperature in degrees Celsius. This program utilizes the following programming features:

➤ arithmetic expressions ➤ operator associativity ➤ format function

Program Execution ...

```
This program will convert degrees Fahrenheit to degrees Celsius
Enter degrees Fahrenheit: 100
100.0 degrees Fahrenheit equals 37.8 degrees Celsius
```

```
1 # Temperature Conversion Program (Fahrenheit to Celsius)
2
3 # This program will convert a temperature entered in Fahrenheit
4 # to the equivalent degrees in Celsius
5
6 # program greeting
7 print('This program will convert degrees Fahrenheit to degrees Celsius')
8
9 #get temperature in Fahrenheit
10 fahrenheit = float(input('Enter degrees Fahrenheit: '))
11
12 # calc degrees Celsius
13 celsius = (fahrenheit - 32) * 5 / 9
14
15 # output degrees Celsius
16 print(fahrenheit, 'degrees Fahrenheit equals',
17       format(celsius, '.1f'), 'degrees Celsius')
```

FIGURE 2-22 Temperature Conversion Program

Lines 1–4 contain the program description. **Line 7** provides the program greeting. **Line 10** reads the Fahrenheit temperature entered, assigned to variable `fahrenheit`. Either an integer or a floating-point value may be entered, since the input is converted to float type. **Line 13** performs the calculation for converting Fahrenheit to Celsius. Recall that the division and multiplication operators have the same level of precedence. Since these operators associate left-to-right, the multiplication operator is applied first. Because of the use of the “true” division operator `/`, the result of the expression will have floating-point accuracy. Finally, **lines 16–17** output the converted temperature in degrees Celsius.

Self-Test Questions

1. What value does the following expression evaluate to?

$219 * ((3 * 12) - 8) / 10$

(a) 27 (b) 27.2 (c) 30.8

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

(a)3 1 2 * 10 **(b)**2 1 5 * 4 1 3 **(c)**20 // 2 * 5 **(d)**2 * 3 ** 2

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.

(a)24 // 4 // 2 **(b)**2 ** 2 ** 3

4. Which of the following is a mixed-type expression?

(a)2 1 3.0 **(b)**2 1 3 * 4

5. Which of the following would involve coercion when evaluated in Python?

(a)4.0 1 3 **(b)**3.2 * 4.0

6. Which of the following expressions use explicit type conversion? **(a)**4.0 1 float(3) **(b)**3.2 * 4.0 **(c)**3.2 1 int(4.0)

ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, 3. (a) 3 (b) 256, 4. (a), 5. (a), 6. (a, c)

COMPUTATIONAL PROBLEM SOLVING 2.5 Age in Seconds Program

We look at the problem of calculating an individual's age in seconds. It is not feasible to determine a given person's age to the exact second. This would require knowing, to the second, when they were born. It would also involve knowing the time zone they were born in, issues of daylight savings time, consideration of leap years, and so forth. Therefore, the problem is to determine an *approximation* of age in seconds. The program will be tested against calculations of age from online resources.

2.5.1 The Problem

The problem is to determine the approximate age of an individual in seconds within 99% accuracy of results from online resources. The program must work for dates of birth from January 1, 1900 to the present.

2.5.2 Problem Analysis

The fundamental computational issue for this problem is the development of an algorithm incorporating approximations for information that is impractical to utilize (time of birth to the second, daylight savings time, etc.), while producing a result that meets the required degree of accuracy.

2.5.3 Program Design

Meeting the Program Requirements

There is no requirement for the form in which the date of birth is to be entered.

We will therefore design the program to input the date of birth as integer values. Also, the program will not perform input error checking, since we have not yet covered the programming concepts for this.

Data Description

The program needs to represent two dates, the user's date of birth, and the current date. Since each part of the date must be able to be operated on arithmetically, dates will be represented by three integers. For example, May 15, 1992 would be represented as follows:

year 51992 month 55 day 5 15

Hard-coded values will also be utilized for the number of seconds in a minute, number of minutes in an hour, number of hours in a day, and the number of days in a year.

