Contents

Chapter 1

Introduction

Python is both a useful all-purpose programming language and an excellent tool for introducing the concepts of programming. In this chapter we'll see how to install and run Python, and also how to use the programming environment that comes with Python. You might just skim the last few sections of this chapter at the start and then come back to them after you have studied the material in chapters 2 through 4. Much of the advice found here will seem more relevant after you know a bit about programming.

1.1 About Python and Other Programming Languages

Although a host of science fiction movies would have it otherwise, computers don't think, they just follow simple instructions. Computers are not even sophisticated enough to understand human languages, they need their own ultra-simple languages. In the end, the instructions that computers can follow are things like "Take the numbers from this memory location and that memory location, add them together, and put result into this location." The first programming languages in the 1950's were expressed in terms very similar to these. Because it is difficult for people to think in such basic terms, computer scientists have gradually learned how to create languages that humans can easily work with, but that can be translated into the simple instructions that can be followed by a computer. These are called *programming languages*. Thousands of programming languages have been created, but a relatively small number are in common use today. In these notes we will work with one language, called *Python* that was invented in 1989 by the Dutch programmer Guido van Rossum. Unfortunately, people don't always get along, and in 2008 Python bifurcated into two similar languages, Python2 and Python3 because the language developers couldn't agree on what were mainly minor differences of syntax. In these notes we will use Python3 and we will usually refer to the language simply as *Python*.

Just to give you a sense of the variety of programming languages that are in use, here are programs from different languages that all print to the computer screen the result of multiplying 3 and 4:

```c
#include <stdio.h>
int main(void)
{
        int x, y;
        x = 3;
        y = 4;
        printf("%d\n", x*y);
        return 0;
}
```

Program 1.1.1: The C programming language

```
#include <iostream>

using namespace std;
int main()
{
        int x, y;
        x = 3;
        y=4;
        cout << x*y << endl;
        return 0;
}
```

Program 1.1.2: The C++ programming language

```
program Times
        integer x
        integer y
        x = 3
        y = 4
        print *, x*y
end program Times
```

Program 1.1.3: Fortran

```
public class Times {
   public static void main(String[] args) {
       int x, y;
       x = 3;
       y = 4;
       System.out.println(x*y);
   }
}
```

Program 1.1.4: Java

Here this program is in Python:

```
x = 3
y = 4
print( x*y )
```

Program 1.1.5: Python

Notice how much simpler this program is in Python. There are two things that Python does differently than most programming languages:

- Most languages group statements together with some type of bracket or connective terminology. The "{" and "}" symbols you see in some of these programs are examples of this. Python uses instead the way the program is laid out on the page. This makes Python programs easier to read.

- Most languages require the programmer to say what kind of data will be stored in variables. In the programs above "x" and "y" are variables – names attached to memory locations that will be used to store data. All of the examples except the one in Python have statements that say that these are "int" or "integer" variables. The Python system deduces this information from the program itself; it does not require the user to specify it.

In general, Python has less verbiage around a program and lets the programmer concentrate on the actual instructions in the code. This helps to make Python programs are easier to read, and, as we will emphasize many times in these notes, readable programs are more likely to be correct.

There are two types of programming languages. In both types the program needs to be run through a software system, because almost no one wants to program in a language that the computer can execute directly. *Compiled* languages can be converted into machine code. Programs in such languages are run through a *compiler*, a program that converts the compilable program into machine code. This produces an *executable* program – a program that runs directly on the machine. Most of the programs you have ever run, such as Microsoft Word or Mozilla Firefox, are compiled. The compiler is needed to produce the executable program, after that the executable stands on its own and can even be copied and transferred to another system; the compile is no longer needed. *Interpreted* programs are not converted to machine code; instead, they are run through a system called an *interpreter* that executes the program one statement at a time in the same way a computer would if it could understand the language the program was written in. An interpreter needs to be used every time the program is run. Python is interpreted, though there are ways to produce standalone Python programs that can be run without an interpreter. In general compiled programs run more efficiently (faster, using less memory) than interpreted programs, though machines have become so fast and memory so cheap that for many applications this is not an important distinction.

Understanding programs

Learning to read programs is an important part of learning to program. You can't read a program the way you read a book. You should read programs the same way a computer does. Every instruction does one of two things – it either changes the computer's memory, or it alters the sequence in which the statements are executed. Consider the following portion of a program:

```
x = 5
if x < 10:
    print( "small" )
else:
    print( "big" )
```

The first statement, x=5, changes the data stored in variable x (a memory location) to 5. The next line, **if** x < 10: looks into the computer's memory for the value stored in variable x (we just set that value in the previous line, but the system doesn't remember this from one statement to the next. The system doesn't *understand* the program, it just follows it one instruction at a time.) If the value in x is less than 10, the system executes the next line, which is **print**("small"); if the value in x is not less than 10, the system executes the line following **else**:, which is **print**("big"). Of course, you can do this in your head if the program only has a few lines, but we will get to more sophisticated programs quickly. When you are reading them, keep a piece of paper that represents the computer's memory. Each time a variable is given a value, write it down on your paper. Walk through the program from the starting point (you'll see where that is in section 2.1) until you run out of instructions and you will be executing the program the same way the computer does. This might seem tedious, and it is. Computers are not "smart", they are just fast and persistent. They don't get bored. Computers do sophisticated things by doing many, many simple instructions very, very quickly. In this course you will learn to write these instructions in ways that will reliably get the computer to do the things you want.

1.2 Obtaining and Installing Python

Python is a free, open source language whose interpreter runs in MS Windows, Mac OS X and Linux environments. Python is built into the Macintosh operating system, but you should install it even if you use Macs because the operating system does not have a full implementation, and what it does have is probably not current. The language is maintained (programming lanugages evolve just like human languages) by the Python Software Foundation. Guido van Rossum, the inventor of Python is the President (also known as "Benevolent Dictator for Life") of the Foundation but its board of directors also includes other prominent computer scientists and representatives of major computing organizations and corporations.

Installing Python is easy. The website for the software maintained by the Python Software Foundation is http://www.python.org. Go to this site, then to the Download tab (the tabs at present are in the upper left corner of the page). There are two dialects of Python – Python2 and Python3. These are not quite compatible, so it is important to know which one you are downloading. These notes all refer to Python3, which is referenced on the Python website as Python3.x.y, where `x` and `y` are release numbers. The current version is Python3.4.3. For these notes, any Python3 release after Python3.2.0 will suffice.

On the Downloads page for Python you are presented with options for many old versions; you should click on the Python3.4.3 link. Scroll down to the *Files* section of the Release3.4.3 page. This has about 10 different options, only a few of which are pertinent for most students:

- `Mac OS X 32-bit i386/PPC installer`
- `Mac OS X 64-bit/32-bit installer`
- `Windows x86-64 MSI installer`
- `Windows x86 MSI installer`

You may see different options when you go to this page. Making sense of this is easier than you might think. If you run Windows the last two choices are for you. You almost certainly want the last of these; the previous one is for those few machines that run a 64-bit version of Windows; you would know if this is you. So choose your link and click on it. After a few seconds this downloads the installer into your Downloads folder. Once it is downloaded, click on the installer and do what it says. There will be some questions about where to install things; the defaults are almost always good options. You will need to accept the license agreement. The whole process takes about 5 minutes if you have a good Internet connection.

If you have a Mac, you need to find out which version of OS X is installed on it. Go to the Apple menu (its icon is a picture of an apple) at the top left of the menu bar. One of the menu items is "About this Mac". Select this and a window will pop up with lots of information about the system, including which version of OS X is installed. Most likely you have 10.6 (also called "Snow

Leopard) or one of the more recent versions. Every Mac produced after August of 2009 has OS X 10.6 or higher. If this is the case with your machine, click on the second Mac link (”`Mac OS X 64-bit/32-bit installer`”) If your machine is older, click on the first Mac link. If somehow neither of these seems right, ask for help.

Just as with Windows, once you have selected the correct link, click on it to download the installer, then run the installer to install Python. There will be a few questions, and you must accept the license agreement. As with Windows, the installation process should only take a few minutes if you have a good connection.

Now that you have installed Python, you need to be able to find it. On a Mac there should be a Python folder in the Applications section of the Finder. This has several programs in it, the two most important of which are called IDLE and Python Launcher. On Windows, go to the Start window. The installer may have left an icon for Python in the Start window. If not, follow the All Programs link to a list of all of your installed programs. This should have a link to Python, again with several programs including IDLE. In the next section we will look at the ways the various Python programs can be used.

1.3 Using Python

The Python system that you have installed on your computer has several tools for developing and running Python programs. One of these is called "python" on Windows and "Python Launcher" on a Mac. This tool allows the system to run programs written in the Python language. This opens a window for the program's input and output, runs the program, and then closes this window. Think for a moment, however, about what this means. Suppose you have a program that opens up a dictionary and looks for words that contain four consecutive vowels, such as "queue". This program needs no input from the user. It opens the dictionary file, opens a window to print its output, prints that output and stops. At the end of the program it closes any files or windows it has opened. This program can run in a small fraction of a second. It does give some output, but that output goes into a window that closes before you have a chance to read it. The entire process is useless – the program has done its job, but its output has not been available for the human user.

In Chapter 8 of these notes we will look at programs that create a graphical user interface,with buttons, menus and similar "widgets". As we will see, these programs need to be run through the Python Launcher. The programs and the windows they create will persist until the user clicks a Quit button, so we will have time to examine any output from the program. For all of the work we will do prior to Chapter 8 we will use a different tool for both creating and running the program. This is called IDLE (after Eric Idle, one of the founding members of the Monty Python comedy group). IDLE can be found the way you find other programs on your computer – from the Start window's All Programs link on Windows or from the Applications folder of the Macintosh Finder. On either system look for the Python (or Python3.4) folder, and IDLE will be one of the programs inside it.

IDLE has two primary windows: an editor window and a command interpreter, or shell, window. (A command interpreter is called a "shell" because it is the interface between the internals of a program and the outside world.) The editor lets you create programs. The command interpreter will run the program that is in the editor window, and it also has a prompt for you to enter Python command that will be excecuted as you enter them. For example, if you enter

```
> 3+4*5
```

at the prompt the system will reply

```
23
```

Each of these windows has an Options menu with a Configure IDLE item that allows you to specify which window should be on top when you start IDLE. From the editor window there is a Shell item in the Run menu that brings up a shell. Alternatively, if you select the Run Module item in the Run menu the program in the edit window will be executed. If you are in a shell window and want to create a new program, from the File menu select New Window and a new editor window will be opened, into which you can type your program.

Note that you must save the program to a file (Save As from the File menu) before you can run it. Be sure you know which window you are saving – it is not uncommon for beginners to save the shell window where the program was executed. You want to save the IDLE editor window that has the code for the program.

Suppose we type the following program into the editor and save it as file Heads.py. Note that by convention Python programs are always given names that end in ".py". This suffix is the what the system uses to recognize the file as containing Python code.

```
from random import *

def Toss():
    x = randint(0, 1)
    if x == 0:
        return "Heads"
    else:
        return "Tails"

def CountHeads(N):
    count = 0
    for x in range(N):
        result = Toss()
        if result == "Heads":
            count = count + 1
    return count

def main():
    num = CountHeads(1000)
    print( "%d out of %d" %(num, 1000) )

main()
```

Program 1.3.1: Counting Heads in a series of tosses

This needs to be typed in exactly as it looks here; if there are typos the program won't run. We will see later how to find and eliminate errors; for now we'll just assume there aren't any. You will see that the editor colors some words differently than others; we will discuss the significance of this later. After we save this program we can run it by selecting Run Module from the Run menu of the edit window. This will open a shell in which you will see the output of the program, a line such as

```
483 out of 1000
```

This program simulates 1000 random tosses of a coin, so the output will change each time we run it. We can run the program again either by going back to the edit window (which should still be open) and again selecting Run Module from the Run menu, or in the shell window we could type

```
> main()
```

You might not fully understand why this works until you have read Chapter 5 on functions, but we will set up all of our programs so at the top level there is a function main(), so you can always re-run a program by typing that into the shell after the program has run once.

Most of the times you use Python you will not be typing a program into the system from scratch. It is much more common to start with an existing program and modify it into the program you want. Sometimes you will just want to examine a program to see how it works. To bring up a program in the IDLE editor you need to click on it in a special way. On Windows systems you click with the right mouse button on a ".py" file. This brings up a *context menu* with the options for this file. One of these is Edit with IDLE and that opens the IDLE editor on the program. Macintoshes generally don't have 3-button mice. On Mac you <CTRL>-click (hold down the Control or <CTRL>-key while clicking the mouse button) on the file. This time there are three choices: IDLE, Build Applet and Python Launcher. One of these will be labeled Default, which means that it is the choice that will apply if you just click normally on the file. You want to choose IDLE here.

To change the default action on a Mac, select any Python (.py) file and go to the File menu of the Finder. This has an option GetInfo, which brings up a window with various items related to this file. One of them is Open With, which has the IDLE, Build Applet and Python Launcher options. You can either change the default for this file, or if you select Change All you can change the default for all .py files.

1.4 Editing Programs with IDLE

IDLE will format programs for you in ways that make them easy for both you and the Python system to understand, but you have to let it do its work. Here are a few tips:

- Be careful with horizontal spaces. As we will see in the next few chapters, Python is very particular about indentation; the way a program is indented affects the way it runs. If two lines are supposed to have the same indentation, as in the last two lines of

```
if x < 10:
        result = "small"
        print( "That number is small." )
```

  it is not enough to make them look the same on the screen – their indentations need to consist of the same characters (either spaces or tabs). I recommend using the `tab` key for all indentation.

- Be generous with vertical spaces. Python ignores blank lines in your code. It helps the reader of your programs (especially you) if you use blank lines to separate the functional parts of your programs. As our programs become more complex, they will consist of more and more short functions, each of which is independent of the others. Use an extra blank line to separate functions. That helps to make the structure of the program easier to understand, and the overall program easier to read. As we will say many times, a program that is easy to read is much more likely to be correct than one that is hard to read.

- Beware of long lines. Because Python uses indentation as a functional part of the language, complex portions of code tend to be shifted to the right in the IDLE window. For some reason the designers of IDLE did not give it a horizontal scrollbar. If you have a line of code that extends to the right of your IDLE window, you must put the cursor on this line and use the right arrow key to move the cursor to see the rightmost portions of the line. This is both awkward and mistake-prone. You can split the line, but you must be careful how you do it because white space is meaningful in Python. If you end a line, Python will probably think you are ending the statement it contains and, if this isn't a complete statement, give you an error message. To avoid this, put a "\" character at the end of the line you will continue. To the system this says "Treat the next line as a continuation of the current one." For example,

```
print( "Dear %s. Please send $%d" to me soon." \
          %( "Mom", 20) )
```

  Without the \ character at the end of the first line this code would not work.

- Write programs non-linearly. Only the very simplest programs are written from top to bottom. Usually I write a small portion of code, then see what supporting code it needs and write that above the original chunk of code. Over the course of writing a complete program I tend to move all over the file. IDLE gives you a nice editor that makes it easy to move around the file; make use of it.

- Write small chunks of code at a time. Beginners want to write a complete program and then have a horrible time trying to debug it. More experienced programmers write a bit of a program, test it out, then write another piece of the program and test it out and repeat this until the program is finished. In an integrated system like IDLE and the Python shell, it takes only a few seconds to test a program and if the new code is only a few lines it can't take long to debug. When you are in a hurry to be done it always feels like it will be faster to just write all of the code, but in the end it will be slower and more frustrating to do it this way. Programming is one area where taking your time and being careful and methodical actually save you time.

IDLE colors

The IDLE editor makes programs colorful. This is not just for the artistic value; the colors are meaningful. Here is how the colors are used by default; if you wish you can change the color assignments in the Configure IDLE option of the Options menu of IDLE.

orange Keywords: words that are meaningful in Python, such as "if", "def", and "while".

green Strings: text in quotes, such as "bob".

red Comments: text for the human reader of the program that ignored by the computer.

blue Names of things being defined.

Chapter 2

Simple Python Code

In this chapter we will look at the basic elements of programming in Python. There isn't a lot you can do without the control structures of Chapter 3, so we will move quickly through this material. You might want to review this after you have read Chapters 3 and 4. Though none of this is difficult, there are some subtleties here that you might not appreciate unil you have some programming experience.

2.1 A first program

In this chapter we will begin to write programs using the simplest elements of Python. There are several factors that make this a complex, and sometimes frustrating, process. One is that programs are more formal than human communication, and everything in a program needs to be correct in order for it to run. If you are just starting to learn a new human language, such as Spanish, you might put a verb in the wrong tense or use the wrong gender for a noun but people will still know what you mean. Computers aren't as good as people at finding the essence of what you are saying. If your program isn't completely correct, the computer will not run it. Fortunately, many of the formal details of programs are the same in program after program; you can use them almost like incantations to achieve what you want.

Another factor that makes programming complicated at the start is that we want to eventually write complex programs that do useful things. In this chapter we are trying to lay the groundwork for a full understanding of programming, so at times we will use techniques that are more complex than necessary for the task immediately at hand, but that use a style that will be easy to adapt later on.

There are four primary types of instructions in Python, and in most programming languages: definitions, expressions, statements and comments. An *executable* instruction does something. For example, the line of code $x = 3 + 4$ adds together the numbers 3 and 4 and stores the result in variable x. There are two kinds of executable instructions. An *expression* represents a value, such as 3+4. A *statement* does something, such as **print** x. A *definition* creates something and gives it a name for later use. Nothing appears to happen when you execute a definition, but the system's memory is modified to include the newly defined object. Finally, a *comment* is a note for human readers of the program. Comments are ignored by the computer; their sole purpose is to help humans understand the program. Comments in Python start with the symbol # and extend to the end of the line.

The Python system starts reading your program at the top; it reads and executes statements until it gets to the end. Of course, all that happens when it executes a definition is that its memory is modifed to include the new object. Consider Program 2.1.1:

```
# This asks for the user's name
# and prints a greeting.

def main():
    name = input( "Who are you? ")
    if name == 'bob':
        print( "Bob rules!" )
    else:
        print( "Hello, " + name + "!" )
    print( "Goodbye." )

main()
```

Program 2.1.1: Our first program

This starts with two lines of comments:

```
# This asks for the user's name
# and prints a greeting.
```

Anyone reading the program can tell the program's purpose from these two lines without reading any further. The program then has a definition:

```
def main():
    name = input( "Who are you? ")
    if name == 'bob':
        print( "Bob rules!" )
    else:
        print( "Hello, " + name + "!" )
    print( "Goodbye." )

main()
```

This defines an object called main(). Objects like this that have parentheses as part of their names are called *functions*. These are the one of the primary building blocks of programs. Most of the programs we will write will have a sequence of function definitions at the start. Note that there are many lines indented underneath the line

```
def main():
```

These lines make up the body of the function - the instructions to be executed when the function is used. Python is different from most programing languages in that it uses indentation as a structural part of programs. Other languages uses braces, such as { and } or words such as begin and end to group together statements. Python's use of white space (space characters and tabs) makes it very visual; many people think this makes Python programs easier to read.

The final line of Program 2.1.1 is

```
main()
```

This invokes, or calls, function main() that was defined earlier. The system calls a function by executing the statements in its body one at a time. First, we have

```
    name = input( "Who are you? ")
```

This is complex. The string "Who are you? " is printed and the system halts and waits for the user to type a line of text that is terminated by the *Return* key. Whatever the user types is given as a string to the variable name.

The next instruction in main() is an *if*-statement:

```
if name ==   bob ':
        print( "Bob rules!" )
    else:
        print( "Hello, " + name + "!" )
```

This compares the value stored in variable name to the string bob '. If they match, the system prints Bob   rules . If they don't match it prints the word Hello , followed by the value in variable name followed by an exclamation point.

The last line of function main() just prints the word "Goodbye." This line is not part of the *if*-statement and so it is executed regardless of what name the user enters. Note how Python's use of indentation helps us to understand the functionality of the program. If we indented the last line, so that main() reads

```
def main():
    name = input( "Who are you? ")
    if name == 'bob':
        print( "Bob rules!" )
    else:
        print( "Hello, " + name + "!" )
        print( "Goodbye." )
```

then, in the case where the user enters bob the system will only print "Bob rules!" and if the user enters 'Mary' the system will respond

```
Hello, Mary!
Goodbye.
```

We run this program by opening it in a Python window inside Idle, or else typing it into a new file in Idle. One of the menus in this window is called Run, and there is an option in this menu called Run module F5 Either select this option or else press the F5 key to run the program. Here is a typical run, with what the user prints in bold and what the user types in normal font:

```
Who are you? Mary
 Hello, Mary!
 Goodbye.
```

Program 2.1.1 has a structure that we will use over and over. Until we discuss functions more completely in Chapter 5, all of our programs will have the following form

```
# comment on what the program does
def main():
          ⋮
main()
```

The use of main() here as the function that holds the body of the program goes back to the C language, which required a function called ain()} to fill this role. Python doesn't care what the
is; our programs would run just as well with the structure:

```
def stuff():
          ⋮

stuff()
```

We will use the name "main" as a concession to tradition.

Note that Program 2.1.1 is more complex than it actually needs to be. Python is a scripting language, which means that it can execute sequences of statements without an encompassing program. Program 2.1.2, below, runs in the same way as Program 2.1.1. For very short programs this can be useful, but as soon as the programs gain a little complexity this style becomes very hard to follow. For this reason we will stick to the style of Program 2.1.1 for all of our subsequent programs.

```
# This asks for the user's name
# and prints a greeting.

name = input( "Who are you? ")
if name == 'bob':
    print( "Bob rules!" )
else:
    print( "Hello, " + name + "!" )
print( "Goodbye." )
```

Program 2.1.2: Our second program

So far, programming should look easy. Programming is a great spectator sport. It is easy to nod your head and say "Yes, I understand." as long as someone else is doing the coding. When you start writing your own programs matters become a little more complex. In the next section we will look at some of the mistakes you might make, and how the system will respond to these mistakes.

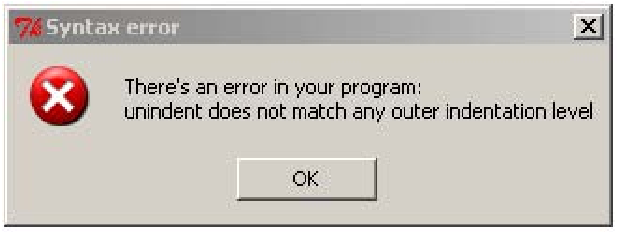
2.2 What can go wrong?

Let's look again at the first program we wrote in Section 2.1 and see what happens if we don't get it quite right. First, suppose we aren't consistent with our indentation:

```
# This asks for the user's name
# and prints a greeting.

def main():
        name = input( "Who are you? " )
    if name == 'bob':
        print( "Bob rules!" )
    else:
        print( "Hello, " + name + "!" )
main()
```

Program 2.2.1: Bad Indentation

In program 2.2.1 the line name = **input**("Who are you?") is not fully indented. If you write a program inside Python's own editors it is pretty easy to avoid indentation problems, but if you write a program in a standard text editor (such as Notepad in Windows) mistakes like this can crop up. If we run this program in the Python system, we get the following error alert:



There's an error in your program:
unindent does not match any outer indentation level

If we look at the code in IDLE, a portion of the line following the bad line is highlighted. Since the error message talks about indentation levels, it doesn't take too long to realize that there is a problem with the indentation somewhere, and if you look carefully at the code knowing that there is an indentation problem, the mistake isn't hard to spot.

Don't expect too much from error messages. Error detection in programs is an art form that is still far from perfect. It is fairly common for the system to flag the line after the one that actually contains the mistake. In this case the system misinterpreted our intention. "unindent does not match any outer indentation level" means that the system thought we were ending the main() function but failed to move to the previous indentation level, as in

```
def main():
    name = input( "Who are you? ")
if name == "bob":
          ⋮
```

That, of course, is not what we wanted, but it is close enough to allow us to find the real problem.

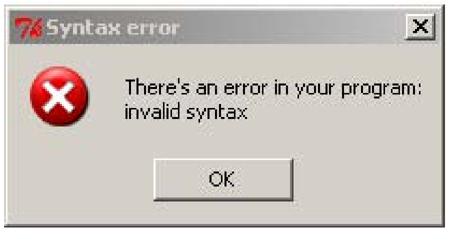
The Python error detection system stops as soon as it finds one problem. This makes error correction a cyclic affair: we fix one bug, try to rerun the program, fix the next bug that appears, and so forth. Experienced programmers often prefer systems that list all of the errors at once; it is faster to do multiple bug-fixes in one iteration than to handle them serially. The problem with systems like this is that one error can cause a whole sequence of errors. As we saw above, our improper indentation caused the Python system to think we were doing something completely different than we actually intended. If it had continued with the program it would have flagged the next line as being improperly indented (according to what it thought we intended) and these errors would continue through the rest of the program. It takes experience to look at a cascade of errors and determine what needs to be fixed. Python's system is much easier for beginners to use. It does have one offputting result. Each time you fix a bug it is the only bug you know about, so you believe the program will run correctly. This leads to a cycle of overconfidence followed by feelings of frustration. One way to minimize this is to keep debugging your program as you write it, so that with each run only a small piece of the code is new. We will discuss this at more length in a later section.

What happens if you forget a comment sign, as in the second line of the following program?

```
# This asks for the user's name
and prints a greeting.

def main():
    name = input( "Who are you? ")
    if name == 'bob':
        print( "Bob rules!")
    else:
        print( "Hello, " + name + "!")
    print( "Goodbye." )

main()
```

Program 2.2.2: Bad Comment

This time we get the cryptic error message:


Syntax error

There's an error in your program:
invalid syntax

OK

This doesn't say much of anything. ”invalid syntax” means that your program isn't constructed according to the grammar rules of Python. Fortunately, if we look at the code in Idle it shows us the problem. The second line of the code is highlighted:

```
and prints a greeting.
```

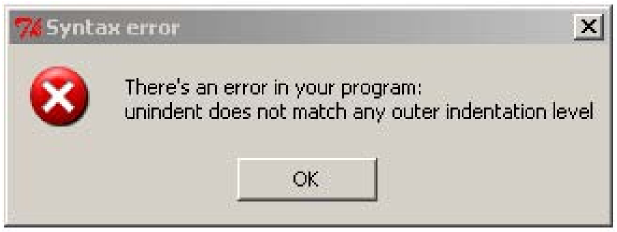
Now we know that there is a syntax problem with the second line, starting with the word **and** Of course, since this line is supposed to be a comment the system shouldn't be checking its syntax; this ought to lead us to realize that the problem is our missing comment symbol.