Algorithmic Approach

The Python Standard Library module `datetime` will be used to obtain the current date. (See the Python 3 Programmers' Reference.) We consider how the calculations can be approximated without greatly affecting the accuracy of the results.

We start with the issue of leap years. Since there is a leap year once every four years (with some exceptions), we calculate the average number of seconds in a year over a four-year period that includes a leap year. Since non-leap years have 365 days, and leap years have 366, we need to compute,

```
numsecs_day 5 (hours per day) * (mins per hour) * (secs per minute)
numsecs_year 5 (days per year) * numsecs_day
avg_numsecs_year 5 (4 * numsecs_year) 1 numsecs_day // 4
avg_numsecs_month 5 avgnumsecs_year // 12
```

(one extra day for leap year)

To calculate someone's age in seconds, we use January 1, 1900 as a basis. Thus, we compute two values—the number of seconds from January 1 1900 to the given date of birth, and the number of seconds from January 1 1900 to the current date. Subtracting the former from the latter gives the approximate age,

1900 date of birth current date

num secs 1900 to present
num secs 1900 to date of birth^{age in seconds}

Note that if we directly determined the number of seconds between the date of birth and current date, the months and days of each would need to be compared to see how many full months and years there were between the two. Using 1900 as a basis avoids these comparisons. Thus, the rest of our algorithm is given below.

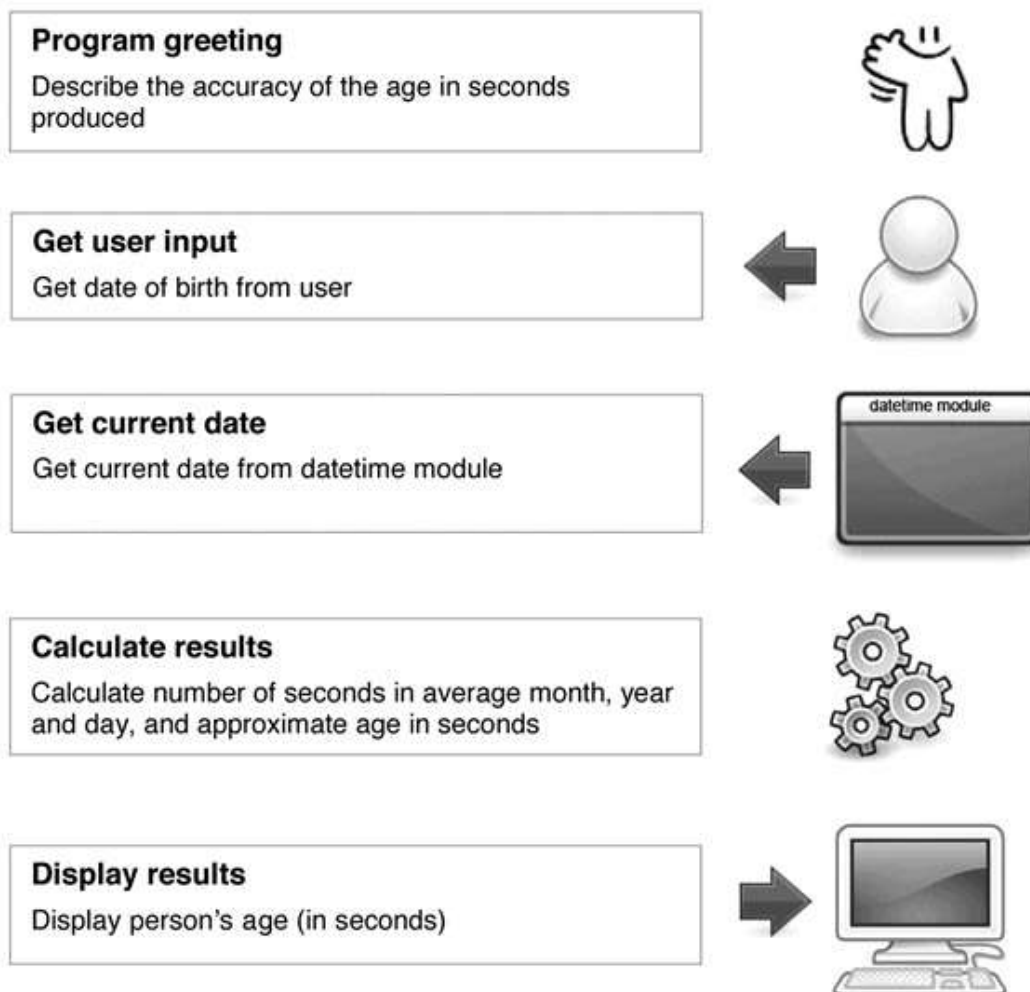
numsecs_1900_to_dob 5 (year_birth 2 1900) * avg_numsecs_year 1
(month_birth 2 1) * avg_numsecs_month 1 (day_birth * numsecs_day)

numsecs_1900_to_today 5 (current_year 2 1900) * avg_numsecs_year 1
(current_month 2 1) * avg_numsecs_month 1
(current_day * numsecs_day)

age_in_secs 5 num_secs_1900_to_today 2 numsecs_1900_to_dob

Overall Program Steps

The overall steps in this program design are in Figure 2-23.



FIGURE

2-23 Overall Steps of the Age in Seconds Program

2.5.4 Program Implementation and Testing

Stage 1—Getting the Date of Birth and Current Date

First, we decide on the variables needed for the program. For date of birth, we use variables `month_birth`, `day_birth`, and `year_birth`. Similarly, for the current date we use variables `current_month`, `current_day`, and `current_year`. The first stage of the program assigns each of these values, shown in Figure 2-24.

```

1 # Age in Seconds Program (Stage 1)
2 # This program will calculate a person's approximate age in seconds
3
4 import datetime
5
6 # Get month, day, year of birth
7 month_birth = int(input('Enter month born (1-12): '))
8 day_birth = int(input('Enter day born (1-31): '))
9 year_birth = int(input('Enter year born (4-digit): '))
10
11 # Get current month, day, year
12 current_month = datetime.date.today().month
13 current_day = datetime.date.today().day
14 current_year = datetime.date.today().year
15
16 # Test output
17 print('\nThe date of birth read is: ', month_birth, day_birth,
18       year_birth)
19
20 print('The current date read is: ', current_month, current_day,
21       current_year)

```

FIGURE 2-24 First Stage of Age in Seconds Program
Stage 1 Testing

We add test statements that display the values of the assigned variables. This is to ensure that the dates we are starting with are correct; otherwise, the results will certainly not be correct. The test run below indicates that the input is being correctly read.

```

Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born (4-digit): 1981 The date of birth read is: 4 12 1981 The current
date read is: 1 5 2010 ...

```

Stage 2—Approximating the Number of Seconds in a Year/Month/Day

Next we determine the approximate number of seconds in a given year and month, and the exact number of seconds in a day stored in variables `avg_numsecs_year`, `avg_numsecs_month`, and `numsecs_day`, respectively, shown in Figure 2-25.

```

1 # Age in Seconds Program (Stage 2)
2 # This program will calculate a person's approximate age in seconds
3
4 import datetime
5
6 ### Get month, day, year of birth
7 ##month_birth = int(input('Enter month born (1-12): '))
8 ##day_birth = int(input('Enter day born (1-31): '))
9 ##year_birth = int(input('Enter year born (4-digit): '))
10
11 ### Get current month, day, year
12 ##current_month = datetime.date.today().month
13 ##current_day = datetime.date.today().day
14 ##current_year = datetime.date.today().year
15
16 # Determine number of seconds in a day, average month, and average year
17 numsecs_day = 24 * 60 * 60
18 numsecs_year = 365 * numsecs_day
19
20 avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
21 avg_numsecs_month = avg_numsecs_year // 12
22
23 # Test output
24 print('numsecs_day ', numsecs_day)
25 print('avg_numsecs_month = ', avg_numsecs_month)
26 print('avg_numsecs_year = ', avg_numsecs_year)

```

FIGURE 2-25 Second Stage of Age in Seconds Program

The lines of code prompting for input are commented out (**lines 6–9** and **11–14**). Since it is easy to comment out (and uncomment) blocks of code in IDLE, we do so; the input values are irrelevant to this part of the program testing.