Many Python constructs use colons to separate various parts of a construction. An easy typing mistake is to leave off one of these colons. We do this in the following program, where the colon on”else” is omitted.

```
# This asks for the user's name
# and prints a greeting.

def main():
    name = input( "Who are  you?" )
    if name == 'bob':
        print( "Bob rules!")
    else
        print( "Hello, " + name + "!" )
    print( "Goodbye." )

main()
```

Program 2.2.3: Omitted colon

When we run this program the system again reports a syntax error:



There's an error in your program:
unindent does not match any outer indentation level

Nothing here points to a missing colon as the problem, but if we look at the code the **else** line is highlighted. For someone who is familiar with how *if*-statements are constructed in Python (we will look at this in detail in Chapter 3) it is a small step to realize that the problem is a missing colon.

As a final example for this section, consider Program 2.2.4:

```
# This asks for the user's name
# and prints a greeting.

def Main():
    name = input( "Who are  you? " )
    if name == 'bob':
        print( "Bob rules!" )
    else:
        print( "Hello, " + name + "!" )
    print( "Goodbye." )

main()
```

Program 2.2.4: Case-sensitivity

In this program we define a function called Main() and at the end call a function called main(). Python, like most programming languages, is case-sensitive. Upper case letters are different from lower-case letters with the same name, so Main() and main() are different functions. If we try to run this program we four lines of red ink:

```
Traceback (most recent call last):
  File "C:\Documents and Settings\bob\My Documents\book\Program2.2
        line 13, in <module> main()
NameError: name 'main' is not defined
```

Don't be put off by the red ink or verbose text. The last two lines of this message have everything you need to know. On line 13 of the program:

```
     main()
```

there is a problem because the word main isn't defined. Of course, it should have been defined in the definition at the top of the program, so as soon as we look at that definition we see that it defines Main() and not main(). Nothing is highlighted in your program because the system can't tell if you misspelled the name of the function in its definition or in its call, but this error message should be sufficient for you to figure out what the problem is and to correct your spelling.

2.3 Variables and types

A paragraph in English is a sequence of words written according to the grammar rules of English so that it is meaningful. The words of the paragraph are either standard words of English, which you can find in a dictionary, or names of objects, which can be almost anything, such as "Caerdydd" or "2Pac". The situation in programming is very similar. A program is a sequence of words and symbols written according to the grammar rules of a programming language (in our case the rules of Python) so that it can be executed on a computer. The words of the program are either the keywords of the language, such as "def", "if", and "else", or they are the names of objects that are constructed in the program, or they are textual data.

Program 2.1.1 showed two ways that a program can make named objects. The line

```
def main():
```

starts the definition of a function called main(). We could name this function anything else. For example, in the following program the main function computes the area of a rectangle, so it is called area() rather than main().

```
# This asks for the dimensions of
# a rectangle and computes its area.

def area():
    print( "This finds the area of a rectangle." )
    L = eval( input( "Enter the length: " ) )
    W = eval( input( "Enter the width: ") )
    print( "The area is ", L*W )

area()
```

Program 2.3.1: The top-level function is called area()

Although the Python language allows this, as we said in Section 2.1 our convention is to always name the top-level function of our programs main().

Definitions are one way to assign a name to an object. Program 2.3.1 illustrates the other common way to make a named object. The line

```
L = eval( input( "Enter the length" ) )
```

creates a variable called L, and gets its initial value from input from the program's user. In any programming language a *variable* is a name attached to a location in memory where we can store data. In many languages, including both C and Java, variables must be *declared*, or registered with the program before they can be used. When a variable is declared the programmer must say what kind of data it will hold so that the right amount of memory can be assigned to it. Python does things differently. In Python a variable is created when we attach a value to a name; no declaration is necessary. Furthermore, we can change the kind of data attached to a variable at any point in the program; the system determines from usage what kind of data a variable holds and adjusts its internal memory accordingly. This is very convenient, but it is not without its problems, which we will discuss below. For now, just focus on the fact that we create a variable by assigning a value to a name.

Finally, note that in Program 2.1.1 we used the **input** function by itself:

```
name = input( "Who are you? " )
```

while in Program 2.3.1 we also invoke the **eval** function:

```
L = eval( input( "Enter the length: " ) )
```

The **input** function always returns a string. If this is what we want, for example if we expect the user to enter a name, we don't need to do anything to it; we just pass this string on to whatever variable it is being assigned to. On the other hand, if we expect the user to enter a number we need to translate the string returned by **input** into a numerical representation. That is the role of the **eval** function – it takes a string and evaluates it into its natural Python representation. For more on this see section 2.6.

Assignments

The = symbol is used for assigning a value to a variable. The variable receiving a value is always on the left side of this symbol, and on the right side is an expression of the value being assigned. In Program 2.3.1 there are two assignment statements:

```
L = eval( input( "Enter the length: " ) )
```

and

```
W = eval( input( "Enter the width: ") )
```

The first of these creates variable L and reads a value from the user to put into L. The right side of an assignment statement can be anything that computes to a value. For example,

```
Z1 = 23*5
```

and

```
Z2 = 1 + eval(input( "Enter a number: " ))
```

are both valid statements. The first of these puts the number 115 into variable Z1; the second reads a number from the user, adds 1 to this value, and stores the result in variable Z2. In Program 2.3.1 we could introduce a variable that holds the area of the rectangle as follows:

```
L = eval( input( "Enter the length: " ) )
W = eval( input( "Enter the width: ") )
A = L*W
print( A )
```

Sometimes we need to increase the value of a variable. For example, in a predator-prey simulation we might have a variable foxCount that represents the current number of foxes. If a new fox is born, we need to add one to this count. Here is a line of code that does this:

```
foxCount = foxCount + 1
```

When assignment statements are evaluated, the right side is always computed first, and that value is given to the variable on the left side. This statement takes the current value of variable foxCount, adds 1 to it, and that result becomes the new value of foxCount. If the old value of foxCount was 97, after this statement is excuted its new value will be 98.

There is a shorthand notation for updating the values of variables. The statement x += a is short for x = x + a: the value of a is added to variable x. Similar definitions apply for −=, *= and /=, though += is by far the most commonly used of these notations.

Names

Every program you write will have variables and functions, and you will need to come up with names for these objects. Python doesn't care what names you use: as far as the Python system is concerned pq23x is as good a name as FoxCount. For human readers names make a difference. You are more likely to write programs that work if you use good names. There are many different naming conventions, and unless you are working in a group programming environment where everyone needs to use the same conventions, it doesn't really matter which conventions you use. Here are some guidelines that will help you write successful programs:

- Names are composed of letters, digits, and the underscore character. All names in Python must start with a letter. 1x is not a legal name, though x1 is.

- Use names that have some meaning for your program. You could use x to represent the width of a rectangle, but W, width, and rectangleWidth are all better choices.

- Use names composed of English words. In a program that deals with rectangles and circles, you could use A1 to represent the area of a rectangle and A2 to represent the area of a circle, but is is easy to forget which is which. A better choice is to use rectangleArea and circleArea . When you want to use more than one English word in a name, push the words together and capitalize the first letter of each word (or the first letter of each word after the first). Another option is to use underscores to separate words, as in rectangle_area . I tend to use capitalization rather than underscores because I find capital letters easier to type, but you might make a different choice.

- Avoid cute names. It might seem fun to give variables the names of your friends, but a few days later you are likely to forget which name goes with which object. If you use descriptive names you are more likely to finish your programs early, and then you can spend quality time with your friends.
- Be consistent with capitalization. Some of the most common bugs are introduced by capitalizing a variable one way in one part of a program and a different way in another part of the program, and so creating two different variables. Some people use a convention that function names all start with a capital letter and variables names start with a lowercase letter. With this convention prime might be a variable that represents a prime number while Prime() is a function concerned with prime numbers. You arent likely to confuse a variable with a function (though it occasionally happens); the real point of this convention is that if you have a variable that represents a part number in an inventory program you know it will be spelled partNumber and not PartNumber.

Example 1. As an illustration of variables and assignment statements, here is a common situation. We have variables x and y that have values. We want to interchange these values give x the value currently in y, and y the value currently in x. The following code doesnt work:

```
x = y
y = x
```

The statement x = y results in the old value of x being lost; after this x and y have the same value. For example, if we start with x having the value 23 and y having 45, then after x = y both will have the value 45. The statement y = x then has no effect; after it both variables will still have the value 45. There is no way to do this with just two variables; to make it work we need to introduce a new variable to hold one of the values for us. Here is code that does work:

```
oldX = x
x = y
y = oldX
```

The variable oldX holds onto the original value of x, so this value is still around after we make the assignment x = y. We can then use this variable to give the right value to y. Note that order matters: the following code has the same three assignment statements, but it doesnt work:

```
x = y
oldX = x
y = oldX
```

Try an example, such as starting values 23 for x and 45 for y, and convince yourself that this code doesnt interchange the values of x and y.

Types

Computers can manipulate many different kinds of data. In some problems the essential data is numeric; other problems need textual information and still others need complex data, such as the position and orientation of planets in their orbits around the sun. Different kinds of data allow for different operations: we can multiply two integers, but it doesn't make sense to multiply two English words. A *type* is a collection of values together with the basic operations that can be performed on these values. For example, the Integer type has the values 0, 1, -1, 2, -2, 3, -3, and so forth, together with operations like addition, subtraction, multiplication and division. You need to be aware of the types of the data you have stored in your variables, and to use the variables consistently according to these types.

Python has several basic types of data, together with some ways to build more complex types from these primitive types. The four most common elementary types are:

- *Integers.* These have whole numbers as values and a wide variety of arithmetic operators for manipulating them.
- *Floating Point numbers*, sometimes called "floats". These have decimal values: 3.14159, -3.5, or 6.0. Again, there is a wide variety of arithmetic operators available for working with floats. Note that the floating point number 6.0 is different from the integer number 6 because they are elements of different types. They are also represented differently in the computer's memory; using one when you mean the other can get you into trouble.
- *Strings.* These are sequences of individual characters, such as "Scooby Doo" or 'Plan 9 From Outer Space'. We write strings inside quotation marks (either single quotes or double quotes). Strings also have their own operations, which we will discuss later.
- *Booleans.* There are only two Boolean values: True and False. Note that they are written with capital initial letters: true is not a Boolean value. They are also not strings: True and "True" are completely different objects. You might not see at the moment why Booleans are useful, but starting with Chapter 3 we will make frequent use of them. By the way, the unusual name "Boolean" is a tribute to George Boole, a 19th Century English mathematician who was one of the original developers of symbolic logic.

The next two sections discuss numeric types, then strings. In Chapter 3 we will discuss the Boolean type. There is also a Complex Number type built into Python, but we will not make use of it in these notes.

2.4 Numeric Types

Integers

Integers are the most familiar of all the standard types. Any whole number is an integer. Older computer languages, like C and Java, have limits on the size of integer values due to the way they represent numbers internally. These limits are large, around 2 billion, but occasionally we need numbers that are larger than than these limits and this can create bugs in programs. Python has no limit on the size of an integer value. For example, here is the number 100-factorial (100!), which is the product of all of the integers between 1 and 100: 9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000000. This is a legitimate integer value in Python.

There are 7 standard arithmetic operators for integers:

Symbol	**Meaning**	**Example**	**Result**
$+$	Addition	$3+5$	8
$-$	Unary: negation Binary: subtraction	$-(3+4)$ $7-3$	-7 4
$*$	Multiplication	$4*7$	28
/	Exact division	11/3 12/3	3.666667 4.0
//	Whole division	11//3 12//3	3 4
$**$	Exponentiation	$2**4$ $10**-1$	16 0.1
%	Remainder	11%3 12%3	2 0

Most of these need no explanation. It may surprise you that there are two different division operators, but both are useful. The "exact division" operator / gives the quotient of its two operands as a floating point number. The "whole division" operator gives the whole number of times its second operand can be divided into its first operand. You have computed remainders before in mathematics classes, but mathematicians don't have a standard symbol for the remainder operation; programming languages do.

The remainder operator `%` is used more than you might expect. Suppose numbers `x` and `d` are integers. If `x % d` is 0, then `d` divides evenly into `x`. This has many applications. For example, we can tell if a number `x` is even or odd by checking `x % 2`; if this is 0 then `x` is even and if it is 1 then `x` is odd. Similarly, if `day1` and `day2` are two different dates of the same month, then they fall on the same day of the week if `day1%7` is the same as `day2%7`. For example, I am writing this on July 8, 2015, which is a Monday. July 15, July 22 and July 29 are also Mondays: `8%7`, `15%7`, `22%7` and `29%7` are all 1.

There is a rule in most programming languages, that an operation between two integers results in an integer, while an operation between an integer and a float is a float. This holds for all of Python's operators except for exact division / and exponentiation where the second operand, the exponent, is negative. In early versions of Python `a**b` wasn't defined when `b` was negative, but negative exponents are sufficiently convenient that the language designers finally relented and allowed non-integer results.

Here is a table of comparison operators for integers. Each of these returns a Boolean (True or False) value:

Symbol	**Meaning**	**Example**	**Result**
$<$	Less than	$3 < 5$	True
		$5 < 3$	False
$>$	Greater than	$3 > 5$	False
		$5 > 3$	True
$<=$	Less than or equal to	$3 <= 5$	True
		$5 <= 5$	True
		$5 <= 3$	False
$>=$	Greater than or equal to	$3 >= 5$	False
		$5 >= 5$	True
		$5 >= 3$	True
$==$	Equal to (comparison, not assignment)	$3 == 5$	False
		$5 == 5$	True
$! =$	Not equal to	$3! = 5$	True
		$5! = 5$	False

Floating Point numbers

Floats are numbers that are not necessarily integers. They have a more complex internal representation inside the computer (in binary) than integers do, and there is not necessarily a perfect correspondence between decimal values and floating point representations. For example, if you ask Python to compute 1.0/10, the result is not 0.1, but rather 0.10000000000000001. Of course, this is close enough for most computational purposes, but it leads to some unexpected results. If you ask the system whether 1.0/10 is 0.1, the answer is "No". If general, it isn't a good idea to directly compare two floating point values. You can ask whether they are close (say within 0.00000001), but don't test them for equality.

Floats have the same arithmetic operators and the same comparison operators as integers:

Symbol	Meaning	Example	Result
+	Addition	3.2+5	8.2
-	Unary: negation Binary: subtraction	-(3.4+4.5) 7.0-3.9	-7.9 3.1
*	Multiplication	4.2*5	21.0
/	Exact Division	12.6/3 11.0/3	4.2 3.6666666666
//	Whole Division	12.6//3 11.0//3	4.0 3.0
**	Exponentiation	2**3.6	12.125732532083186
%	Remainder	11.0%3 10.5%3.3	2.0 0.6
<	Less than	3.2 < 5.6	True
>	Greater than	3.2 > 5.6	False
<=	Less than or equal to	3.2 <= 5.6	True
>=	Greater than or equal to	3.2 >= 5.6	False
==	Equal to (comparison, not assignment)	3.2 == 5.6	False
! =	Not equal to	3.2! = 5.6	True

When dividing decimal numbers we dont usually think of getting a ”remainder”, but the definition for floats is the same as it is for integers: `a%b` is what remains after we remove the largest possible integer multiple of `b` from `a`. One way we might compute this for positive values of `a` and `b` is to repeatedly subtract `b` from `a` until the result is smaller than `b`; that result is the remainder.

The *print()* function works with floats just as it does with integers. The easiest way to control the number of decimal places printed with a float is to use string formatting, which we describe in the next section. Similarly, input works with floats just as it does with integers. Finally, the comparison operators for floats are the same as those for integers.

Converting between types

Sometimes you have an integer variable that you want to change to a float, or vice versa. There are several ways to achieve this. In Python you can use type names like functions to convert values to these types. For example, if `x` has an integer value, such as 23, and you want to get the floating point version of this, 23.0, you can get it as **float**(`x`). Similarly, we can get the integer version of a floating point number `x` with **int**(`x`). This always rounds `x` towards 0: **int**(`34.9`) is 34, and **int**(`−34.9`) is -34.

The Math library

The following table lists many functions that are available in the Math library for Python. To use any of these you need to put the following line at the top of your program:

```
from math import *
```

If you omit this line you will get an error message saying that the functions are not defined.

Alternatively, you can include at the top of your program

```
import math
```

and prefix any item that you use from the library with "math.", as in

```
area = math.pi*(radius**2)
```

The idea here is that you can either import all of the names from the library into your program or you can just import a link that allows you to refer to objects in the library though the <library-name> dot notation.

Symbol	**Meaning**	**Example**	**Result**

ceil(x)	The ceiling of x: the smallest whole number (still a float) greater than or equal to x.	ceil(34.2)
		ceil(-34.2)
			35.0
			-34.0

floor(x)	The floor of x: the largest whole number (still a float) less than or equal to x.	floor(34.2)
		floor(-34.2)
			34.0
			-35.0

fabs(x)	The absolute value of x: if $x > 0$ this is x, and if $x < 0$ this is x.	fabs(34.2)
		fabs(-34.2)
			34.2
			34.2

exp(x)	This returns where e is the base of the natural logarithm.	exp(2)	7.38906
log(x)	This is the natural logarithm; some math books call it $ln(x)$.	log(15)	2.70805
log(x, base)	This is the logarithm of x with the given base.	log(32, 2)	5.0
log10(x)	This is log(x, 10).	log10(100)	2.0
sqrt(x)	The square root of x.	sqrt(144)	12.0

sin(x)	This is the sine of angle x, where x must be measured in radians. If you want to use degree measure, use sin(radians(x)).	sin(30)
		sin(radians(30))
			-0.988032
			0.5

cos(x)	This is the cosine of angle x, where x must be measured in radians. If you want to use degree measure, use cos(radians(x)).	cos(60)
		cos(radians(60))
			-0.952413
			0.5

tan(x)	This is the tangent of angle x, where x must be measured in radians. If you want to use degree measure, use tan(radians(x)).	tan(45)
		tan(radians(45))
			1.619775
			1.0

asin(x)	This is the angle, in radian measure, that has x as its sine.	asin(0.5)
		degrees(asin(0.5))
			0.523599
			30.0

acos(x)	This is the angle, in radian measure, that has x as its cosine.	acos(0.5)
		degrees(acos(0.5))
			1.047198
			60.0

atan(x)	This is the angle, in radian measure, that has x as its tangent.	atan(1.0) 0.785398 degrees(atan(1.0))	 45.0
degrees(x)	This takes angle x, measured in radians, and returns the number of degrees in x.	degrees(pi/4)	45.0
radians(x)	This takes angle x, measured in degrees, and returns the number of radians in x.	radians(45)	0.785398
hypot(x, y)	This returns which is the length of the hypotenuse of a right triangle whose other sides have lengths x and y.	hypot(3, 4)	5.0
pi	pi is the ratio of the circumference of a circle to its diameter.	pi	3.14159
e	e is the base of the natural logarithm.	e	2.71828

The Random library

Random numbers are very useful for games, simulations, and other sorts of programs that we will write. Python has a very powerful, easy-to-use library of random number utilities. To use any of these, you need to put one of the following line at the top of your program:

```
from random import *
import random
```

With the former line you can use the random functions directly; with the latter line you need to prefix them with "random.", as in `random.randint(1, 6)`. The following is a small selection of the functions that are available in this library. All of these functions return floating point values.

Symbol	**Meaning**	**Example**	**Result**
random()	This returns a random floating poing number between 0 and 1.	random()	0.23145
uniform(a, b)	This returns a random floating poing number between a and b.	uniform(3, 5)	4.21368
randint(a, b)	This returns a random integer between a and b, possibly equal to either a or b.	randint(0, 1) randint(0, 1)	0 1
seed(x)	This resets the starting point of the random number generator based on the value of x, which could be an integer, a floating point value or a string. If you want number to be random but repeatable, set the seed to the same value each time. This doesnt return anything.	seed("bob")	No output.

2.5 Strings

Computers were originally designed to "compute" numeric values, but at a very early stage scientists realized that they were just as useful for manipulating textual information. The basic data type for working with text is called a *string*. Python has a particularly strong set of operations for working with strings. Here we will introduce the basic techniques for strings, which will allow us to use text as well as numbers in our programs. In a later we will give a more thorough coverage of strings and their capabilities.

There are three ways you can get strings into a program: they can be literal strings in the program code, you can input them from the user or a file, or you can compute them from simpler strings, even from individual letters. String literals are sequences of characters inside quotation marks. You can use either single quotes or double quotes, as long as you end the string with the same kind of quote that you used to start it. For example "Four score and seven years ago" and 'Barack Obama' are two examples of string literals. Note that "example" and 'example' are exactly the same string. The string "", which contains no characters at all (in other words, this string has length 0) is called the empty string. Empty strings serve a role similar to 0. We often will build up strings by starting with the empty string and adding letters. Note that in Python there is no separate type for individual characters, such as the letter 'a'. Characters in Python are just strings of length 1.

You can work with strings just as you work with numbers. Variables can be assigned string values, as in

```
name = "Hieronymous Bosch"
```

Strings can be given values through user input, as in

```
name = input("Enter a name: " )
```

The **input**() function doesn't try to evaluate what the user types, it just returns it as a string. Finally, you can build up a string value within a program, as in

```
name = "Marvin " + "Krislov"
```

There are many operators and functions for working with strings. One of the most common is the length function **len**(s), which returns the number of letters in string s. The + operator when used with strings gives the concatenation. or pushing together, of the operands. For example, "multi" + "media" is the new string "multimedia". Note that you can add two numbers with + and you can concatenate two strings with +, but it doesn't make sense to use + between a string and a number.

Sometimes we want to grab the individual letters that make up a string. The expression s[n] returns the letter in the *nth* position of string s. For technical reasons Python, and most other computer languages, start indexing strings at position 0. Thus, if test is the string "abcde", then test [0] is the letter "a", test [1] is "b", and so forth. String x is indexed from position 0 to position **len**(x)−1.

A *slice* of string s is a substring made up of contiguous letters in s. s[i:j] is the slice of string s consisting of letters from position i to position j (including i and not including j). Again, we start indexing positions with 0. For example, if word is the string "together", then word[2:5] is the string "get".

The following table summarizes the most commonly used string operations. In this table x, s, and t are strings, n, i, j, and k are integers.

Symbol	Meaning	Example	Result
str(n)	Returns the string version of n.	str(23)	"23"
len(s)	Length of string s.	len("Katrina")	7
s+t	Concatenation of strings s and t.	"multi" + "media"	"multimedia"
s*n or n*s	String s is repeated n times.	"bob"*3	"bobbobbob"
s[n]	The nth letter of string s. Indexing starts at 0.	x ="abcde" x[3]	"d"
s[i:j]	The slice of string s, consisting of the letters starting at position j (indexing starts with 0) and extending up to but not including position j.	type = "deciduous" type[4:7]	"duo"
s[i:]	This is the slice of string s starting at position i and going to the end of s.	type = "deciduous" type[4:]	"duous"
s[:j]	This is the slice of string s starting at the beginning of s and extending up to but not including position j.	type = "deciduous" type[:4]	"deci"
x in s	True if x is found in s and False otherwise. x can be either a single letter or a possible substring of s.	'o' in "bob" 'd' in "bob"	True False
x not in s	False if x is found in s and True otherwise. x can be either a single letter or a possible substring of s.	"o" not in "bob" 'd' not in "bob"	False True

s.find(x)	If string s contains string x, this is the index of x's starting position in s. If s does not contain x, this is -1.	s = "abcde"
		s.find("cd")
		s.find("x")
			2
			-1

s.lower()	Returns a string with the same characters as s, but all in lowercase.	s = "wHimseY"
		s.lower()
			"whimsey"

s.upper()	Returns a string with the same characters as s, but all in uppercase.	s = "wHimseY"
		s.upper()
			"WHIMSEY"

s.islower()	Returns True if s has at least one alphabetic letter and all of the alphabetic letters in s are lowercase.	s = "bob"
		s.islower()
			True

s.isupper()	Returns True if s has at least one alphabetic letter and all of the alphabetic letters in s are uppercase.	s = "OH!"
		s.isupper()
			True

Strings in Python can't be changed once they are created. For this reason strings are called *immutable*. This doesn't mean that a string variable can't be changed; it is only the string itself that cannot be altered. In this regard Python strings are no different from numbers: you cannot change the number 23 to be a different number. Some other languages have mutable strings. For example, in Java if s is the string "picket" and we execute a statement s[1] = 'o', this changes string s to "pocket". We can get the same effect in Python from the code:

```
s =  "picket"
s = s[:1] + 'o' + s[2:]
```

Note that rather than change a string, in Python we usually reconstruct it and assign the new string to the old string variable.

The string comparison operators use the same symbols as arithmetic comparisons. Strings are ordered by the usual dictionary (alphabetical) ordering, where a standard table of characters, called the *ASCII* table, is used to determine the ordering of individual characters. See the optional section on string encodings, below, for more details about this table. Here is a table of the comparison operators:

Symbol	**Meaning**	**Example**	**Result**
<	Less than	"*bike*" < "*car*"	True
>	Greater than	"*bike*" > "*car*"	False
<=	Less than or equal to	"*bike*" <= "*car*" "*car*" <= "*car*"	True True
>=	Greater than or equal to	"*bike*" >= "*car*" "*car*" >= "*car*"	False True
==	Equal to (comparison, not assignment)	"*bike*" == "*car*" "*car*" == "*car*"	False True
! =	Not equal to	"*bike*"! = "*car*" "*car*"! = "*car*"	True False

String formatting

It is often the case that you want to print a line that is composed of text plus values that are contained in one or more variables. For example, an averaging program might have a variable average that contains the average, and a variable count that holds information about the number of item that went into this average. We might want to print all of this information on one line, such as:

```
7 items were found, with average 16.2.
```

To do this with string concatenation we would need to convert the numeric variables into strings:

```
print(str(count)+" items were found, with average "+str(average)+".")
```

This works, but it is hard to read and easy to get wrong. String formatting is a technique Python provides to simplify expressions like this. It is particularly useful for print statements, but it can be used anywhere you need to make a string out of values contained in variables.

A formatted string consists of two parts separated by a percent sign:

```
    pattern % (values)
```

The pattern can have percent-fields that act as placeholders. There are three such fields:

%d	is a placeholder for an integer value.
%f	is a placeholder for a floating point value.
%s	is a placeholder for a string.

For each percent-field in the pattern, you need to give a corresponding value. The values are a list, separated by commas, inside parentheses, such as (count, average) If you have only one value you may omit the parentheses.

The averaging example above would be expressed as follows with formatted strings:

```
print("%d items were found, with average %f."%(count,average))
```

This is easier to read, and easier to write correctly.

Formatted strings have another advantage. You are allowed to supply a field-width for each percent-field. This is very useful if you are trying to print data in a table so that it comes out in columns. If, for example, you have numbers that might need anywhere from 1 to 4 digits, printing them with a fieldwidth of 4 or more will guarantee that they always occupy the same amount of space in the output line.