Stage 2 Testing

Following is the output of this test run. Checking online sources, we find that the number of seconds in a regular year is 31,536,000 and in a leap year is 31,622,400. Thus, our approximation of 31,557,600 as the average number of seconds over four years (including a leap year) is reasonable. The avg_num_seconds_month is directly calculated from variable avg_numsecs_year, and numsecs_day is found to be correct.

```

numsecs_day 86400
avg_numsecs_month 5 2629800
avg_numsecs_year 5 31557600
...

```

Final Stage—Calculating the Number of Seconds from 1900

Finally, we complete the program by calculating the approximate number of seconds from 1900 to both the current date and the provided date of birth. The difference of these two values gives the approximate age in seconds. The complete program is shown in Figure 2-26.

```
1 # Age in Seconds Program
2 # This program will calculate a person's approximate age in seconds
3
4 import datetime
5
6 # Program greeting
7 print('This program computes the approximate age in seconds of an')
8 print('individual based on a provided date of birth. Only dates of')
9 print('birth from 1900 and after can be computed\n')
10
11 # Get month, day, year of birth
12 month_birth = int(input('Enter month born (1-12): '))
13 day_birth = int(input('Enter day born (1-31): '))
14 year_birth = int(input('Enter year born (4-digit): '))
15
16 # Get month, day, year of birth
17 current_month = datetime.date.today().month
18 current_day = datetime.date.today().day
19 current_year = datetime.date.today().year
20
21 # Determine number of seconds in a day, average month, and average year
22 numsecs_day = 24 * 60 * 60
23 numsecs_year = 365 * numsecs_day
24
25 avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
26 avg_numsecs_month = avg_numsecs_year // 12
27
28 # Calculate approximate age in seconds
29 numsecs_1900_dob = (year_birth - 1900 * avg_numsecs_year) + \
30                   (month_birth - 1 * avg_numsecs_month) + \
31                   (day_birth * numsecs_day)
32
33 numsecs_1900_today = (current_year - 1900 * avg_numsecs_year) + \
34                     (current_month - 1 * avg_numsecs_month) + \
35                     (current_day * numsecs_day)
36
37 age_in_secs = numsecs_1900_today - numsecs_1900_dob
38
39 # output results
40 print('\nYou are approximately', age_in_secs, 'seconds old')
```

FIGURE 2-26 Final Stage of Age in Seconds Program

We develop a set of test cases for this program. We follow the testing strategy of including “average” as well as “extreme” or “special case” test cases in the test plan. The test results are given in Figure 2-27.

Date of Birth	Expected Results	Actual Results	Inaccuracy	Evaluation
January 1, 1900	3,520,023,010 \pm 86,400	1,468,917	99.96 %	failed
April 12, 1981	955,156,351 \pm 86,400	518,433	99.94 %	failed
January 4, 2000	364,090,570 \pm 86,400	1,209,617	99.64 %	failed
December 31, 2009	48,821,332 \pm 86,400	-1,123,203	102.12 %	failed
(day before current date)	86,400 \pm 86,400	86,400	0 %	passed

FIGURE 2-27 Results of First Execution of Test Plan

The “correct” age in seconds for each was obtained from an online source. January 1, 1900 was included in the test plan since it is the earliest date (“extreme case”) that the program is required to work for. April 12, 1981 was included as an average case in the 1900s, and January 4, 2000 as an average case in the 2000s. December 31, 2009 was included since it is the last day of the last month of the year. Finally, a test case for a birthday on the day before the current date was included as a special case. (See sample program execution in Figure 2-28). Since these values are continuously changing by the second, we consider any result within one day’s worth of seconds (\pm 84,000) to be an exact result.

```

This program computes the approximate age in seconds of an
individual based on a provided date of birth. Only ages for
dates of birth from 1900 and after can be computed

Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born: (4-digit)1981

You are approximately 518433 seconds old
>>>

```

FIGURE 2-28 Example Output of Final Stage Testing

The program results are obviously incorrect, since the result is approximately equal to the average number of seconds in a month (determined above). The only correct result is for the day before the current date. The inaccuracy of each result was calculated as follows for April 12, 1981,