For strings and integers the fieldwidth is a number that comes between the %-symbols and the s or d symbol. If the fieldwidth is positive, smaller values are right-justified; if it is negative they are left-justified. For example "%5d" means to print the integer value using at least 5 spaces, and to put the value at the right edge of this field. "%-5d" means to make the fieldwidth 5, and to put the value at the left edge of this field. With floats the fieldwidth is a bit more complex. "%w.df" means to use a total fieldwidth of w, including d places after the decimal point. For all of these fieldwidths, an omitted fieldwidth defaults to 1.

Altogether, our averaging example would be formatted:

```
print( "%d items were found, with average %.2f."%(count,average))
```

This means to print count using no more spaces than necessary, but to use 2 decimal places for average.

Here is another example, this time a complete program. We want to read in a person's name and age, and then print them out on one line. Program 2.5.1 does this. Note that we use **eval()** after the numeric input. Note also the use of formatted strings in the print statement. name is a string and age is an integer, so we use formats %s and %d formats for them.

```
# This reads a person's name and age,
# then prints them on one line

def main():
    name = input( "Enter a name: " )
    age = eval( input("Enter this person's age: ") )
    print( "%s is %d years old." % (name, age))

main()
```

Program 2.5.1: Names and Ages

Strings and encodings (optional)

Internally, all data inside a computer is represented by sequences of 0's and 1's. A given sequence of bits could represent either a number or a string, depending on how it is interpreted. Python lets us move easily between these two interpretations. You don't usually need to do this the mechanisms the language provides for manipulating text usually make it unnecessary to do this. In case you are curious, here are some details about how strings are encoded.

There are two built-in functions for working with the numeric encodings of strings:

ord(c)	gives the numeric value of c, which must be a single character: a string of length 1.
chr(n)	gives the character corresponding to integer n.

For example, ord('A') is 65 and ord('a') is 97. Python uses a standard character encoding called the *ASCII* character set. "ASCII" stands for "American Standard Code for Information Interchange". The ASCII encoding goes back to the early 1960's and has been the standard representation for text for many years. The first 31 characters in this set are non-printing "control" characters for controlling line printers and other output devices. In the table below we list all of the printing ASCII characters.

The ASCII Character Set														
x	**chr(x)**		**x**	**chr(x)**		**x**	**chr(x)**		**x**	**chr(x)**		**x**	**chr(x)**
32	space		51	3		70	F		89	Y		108	l
33	!		52	4		71	G		90	Z		109	m
34	”		53	5		72	H		91	[		110	n
35	#		54	6		73	I		92	\		111	o
36	$		55	7		74	J		93	]		112	p
37	%		56	8		75	K		94	ˆ		113	q
38	&		57	9		76	L		95	_		114	r
39	’		58	:		77	M		96	‘		115	s
40	(		59	;		78	N		97	a		116	t
41	)		60	¡		79	O		98	b		117	u
42	*		61	=		80	P		99	c		118	v
43	+		62	¿		81	Q		100	d		119	w
44	,		63	?		82	R		101	e		120	x
45	-		64	@		83	S		102	f		121	y
46	.		65	A		84	T		103	g		122	z
47	/		66	B		85	U		104	h		123	{
48	0		67	C		86	V		105	i		124	|
49	1		68	D		87	W		106	j		125	}
50	2		69	E		88	X		107	k		126	˜

You can see that this character set has some convenient characteristics: the lower-case letters are all contiguous and the upper-case letters are too, though the upper-case letters and the lower-case ones do not appear together. In fact, all of the upper-case letters come before all of the lower-case ones. This ordering is used when we alphabetize words in Python: unless we do extra work to avoid this, anything that starts with an upper-case letter will appear before anything that starts with a lower-case one.

Python uses ASCII encodings for strings when it is possible, because each of the ASCII characters can be encoded in one byte (8 bits) of data. However, Python also implements an extension of ASCII called /em Unicode that uses two bytes per character. This allows for about 65,000 different characters. For example, chr(9924) is a cute picture of a snowman while chr(9824) is the ”spade” character from a deck of playing cards. If you run the code

```
for i in range(0, 20000):
    print( "%d: %s" %(i , chr(i)))
```

you can see many of the options for this extended character set.

2.6 Input and Output

We have already seen Python's **input**(), **eval**() and **print**() functions. In this section we will look at them with a little more care.

Input from the user

Python's **input**() function is very basic. When it is called the system waits for the user to type something and hit the Return key. When this is completed the function takes whatever the user has typed, packages it into a string, and returns it. It does not evaluate what the user has typed in any way. You can use this function with no argument, in which case it just halts and waits for input, or you can give it a string which it interprets as a prompt and prints before it waits for the input:

```
name = input( "Who are you? " )
```

I usually leave one blank space at the end of the prompt string (just before the closing double-quotes) so the user's input doesn't run right up against the prompt; it makes the input dialog easier to read.

The **input**() function will only get you a string. If you want something else, you need to evaluate in some way the string that **input**() returns. The easiest way to do this is with the **eval**() function. **eval**(**input**(<prompt>)) prints the prompt, waits for input, and then tries to interpret the input as a Python value. This is easy and straightforward if the user has entered a number, as in

```
length = eval( input( "Enter the length: " ) )
```

We will usually restrict our use of **eval**() to this situation: interpreting strings as numbers. It is possible to evaluate more elaborate strings. The string "[2, 4, 6, 8]" evaluates to a *list* with 4 elements. For more on lists, see Chapter 6. The string "1, 2, 3" is evaluated as a *tuple* with 3 elements. Tuples are also discussed in Chapter 6. The problem with trying to use **eval**(**input**()) for these is getting the user to type in exactly the right format. The wrong format, such as omitting the commas between the elements, will cause **eval**() to crash on the input string.

One option for inputting compound data is to give the user a format and then use the string method split () to separate it into fields that can be easily evaluated. For example, suppose we want the user to enter a date. We might signal the input with

```
dateString=input("Enter a date in the form m/d/y: ")
```

We can then use the string split () method to divide the input into fields. split () takes one string argument that it regards as a *separator*, and it splits the string into a list of fields determined by this separator. For example, if dateString is "10/29/1929" then dateString. split ("/") is the list ["1","29", "1929"]. The individual fields in this are still strings, but we can use the **eval**() function to turn them into numbers. Here is a complete block of code that does this, giving integer values to the variables month, day, and year.

```
dateString=input("Enter a date in the form m/d/y: ")
dateList=dateString.split("/")
month=eval(dateList[0])
day=eval(dateList[1])
year=eval(dateList[2])
```

Printing

The **print()** function is used to make output appear on the *console*, the computer screen. The simple statement

```
print(x)
```

causes the value of x to be printed. You can print multiple values with

```
print(x, y, z)
```

By default a single space is printed between each of the values. We will see below how to change this default. The **print()** function also terminates its line of output, so the code

```
print("Marvin Krislov")
print("Oberlin College")
```

results in

```
Marvin Krislov
Oberlin College
```

Again, we will see how to change this default. The full **print()** function is

```
print( <values>, sep=<separator>, end=<terminator> )
```

with all three fields optional. Here <values> is either a single value or a comma-separated list of values, such as 23, 45. If the values are omitted, only the end-string is printed. <separator> is any string we want to appear between the values. For example,

```
print( 23, 45, 58, sep="***")
```

will print

```
23***45***58
```

If there is only one value the separator is not printed at all. By default the separator is a string consisting of one single space character: " ".

It is often convenient to build up a line of output through several calls to **print()**. The end-string, or terminator, defaults to the newline character, which in Python is written "\n". This is what causes **print()** to end the current line and prepare to start a new line. If we change the end-string to something else we stay on the same line after printing, so the next call to **print()** appears on the same line. If we want to do this the most common value to use for the end-string is the empty string, or "". For example, the following code

```
print("Marvin Krislov", end="")
print("Oberlin College")
```

will print

```
Marvin KrislovOberlin College
```

Here is a simple example of how we might use these techniques to print a table in nice orderly columns. The following code is a little redundant; we will be able to do this much more simply after we see *loops* in Chapter 3.

```
print( "%5d" % 123, end="" )
print( "%5d" % 5, end="" )
print( "%5d" % 17, end="" )
print( "%5d" % 444, end="" )
print()
print( "%5d" % 11, end="" )
print( "%5d" % 9, end="" )
print( "%5d" % 123, end="" )
print( "%5d" % 3, end="" )
print()
print( "%5d" % 422, end="" )
print( "%5d" % 73, end="" )
print( "%5d" % 12, end="" )
print( "%5d" % 9, end="" )
print()
```

This prints

```
123    5   17  444
 11    9  123    3
422   73   12    9
```

2.7 Exercises

Write a complete Python program to solve each of the following problems. Use formatted strings to make nice output statements. Note that a few of these require *if*-statements. Well discuss these statements in detail in the next chapter; for you you might look at program **??** for an illustration of *if*-statements.

2.1. Write a program that asks the user to enter two numbers and then prints the product of those numbers. Here is a typical run of the program:

```
Enter a number:  23
Enter another number:  45
23 x 45 is 1035
```

2.2. Write a program that asks the user to enter a string, and then reports the number of letters in that string. Here is a typical run:

```
Enter a string:  Ask not what you can do for your country.
That string has 41 letters.
```

2.3. Write a program that asks the user to enter one number. The program should then say if that number is even or odd. Remember that a number `x` is even if `x%2` is 0, and odd if `x%2` is 1. Here is a a typical run:

```
Enter a number: 23
23 is an odd number.
```

Here is another run:

```
Enter a number: 34
34 is an even number.
```

2.4. Write a program that asks the user to enter two numbers and then prints the smaller of those numbers. Here is a typical run:

```
Enter a number: 34
Enter another number: 23
The smaller of those is 23
```

2.5. Redo program (2.4), but with three numbers. You will have to do more logic to figure out which is the smallest number:

```
Enter one number: 23
Enter another number: 16
Enter yet another number: 45
The smallest of those is 16
```

2.6. Write a program that inputs a non-negative integer as a number of cents and prints that value in the standard dollars-and-cents format. For example, with input 405 the output should be \$4.05; for 23 the output should be \$0.23, and for 1900 the output should be \$19.00.

Chapter 3

Control Structures

Control structures determine the order in which the statements of a program are executed. They are the most fundamental building blocks of programs. It can even be shown that in a programming language with conditional statements, loops, and assignment statements (and nothing else) one can write any program that can be written in any language. Control structures are all that you really need. In this chapter we will look at control structures that are common to most programming languages in wide use, including **if**-statements, **while**-loops and **for**-loops.

3.1 If-statements

An **if**-statement is used to create conditional code: code that is executed if some condition is true, and not executed if it isn't. There are several versions of the **if**-statement; all of these have the meanings you would expect from similar usage in English.

```
if <condition>:
        statement_block
```

If the condition is True the statement_block is executed; if the condition is False, the statement_block is ignored. The statement_block is any Python code indented inside the **if**-statement.

```
if <condition>:
        statement_block_1
else:
        statement_block_2
```

If the condition is True then statement_block_1 is executed; if the condition is False then statement_block_2 is executed. The statement_block_1 is any code indented between the **if** and the **else**. The statement_block_2 is any code indented under the **else**.

```
if <condition_1>:
        statement_block_1
elif <condition_2>:
        statement_block_2
elif <condition_3>:
        statement_block_3
           ⋮
else:
        final_statement_block
```

The conditions are evaluated one at a time, starting from the top. If any of them evaluate to True the corresponding statement_block (the group of statements indented under the condition) is executed and then we leave the entire statement. If none of the conditions are True then the final_statement_block is executed. Note that **elif** is just an abbreviation for **else if**. In all of these

statements the conditions are anything that evaluates to True or False. We call such expressions *Boolean* expressions; the next section has more details about them. The statement blocks contain any valid Python statements.

This is much simpler than it might appear. Consider, for example, the following program. This asks the user to enter two numbers, and says which of the numbers is the larger one and which is the smaller.

```
# This asks for two numbers and
# prints them out in order.

def main():
    x = eval( input( "Enter a number: ") )
    y = eval( input( "Enter another number: ") )
    if x == y:
        print("Those numbers are the same.")
    elif x < y:
        print("%d < %d" % (x, y))
    else:
        print("%d < %d" % (y, x))

main()
```

Program 3.1.1: Example of an **if**-statement

When you are writing a program using conditional statements, keep in mind that you don't know which blocks will be executed unless you know what the user input will be. You can get into trouble if you create variables inside some branches of an **if**-statement and not in others. Consider the following code:

```
x = eval( input("Enter a number: ") )
if x < 10:
    size = "small"
print(size)
```

If the user enters a number that is less than 10 this prints the word "small". However, if the user enters a number that is 10 or larger the variable size is never created and so you will get an error message for the unknown varible size when it gets to the **print**-statement. One way to correct this is to define variable size outside the **if**-statement, so it will exist in any case:

```
size = "large"
x = eval( input("Enter a number: ") )
if x < 10:
        size = "small"
print(size)
```

Alternatively, you can put an **else** on the **if**-statement to define size in case the condition is False:

```
x = eval( input("Enter a number: ") )
if x < 10:
    size = "small"
else:
    size="large"
print(size)
```

When you have a chain of **if**-statements, as in **if-elif-elif** ..., remember that you only get to the lower conditions if all of the upper conditions fail. You can make use of this to simplify the conditions. For example, suppose we want a program that takes a number and says how many digits it has. Program 3.1.2 does this for numbers up to 4 digits long.

```
# This asks for a number and reports how
# many digits it contains.

def main():
    x = eval( input("Enter a number between 0 and 9999: ") )
    if x < 10:
        digits = 1
    elif x < 100:
        digits = 2
    elif x < 1000:
        digits = 3
    else:
        digits = 4
    print("%d has %d digits." % (x, digits))

main()
```

Program 3.1.2: Compound **if**-statement

Note that by the time we get to the condition

```
elif x < 100:
```

we already know that the condition x < 10 has failed, so if x is less than 100 it must have exactly 2 digits. If would be correct, but unnecessarily complicated to write this condition as

```
        elif x >= 10 and x < 100:
```

By carefully analyzing a situation, it is often possible to simplify conditions in your code and thus make your programs easier to read and easier to write correctly.

We finish this section with one more example, this time computing leap years. We will make use of this code for several programs in subsequent chapter

Example The Julian Calendar, introduced by Julius Caesar in 46 B.C., had leap years occurring every 4 years. This equates the *tropical year*, the span from one point in the cycle of the seasons to the same point the following year, to 365.25 days. This is slightly too long, and over the centuries the Julian calendar grew out of synchronization with the natural seasons. In 1582 Pope Gregory XIII created the Gregorian Calendar, which differs from the Julian Calendar only in the way it computes leap-years. The Gregorian calendar was gradually adopted throughout Europe and most of Asia. Here is its leap-year algorithm:

A year is a leap year if it is divisible by 4, unless it is also divisible by 100, in which case it is not a leap year unless it is also divisible by 400.

According to this algorithm, the year 1900 was not a leap year because it is divisible by 100 and not by 400, but the year 2000 was a leap year. This puts the length of the tropical year at 365.2425, which is very close to current observations. Our task is to write a program that asks the user to specify a year; the program will say whether or not this year is a leap year.

The way we stated the leap-year algorithm is not very helpful; the "unless" expressions don't translate easily into code. For our program we will reformulate the algorithm, taking the most specific conditions first. This is often a useful strategy. We express the most specific conditions first because we know what the answer is if those conditions are met.

For our program the conditions are:

- the year is divisible by 4
- the year is divisible by 100
- the year is divisible by 400

The most specific of these is being divisible by 400, so we start there: if the year is divisible by 400 it must be a leap year. We can use our remainder operator % to check for divisibility: a year is divisible by 400 if year % 400 == 0. Rather than sprinkling print statements throughout the code, we introduce a boolean varible isLeap that becomes True or False in the various conditions of our **if**-statement. At the end we use this variable to print out whether the year is a leap year.

The remaining cases of the analysis are similar. Program 3.1.3 shows the resulting code for this.

```
# This reads a year from the user and
# says whether this year is a leap-year.

def main():
    year = eval( input("Enter a year: ") )
    if year % 400 == 0:
        isLeap = True
    elif year % 100 == 0:
        isLeap = False
    elif year % 4 == 0:
        isLeap = True
    else:
        isLeap = False

    if isLeap:
        print("%d is a leap-year." % year)
    else:
        print("%d is not a leap-year." % year)

main()
```

Program 3.1.3: Determining leap years

3.2 Booleans

The *Boolean* data type consists of just two values: True and False. In Python this is implemented as a sub-type of the integers, where 0 corresponds to True and any non-zero value corresponds to False. You can actually use integer operations on Boolean values if you choose (such as True + False), but this will just make your programs harder to read. The most common ways to produce Boolean values is through the comparison operators (such as < or ==), and through the membership search operations on strings and lists.

Sometimes you need to build compound conditions out of simpler ones. For example, if you want to check that a number x is between 1 and 10, you can do so with

```
if x >= 1 and x <= 10:
```

In this example we took two complete Boolean conditions and connected them with the operator **and**.

There are three Boolean operators: **and**, **or**, and **not**.

Symbol	Meaning	Example	Result
and	x and y is True only when both sides are True; if either side is False the result is False.	$x = 23$	
		$x < 10$ and $x > 0$	False
		$x > 10$ and $x < 100$	True
or	x or y is True when either side is True; it is only Fals when both sides are False.	$x = 23$	
		$x < 10$ or $x > 0$	True
		$x > 10$ or $x < 100$	True
		$x < 10$ or $x < 20$	False
not	not x has the opposite value of x: it is False when x is True and True when x is False.	$x = 23$	
		not $x < 10$	True
		not $x > 20$	False
		not $x == 23$	False

You can build very complex expressions using these operators. In general, if you aren't sure how Python will group together sub-expressions, it is best to use parentheses to force the expression to be parsed in the way you intend. For example, the following condition describes the numbers 6, 7, 8, 9, 21, 22, 23;

```
if ((x > 5) and (x < 10)) or ((x >20 and (x < 24)):
```

One comforting thing about the way Python implements the Boolean operators is that it stops evaluating as soon as it knows the answer. For example, the expression a **and** b is False if either of its operands is False, so if a is False there is no need to evaluate b at all. If s is a string and n is an integer might have the code

```
if n < len(s) and s[n] == 'b':
        print( "The string contains letter 'b'." )
```

If n is *not* less than the length of the string, the right side of the expression s[n] == 'b' would normally cause the program to crash. It doesn't in this case because Python never evaluates the right side of the expression; it knows the result just from the left side, n < **len**(s). Note that if we wrote the expression in the opposite order,

```
if s[n] == 'b' and n < len(s):
        print( "The string contains letter 'b'." )
```

then the program will crash when n gets a value large than the length of s. The **or** operator is implemented in the same way: if its left operand evaluates to True it doesn't bother to evaluate the right operand since it knows the result of the full expression is True.

As a final example we will develop a program that inputs three numbers and *sorts* them, which means it puts them into increasing order. There are many sorting algorithms, but with just three numbers we can take a common-sense approach. First we need to input the three numbers, which we will store in variables a, b, and lstinlinec. This calls for three input statements, each of which will look like

```
a = eval( input("Enter a number: ") )
```

Next, we need to determine the ordering of lstinlinea, b, and lstinlinec. There are factorial (n) ways to order n numbers (There are lstinlinen different choices for the first number; for each of these there are n−1 choices for the second number, and so forth....); since factorial (3) is 6, we know that there are 6 possible orderings for our numbers. One way to write the program would be to have 6 variations of a statement such as

```
if a <=b and b <= c:
    print(a, b, c)
```

We will take an alternative approach that finds the smallest number and then orders the other two. This uses three variations of the following statement:

```
if a <= b and a <= c: # a is the smallest
    if b <= c:
        print(a, b, c)
    else:
        print(a, c, b)
```

Note the use of **else** here. If a is the smallest of the three numbers and b is not less than or equal toc, then c must be smaller than b.

Here is the entire program

```
# This inputs 3 numbers and prints
# then out in inceasing order.

def main():
    a = eval( input("Enter a number: ") )
    b = eval( input("Enter another number: ") )
    c = eval( input("Enter a third number: ") )
    print( "The correct ordering of those is ", end="")
    if a <= b and a <= c: # a is the smallest
        if b <= c:
            print(a, b, c)
        else:
            print(a, c, b)
    elif b <= a and b <= c:  # b is the smallest
        if a <= c:
            print(b, a, c)
        else:
            print(b, c, a)
    else: # c is the smallest
        if a <= b:
            print( c, a, b)
        else:
            print(c, b, a)

main()
```

Program 3.2.1: Sorting three numbers

3.3 While-loops

A *loop* is a block of code that a program will execute over and over until some condition is met. Loops are among the most important concepts in all of programming. Most programs spend most of their execution time inside loops. If we didn't have loops the runtime of a program would be proportional to its length; loops allow us to have short programs that perform lengthy computations.

Python has two loop constructs, which we call **while**-loops and **for**-loops. In this section we will introduce **while**-loops and in the next section we present **for**-loops. The remainder of the chapter provides examples of these useful constructs.

A **while**-loop has the structure:

```
while <condition >:
        statement_block
```

When this statement is executed, the statement_block is executed over and over until the condition finally becomes False. If the condition is False when we first get to the loop, the statement_block is not executed at all.

Program 3.3.1 prints the integers from 1 to 3.

```
# This prints the numbers 1 to 3
def main():
    x = 1
    while x <= 3:
        print(x)
        x = x + 1
main()
```

Program 3.3.1: First loop example

We will now step through this program one instruction at a time. You want to think of loops functionally in terms of what they do, but occasionally it is useful to simulate the way the computer deals with the loop.

At the start we set x = 1.
We test x <= 3. This is True, so we enter the loop body.
We print x, which is 1.
We set x = 1+1, which is 2.

We return to the top of the loop and test x <= 3. This is True so we enter the loop body.
We print x, which is now 2.
We set x = 2+1, which is 3.

We return to the top of the loop and test x <= 3. This is True so we enter the loop body.
We print x, which is now 3.
We set x = 3+1, which is 4.

We return to the top of the loop and test x <= 3. This is False, so we leave the loop.

Of course, this is a lot of work to print the numbers 1, 2, and 3, but we can easily change the program to Program 3.3.2, which prints the numbers from 1 to 10,000.

```
# This prints the numbers 1 to 10,000
def main():
    x = 1
    while x <= 10000:
        print(x)
        x = x + 1
main()
```

Program 3.3.2: A longer-running version of program 3.3.1

In both Program 3.3.1 and Program 3.3.2 we give an initial value to variable x before the top of the loop. This is important. Since x is part of the loop condition, it must have a value before the condition can be evaluated. If we left off the assignment statement x=1 we would get an error for an unknown variable x at the start of the loop. In general, every variable mentioned in the condition of a **while**-loop must be given a value before we get to the loop; if we fail to do this the program will crash with an ”unknown variable” error message.

Here is another example. Program 3.3.3 reads a positive integer from the user and then prints all the ways it can be factored into a product of two integers. For example, if you enter 6 it prints

```
1 * 6 = 6
2 * 3 = 6
3 * 2 = 6
6 * 1 = 6
```

```
# This reads a positive integer and prints all
# the ways it can be factored.

def main():
    n = eval(input("Enter a positive number: "))
    factor = 1
    while factor <= n:
        if n % factor == 0:
            print("%d * %d = %d"%(factor,n/factor,n))
        factor = factor + 1

main()
```

Program 3.3.3: This factors a number entered by the user.

Variable factor plays the role that x did in the previous programs: it takes on all of the values between 1 and an upper bound, which in this case is n. It is very important in this program that the line that increments variable factor: factor = factor + 1 is not inside the **if**-statement. If it was, then once we got a value of factor that did not divide evenly into n we would never change from this value, and the program would never halt. If this should ever happen to you, there are two ways to stop a runaway program. The <Ctrl>-C combination generates a "keyboard interrupt" that will stop any running program. You can also close the shell window in which the program is running. It is better, however, to avoid this situation by thinking carefully about your programs. Every time through a loop you should make some progress towards eventually terminating.

Input loops

One very common use for a loop is to collect input from a user. The loop reads data and processes it until some condition regarding the input is satisfied. The loop condition in this situation is almost always dependant on computations in in the body of the loop, so it might seem difficult to set up this condition so that it has a value at the start, before the body of the loop has been executed.

Program 3.3.4 shows a way to handle this situation. In this program the user is repeatedly asked to enter numbers. When the user responds with a 0 the loop ends and the average of the user's numbers is printed.

```
def main():
    total = 0.0
    count = 0
    done = False
    while not done:
        x = eval(input("Enter a number, or 0 to exit: "))
        if x ==0:
            done = True
        else:
            total = total + x
            count = count + 1

    print("%d numbers were read for an average of %.2f"
       % (count, total/count))

main()
```

Program 3.3.4: Computing an average of data supplied by the user

This program bases the termination condition for the loop on variable done, which reports whether we are done with the input phase of the program. Before the loop starts we have done no input so we can't be done: this variable is always initialized to False. Each time we read new input we immediately check it against the exit condition and adjust the value of done accordingly.

The following program illustrates an input loop that reads text strings rather than numbers. The idea of the loop is the same as Program 3.3.4.

```
# This is a guessing game based on
# a Brothers Grimm tale
def main():
    done = False
    while not done:
        name = input( "What is my name? " )
        if name.lower() == "rumpelstiltskin":
            done = True
        else:
            print( "Wrong! Guess again." )
    print( "Drat!  You guessed it." )
main()
```

Program 3.3.5: What's my name?

The program repeatedly prompts the user to enter a name. If the name is "rumpelstiltskin" the loop halts and the program prints `"You guessed it."` For any other input there is a `"Guess again"` message. This program accepts any capitalization of the name: "Rumpelstiltskin", "rumpelstiltskin" or even "RumpelStiltSkin". It does this by comparing name.lower(), which is the translates all of the letters to lower0-case, to the lower-case version of the correct answer: "rumpelstiltskin". Note how easy this is; with very little work you can make any input be case-insinsitive.

For a final example, we will nest one loop inside another. This is very common in programming, and also a common source of errors. To do this correctly, we need to be in control of our code and know what each element of the program is doing.

The problem we will address with the program is this: *Write a program that inputs numbers and prints their factorizations into prime numbers. The program should end when the user inputs the number 0.* For example, for input 84 the program should print `2 2 3 7`.

We will write this program in stages. The first stage just handles the user input getting numbers from the user until we get the number 0. This is similar to programs 3.3.5 and 3.3.6.

```
# This gets numbers from the user and
# prints their prime factorizations

def main():
    done = False
    while not done:
        n = eval( input("Enter a number, or 0 to exit: ") )
        if n == 0:
            done = True
        else:
            print( "factorization of %d goes here" % n )

main()
```

The user-input loop

Note the way we left a marker for where the next portion of the code goes the " factorization goes here" line will be replaced by a new block of code. Note also that, although this doesn't completely solve the problem we started with, it is a complete, working program. We can run it to test out our code so far. This is important. As you start to develop longer and longer programs, you should try to keep your programs in a working state as you generate more code. This allows you to test out your code is small portions that are easy to understand. If we run this much of our program we get the following interaction:

```
Enter a number, or 0 to exit: 2
factorization goes here
Enter a number, or 0 to exit: 5
factorization goes here
Enter a number, or 0 to exit: 0
>>>
```

Our next step is to replace the " factorization goes here" line with code that actually computes the prime factorization. We don't need to actually generate prime numbers for this. If we start looking for factors at 2 and replace n by its quotient with each successful factor, the only factors we will find will be prime numbers. For example, suppose we start with n = 90. 2 is a factor, so we write down 2 and reduce n to 45. 2 is no longer a factor of this, so we try the next number, 3, as a factor. 3 divides evenly into 45, so we write down 3 and reduce n to 15. 3 is still a factor of this, so we write down 3 again and reduce n to 5. 3 is no longer a factor, so we try 4, which is also not a factor. We eventually test 5 as a factor; this is successful so we write down 5 and reduce n to 1. We can now stop, since there are no more possible factors. Here is the code that implements this:

```
factor = 2
while factor <= n:
        if n % factor == 0:
                print( factor , end=" ")
                n = n/factor
        else:
                factor = factor + 1
print
```

Finding factors

Note that we print each successful factor with the statement

```
    print( factor , end=" " )
```

so we don't actually terminate the output line. At the end of the factorization we have one **print** statement, which completes the line of output.

Here is our current version of the program.

```
# This gets numbers from the user and
# prints their prime factorizations

def main():
    done = False
    while not done:
        n = eval(input( "Enter a number, or 0 to exit: "))
        if n == 0:
            done = True
        else:
            factor = 2
            while factor <= n:
                if n % factor == 0:
                    print( factor , end= " ")
                    n = n/factor
                else:
                    factor = factor + 1
            print
main()
```

User-input plus factoring

If we run this we get the following interactions:

```
Enter a number, or 0 to exit: 34
2 17
Enter a number, or 0 to exit: 160
2 2 2 2 2 5
Enter a number, or 0 to exit: 3344
2 2 2 2 11 19
Enter a number, or 0 to exit: 0
>>>
```

This works, but it seems awkward to have the same factor printed over and over again, as in the factorization of 160, which is 2x2x2x2x2x5. With a little more thought we can have a third loop count how many times each factor divides into n. If the count is 1 we just print the factor; if it is more than 1 we print the factor and the count, with the format factor\^count. Here is that code:

```
factor = 2
while factor <= n:
    if n % factor == 0:
        count = 0
        while n%factor == 0:
            count = count + 1
            n = n/factor
            if count == 1:
                print( factor , end=" ")
            else:
                print( "%d^%d"%(factor ,count), end=" " )
            else:
                factor = factor + 1
print
```

Counting the factors

and here is the final version of the program:

```
# This gets numbers from the user and
# prints their prime factorizations

def main():
    done = False
    while not done:
        n = eval(input "Enter a number, or 0 to exit: "))
        if n == 0:
            done = True
        else:
            factor = 2
            while factor <= n:
                if n % factor == 0:
                    count = 0
                    while n%factor == 0:
                        count = count + 1
                        n = n/factor
                    if count == 1:
                        print(factor, end=" ")
                    else:
                        print("%d^%d"%(factor, count), end=" ")
                else:
                    factor = factor + 1
            print( )

main()
```

Program 3.3.6: The final version

This time we get the following interactions:

```
Enter a number, or 0 to exit: 34
2 17
Enter a number, or 0 to exit: 84
2^2 3 7
Enter a number, or 0 to exit: 3344
2^4 11 19
Enter a number, or 0 to exit: 0
>>>
```

3.4 For-loops

In the previous section we looked at **while**-loops, Python's basic looping structure. There is a second loop construct in Python called a **for**-loop. This is more specialized. We use **for**-loops when there is a particular set of values that we want to work through, doing some task once with each of these values.

A **for**-loop has the structure:

```
for <variable> in < sequence>:
        statement_block:
```

When this statement is executed, the variable takes on the first value in the sequence and the statement_block is executed. The variable then takes on the next value and the statement_block is executed. This process continues with each of the values in the sequence.

What is a *sequence*? It is any collection of "sequential" data values: for something to be a sequence we must be able to say what the first value is, and for each value but the last we need to be able to say what the next value is. The non-negative integers form a sequence: 0 is the first value, 1 comes next, 2 is after that, and so forth. The non-negative real numbers do not. For example, there is no "next" value after 1.3. There is 1.31, but 1.301 comes before that, and 1.3001 precedes that , and so forth. Python has many ways to construct a sequence; among the most imporant of these is the *list*. A list is a sequence specified by listing values inside square brackets, as in ["John", "George", "Paul", "Ringo"]. Python makes it easy to work with lists. For example, Program 3.4.1 will print

```
The Beatles were:
  John
  George
  Paul
  Ringo
```

```
# This prints a list of the Beatles
def main():
    print( "The Beatles were:" )
    for name in ["John","George","Paul","Ringo"]:
        print( "    %s" % name )

main()
```

Program 3.4.1: The Beatles

Any values can be put into lists: strings, numbers, even other lists. For example, [1, 2, [3, 4]] is a list where the first two elements are numbers and the third is a list with two elements.

It is often useful to have a loop walk through a range of numbers. Python's **range**() function generates sequences of numbers for this purpose. There are three ways to use this function:

- **range**(a, b) generates a list-like structure containing the numbers from a to b: including a but not including b. For example, **range**(3, 7) is the sequence 3, 4, 5, 6s. This assumes a is smaller than b; if the reverse is true the list generated is empty.
- **range**(b) is the same as **range**(0, b). For example, **range**(5) is [0, 1, 2, 3, 4].
- **range**(a, b, c) uses an increment of c to go from one number to the next. So **range**(2, 12, 3) is [2, 5, 8, 11] and **range**(5, 0, −1) is [5, 4, 3, 2, 1].

In Python 2 **range**(a, b) is an actual list. This is simple but runs into trouble if you work with very large ranges. For example, if you try to roll a pair of dice a billion times with the code

```
for x in range(1000000000):
    value = RollDice()
    ....
```

Python 2 would actually generate in memory a list with a billion items. This would probably cause your program to crash. In Python 3 the **range**-function returns a **range−object** that can be iterated through with a for-loop and works like a list, but it has the same small size regardless of the size of the range. The only time you are likely to see a difference between lists and range-objects is if you try to print a range-object.

```
print( range(0, 10) )
```

prints the words **range**(0, 10) rather than the numbers from 0 through 9. If you want to print the elements of the range you can convert it into an actual list with the **list**( ) function:

```
print( list(range(0, 10)) )
```

prints [0,1,2,3,4,5,6,7,8,9]

The following code fragment uses the **range**() function to print the factors of the number 36. Note that we need to extend the range up to 37 so that the last number in the list is 36.

```
for x in range(1, 37):
    if 36%x == 0:
        print( x )
```

Program 3.4.1: Factoring 36

We can also treat strings as sequences — the letters are taken in the order in which they appear in the string. The next program reads a string from the user and reports how many instances of the letter "a" it contains. As is common with programs that count things, this has a variable count that starts at 0 and is incremented each time the appropriate letter is found.

```
# This counts the a's in a string
def main():
    string = input( "Enter a string: " )
    count = 0
    for letter in string:
        if letter == "a":
            count = count + 1
    print( "The letter 'a' occurs %d times."%count )
main()
```

Program 3.4.2: Counting instances of a letter

We will illustrate **for**-loops with several programs that find prime numbers. Remember that a number is *prime* if its only factors are 1 and itself. 2, 3, 5, 7, 11, and 13 are all prime numbers, 15 is not. First, we will write a program that lets the user input a number; the program will say if this number is prime. The basic structure inputs a single number and says if it is prime:

```
def main():
    number = eval(input("Enter a number: "))

    # test if it is prime; for now we
    # always say no.
    isPrime = False

    if isPrime:
        print( "%d is prime." % number )
    else:
        print( "%d is not prime." % number )

main()
```

Program 3.4.3: Initial version

Next, we need to fill in code for testing whether the number we have is prime. One way to do this is to think about the definition. A number is prime if it is divisible only by 1 and itself, so it is not prime if we can find any divisor greater than 1 and less than the number itself. This is a natural application for a **for**-loop we know exactly the range of values we want to check, so we use the **range**() function to generate them and a **for**-loop to check them. We will start the isPrime variable out at True and change it to False if we ever find a divisor.

```
isPrime = True
for factor in range(2, number):
        if number % factor == 0:
                isPrime = False
```

Putting this altogether, we have the first complete version of our prime finder:

```
# This lets the user enter a number and reports
# whether that number is prime.
def main():
    number = eval(input( "Enter a number: " ))

    isPrime = True
    for factor in range(2, number):
        if number % factor == 0:
            isPrime = False

    if isPrime:
        print( "%d is prime." % number )
    else:
        print( "%d is not prime." % number )

main()
```

Program 3.4.3: Final version

We can turn this into a program that generates prime numbers if we surround it with a loop in place of the single input statement. For example, if we want to find all prime numbers between 2 and 100 we would use the loop:

```
for number in range(2, 101):
        isPrime = True
        for factor in range(2, number):
            if number % factor == 0:
                isPrime = False
```

Note that for each new value of number we need to reset isPrime to True; otherwise, once we found a non-prime number there would never be a way to find a prime one: isPrime would always be False.

One more change to Program 3.4.3 will help make it a better prime number generator. The prime numbers are fairly sparse; most integers are not prime. If we print "is prime" or "is not prime" with every number, there are so many non-primes that it becomes hard to see the actual primes. We accordingly change the **if**-statement to report only the prime numbers and be silent on the non-primes. Here is the resulting program:

```
# This prints all of the prime numbers up to
# a limit supplied by the user
def main():
    max = eval(input("Enter the largest number to check: "))
    for number in range(2, max+1):
        isPrime = True
        for factor in range(2, number):
            if number % factor == 0:
                isPrime = False

        if isPrime:
            print( "%d is prime." % number )

main()
```

Program 3.4.4: Final version

As a final example for this section, here is a third prime number program. This time we make two changes that illustrate some of the differences between **for**-loops and **while**-loops. The first change is that instead of asking the user how high in the list of integers we should check for prime numbers, this time we ask the user how many prime numbers we should find. This might sound like a minor change, but it forces us to rethink our loops. In Program 3.4.4 our basic structure is this:

```
for number in range(2, max+1):
        <if number is prime, print it>
```

If N is the number of primes we need to find, we could change our main loop to:

```
for i in [1,2, 3,     . N]:
     <Find the ith prime and print it>
```

This isn't so simple. How do we find the *ith* prime number? This seems unnecessarily complicated. An easier solution is to have a loop that generates numbers to test, with the exit condition based on whether we have found enough primes. This needs to be a **while**-loop rather than a for-loop because we don't know how many times we will execute its body; we just keep going around until we have enough primes. This looks something like the following:

```
number = 2
primeCount = 0
while primeCount < N:
     <if number is prime:>
          print number
          primeCount = primeCount + 1
      number = number + 1
```

It should be an easy matter to convert the body of this **while**-loop to Python code. First, we are going to make another change in the program. What we have so far will make a long list of prime numbers. The output will be shorter if we allow for multiple columns, such as

```
     2         3         5         7        11        13
    17        19        23        29        31        37
    41        43        47        53        59
```

The first thought of many beginners at this point would be "We need another loop." But what would this loop do? We already have a loop to generate numbers. We don't need a loop to count the entries in a line; a simple variable can do that. We will print the numbers just as before, only with a formatting statement to control the number of spaces used for each number regardless of how many digits it has (so we get nice columns of output), and we will end each print statement with a comma, so that successive prints appear on the same line. Each time we print a number we will increment a lineCount variable that tells us how many numbers are on the current line. When this is large enough we execute a **print** statement (with no comma) to terminate the line and start over. Of course, we will also start our lineCount variable over at 0.

Now our code looks like this:

```
number = 2
primeCount = 0
linecount = 0
while primeCount < N:
     <if number is prime:>
          print("%7d " % number, end = "")
          primeCount = primeCount + 1
          lineCount = lineCount + 1
          if lineCount == C:
               print( )
               lineCount = 0
     number = number + 1
```

Here is the whole program:

```
# This prints the first N prime numbers,
# where the value of N is supplied by the user.
# The output is printed in C columns.

def main():
    N = eval(input("How many prime numbers do you want? "))
    C = eval(input("How many columns of output do you want? "))
    number = 2
    primeCount = 0
    lineCount = 0
    while primeCount < N:
        isPrime = True
        for factor in range(2, number):
            if number % factor == 0:
                isPrime = False

        if isPrime:
            print( "%7d " % number, end="")
            primeCount = primeCount + 1
            lineCount = lineCount + 1
            if lineCount == C:
                print( )
                lineCount = 0

        number = number + 1
main()
```

Program 3.4.5: Final version

3.5 Designing code with loops

In the last two sections we have seen **while**-loops and **for**-loops. Beginning programmers are often mystified as to when to use which type of loop. The good news is that it often doesn't matter. In many situations we can set up natural, easy to read code with either type of loop. For example, the following blocks of code both sum the numbers from 1 to 10:

```
sum = 0
for x in range(1, 11):
    sum = sum + x

print( sum )
```

```
sum = 0
x = 1
while x <= 10:
    sum = sum + x
    x = x + 1

print( sum )
```

In either case the loop generates the list of numbers; inside the loop we have code to compute their sum. The first of these is a little shorter, but not enough to matter. To someone familiar with the syntax of **for**-loops and **while**-loops they seem equally clear. In a situation like this you can use either looping construct.

Suppose we want to generate a sequence of ten coin tosses: "Heads" or "Tails". At the end of section 2.4 we discussed the random library that has random number generators. In this case we will use randint(0, 1), which gives random numbers that are either 0 or 1. The loop we use is just a counting device; it will generate values, but we won't do anything with them other than to count how many there are. Again, we could do this with either looping construct:

```
from random import *

for x in range(10):
    number = randint(0, 1)
    if number == 0:
        print( "Heads" )
    else:
        print( "Tails" )
```

```
from random import *

x = 0
while x < 10:
    number = randint(0, 1)
    if number == 0:
        print( "Heads" )
    else:
        print( "Tails" )
    x = x + 1
```

Programs often use loops to generate sequences of values that are to be manipulated in some way. For example, suppose we want to count the number of leap years between two given years, such as 1878 and 1945. There are two separate tasks here: one is to generate the sequence of years and the other is to determine which of them are leaps years and keep track of their number. The first task requires a loop, and it doesn't make much difference whether we use a **for**-loop or **while**-loop for it. The second task needs an elaborate **if**-statement; we saw code for this in Program 3.1.3. Here again we show parallel constructions, with **for**- and **while**-loops.

```
startYear = 1878
finalYear = 1945
leapCount = 0
for year in range(startYear, finalYear+1):
    if year % 400 == 0:
        isLeap = True
    elif year % 100 == 0:
        isLeap = False
    elif year % 4 == 0:
        isLeap = True
    else:
        isLeap = False

    if isLeap:
        leapCount = leapCount + 1

print( "%d leap years." % leapCount )
```

Program 3.5.1: Counting leap years, **for**-loop

```
startYear = 1878
finalYear = 1945
leapCount = 0
year = startYear
while year <= finalYear:
    if year % 400 == 0:
        isLeap = True
    elif year % 100 == 0:
        isLeap = False
    elif year % 4 == 0:
        isLeap = True
    else:
        isLeap = False

    if isLeap:
        leapCount = leapCount + 1

    year = year + 1
print( "%d leap years." % leapCount )
```

Program 3.5.2: Counting leap years, **while**-loop

In section 3.4 we saw a similar program where we wanted to find the prime numbers in a range of integers. We used a **for**-loop to generate all of the integers in this range, but we could have just as easily used a **while**-loop. Later in that section we wanted to find the first N prime numbers for some value N. That would not have been easy to do with a **for**-loop because we didn't know the endpoint of the range of values to generate as candidates.

When there is a definite sequence of values that need to be processed, **for**-loops are often more natural; this was the reason **for**-loops were invented. Suppose, for example, that we want to count the number of instances of the letter 'a' in the word 'abracadabra'. We can walk through the letters of the string with a simple **for**-loop:

```
for letter in "abracadabra":
```

With a while-loop we need to use numeric indexes for the string:

```
index = 0
s = "abracadabra"
while index < len(s):
    letter = s[index]
```

Here are the resulting programs:

```
count = 0
for letter in "abracadabra":
    if letter == "a":
        count = count + 1

print( count )
```

```
index = 0
s = "abracadabra"
count = 0
while index < len(s):
    letter = s[index]
    if letter == "a":
        count = count + 1
    index = index + 1

print( count )
```

The version with **for**-loops is certainly shorter. More importantly, it is more natural and easier to read. This means the programmer is more likely to write it correctly.

The following rule of thumb can help you decide which looping construct to use:

If you need to generate a *definite* list of values where you can say in advance which values belong in this list, **for**-loops are usually easier to use. If you need to generate an *indefinite* list of values, for which you cannot say in advance where to stop, **while**-loops are usually easier.

We will illustrate these ideas by developing a program that prints a calendar for one month. In the next few chapters we will make several versions of this program to illustrate various programming constructs. This version asks the user to input some of the information about the calendar that we will later be able to compute directly.

Here is typical output from the program:

```
                July 2009
 Sun    Mon    Tu   Wed    Th   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
```

We will ask the user to enter the name of the month: (”July 2009” in this case), the number of days in the month (here 31) and the day of the week the month starts on (here 3 for Wednesday). We will call these three variables monthName, daysInMonth, and startsOn. Our job is to center the name, to print the header with the names of the days of the week, and then to print a table with the dates under the appropriate days.

The first decision is to settle on how we will print the table. It is easy to generate the numbers that need to be printed: 1 up to daysInMonth. We can do this with either

```
for day in range(1, daysInMonth + 1):
```

or

```
day = 1
while day <= daysInMonth:
      ⋮
      day = day + 1
```

The former seems a little clearer, so we will go with that. We can get the numbers to print in columns in the same way we did in Program 3.4.5: each number is printed with a formatted **print** statement using a fixed column width, followed by a comma. At the completion of each row we do a single **print** statement (with no comma) to terminate the line. Using a column width of 5, this gives us the following code:

```
dayOfWeek = 0
for day in range(1, daysInMonth + 1):
        print( "%5d" % day, end='''')
        dayOfWeek += 1
        if dayOfWeek == 7:
                print( )
                dayOfWeek = 0
```

This almost does what we need, but it makes every month start on Sunday. This is where our startsOn variable comes into play. If a month starts on a Wednesday, as does July 2009 in our example, startsOn will be 3. We want to print blanks corresponding to the first 3 days of the week (Sunday, Monday and

Tuesday) and print a 1 in the column for Wednesday. Note that since we print the numbers with

```
print( "%5d" % day, end='''' )
```

we can print blanks that occupy the same column width with

```
print( "%5s" % " ", end='''' )
```

Here is the loop that prints the blanks:

```
dayOfWeek = 0
for day in range(0, startsOn):
    print( "%5s" % " ", end='''' )
    dayOfWeek += 1
```

The rest of our program is easy. We need to print a header with the names of the days of the week. If we put them into a list we can use a **for**-loop to walk through it; we print the names with the same column widths as the body of the calendar:

```
for day in ["Sun","Mon","Tu","Wed","Th","Fri","Sat"]:
        print( "%5s" % day, end='''' )
print( )
```

This is so simple because Python gives us good tools for working with lists and strings.

Finally, we need to center the name of the month. Our calendar has 7 columns, each 5 spaces wide, so the calendar is a total of 35 spaces wide. The name will occupy **len**(monthName) of these spaces, leaving 35−**len**(monthName) left over. We want to print half of the leftover spaces prior to the name. There is no easy way to use the formatted **print**-statements with a variable width, so we use the * operator to make a string with the right number of blanks:

```
leftover = 35-len(monthName)
print( " "*(leftover/2), end='''')
print( monthName )
```

Here is the full program, with some comments inserted to make it more readable:

```
# This prints the calendar for one month

def main():
   monthName = input( "Name of month: " )
   daysInMonth = eval(input("Number of days in the month: "))
   print( "What day of the week does the month start on?" )
   startsOn=eval(input("Enter 0 for Sunday, 1 for Monday, etc: "))

    # First we center the month name.
   leftover = 42-len(monthName)
   print( " "*(leftover/2), end='''')
   print( monthName )

    # Then print the names of the days of the week,
    # in columns:
   for day in ["Sun","Mon","Tu","Wed","Th","Fri","Sat"]:
      print( "%5s" % day, end='''')
   print( )

    # This indents 5 blanks for each of the days
    # of the week prior to the day this month starts on:
   dayOfWeek = 0
   for day in range(0, startsOn):
      print( "%5s" % " ", end='''')
      dayOfWeek += 1

    # Finally, we print the days in the month,
    # with a line break after each Saturday:
   for day in range(1, daysInMonth + 1):
      print( "%5d" % day, end='''')
      dayOfWeek += 1
      if dayOfWeek == 7:
         print( )
         dayOfWeek = 0
   if dayOfWeek != 0:
      print( )

main()
```

Program 3.5.3: Calendar Program: FirstVersion

3.6 Break-statements and Continue-statements

Break-statements

Both **for**-loops and **while**-loops are set up to exit only at the beginning or end of the loop. Once we start executing the body, the entire body is executed. Sometimes that isn't what you need to do. Suppose, for example, that you have a loop that reads names, and follows each name with a greeting. Suppose further that we want the loop to end when we get the empty string for a name. Since this is an indefinite situation (we don't know in advance how many names we will see, we need a **while**-loop for this. It is tempting to write the following code:

```
done = False
while not done:
    name = input( "Enter a name: " )
    print( "Hi %s!" % name )
    if name == "":
        done = True
```

This works, to some extent, but when we enter the empty string to exit it still prints
"Hi !"
What we really want is to check the exit condition as soon as we read the name, and to leave the loop as soon as this condition becomes True. Since the loop only checks this condition at the top, we need to put the rest of the loop body inside an **if**-statement:

```
done = False
while not done:
    name = input( "Enter a name: " )
    if name == "":
        done = True
    else:
        print( "Hi %s!" % name )
```

For a short loop this is simple and clear. As our programs become more complicated, it will sometimes be useful to exit from a loop immediately without putting the rest of the loop in a conditional statement. The **break**-statement is designed for this. The statement **break** causes execution to immediately leave the innermost loop in which it occurs. For example, the loop above could be written:

```
while True:
    name = input( "Enter a name: " )
    if name == "":
        break
    print( "Hi %s!" % name )
```

The "while True" header of this loop indicates that the loop body will handle the termination of the loop with a **break**-statement. Note that there is no **else**-clause on the **if**-statement. This isn't needed, because if the condition is True we will exit from the loop and never get to the **print**-statement.

Break-statements work with **for**-loops just as they do with **while**-loops. For example, the following code will print a statement about sending money to John and Paul, but not to George or Ringo:

```
for name in ["John", "Paul", "George", "Ringo"]:
    if name == "George":
        break
    print( "Send money to %s" % name )
```

If your code has loops inside loops, **break**-statements only exit from the innermost loop inside which they occur. Consider the following code fragment:

```
while True:
    name = input( "Enter a name: " )
    if name == "":
        break
    firstName = ""
    for letter in name:
        if letter == " ":
            break
        firstName = firstName + letter
    print( firstName )
```

Here there are two loops. The outer loops handles the input of names, and exits when it sees a blank name; the first **break**-statement handles this termination. The second loop, inside the first, builds up a string firstName by walking along the letters of name until it gets to the end or to a blank. When it gets to a blank, the **break**-statement exits it from the inner loop, causing firstName to be printed, but the outer loop is not ended, so the program will again ask for another name.

Continue-statements

It is very common for a loop to generate values (perhaps by asking the user for them) and then do something with these values. For certain values there might not be anything to do. The **continue**-statement is designed for this situation.

When a **continue**-statement is executed, control goes back to the top of the loop. For example, the following code prints the odd numbers between 1 and 10:

```
for x in range(1, 10):
    if x % 2 == 0:
        continue
    print( x )
```

In practice **continue**-statements are used less frequently than **break**-statements but they are occasionally handy. Of course, we could always get the same effect with a conditional statement:

```
for x in range(1, 10):
    if x % 2 != 0:
        print( x )
```

3.7 Loop Patterns

Input Loops

Both of the following patterns read strings until they get a blank input.

```
done = False
while not done:
    x = input("Enter something: " )
    if x == "":
        done = True
    else:
        <whatever you want to do with x>
```

or

```
while True:
    x = input("Enter something: " )
    if x == "":
        break
    <whatever you want to do with x>
```

Process elements of a sequence

```
for x in <sequence>:
    <whatever you want to do with x>
```

or

```
x = <first element of the sequence>
while <x is in the sequence>:
    <whatever you want to do with x>
    x = <next element of the sequence>
```

The second version usually involves using a numeric index to refer to the first element of the sequence (index 0), the second element (index 1) and so forth. Where it makes sense the first version is usually easier.

Do something until a condition is satisfied

```
while not <condition >:
    <something>
```

or

```
while True:
    if <condition >:
        break
    something
```

This is equivalent to the first version; it uses a **break**-statement to exit from the loop

or

```
done = False
while not done:
    <something>
    if <condition >:
        done = True
```

This allows the <condition> to be created by the action <something>.

or

```
while True:
    <something>
    if <condition >:
        break
```

This is completely equivalent to the preceding version; it just uses a **break**-statement to exit from the loop. Note that this differs from the second version only in the other of the statements.

3.8 Exercises

Write a complete Python program to solve each of the following problems.

3.1. Write a program that inputs a number and says if it is "nice" or not. If the number is even it is nice if it is divisible by 100. If the number is odd it is nice if it is divisible by 25. Here is a typical run

```
Enter a number: 125
That is a nice number!
```

3.2. Now modify the program from (Problem 3.1) by wrapping a loop around it. Your program should quit when the user enters the number 0:

```
Enter a number: 125
That is a nice number!
Enter a number: 350
That number is not nice.
Enter a number: 5400
That is a nice number!
Enter a number: 0
goodbye!
```

3.3. Write a program that inputs a sequence of strings, terminated by the empty string. For each string the program should say whether it contains "bob" as a substring. You can use the **in** operator to check for substrings: "bob" **in** s is True if string s contains "bob" as a substring and False if it doesn't.

3.4. Write a program that reads numbers from the user, and then prints their average. As you know, the average is the sum of the numbers divided by how many there are. Be sure to make the sum be a float, not an integer; one way to do this is to start it off at 0.0.

```
Enter a number: 4
Enter a number: 12
Enter a number: 8
Enter a number: 0
The average of those numbers is 8.00
```

3.5. Revise the program from Problem 3.4 so that in addition to the average it prints the largest and smallest of the values it has seen. For example:

```
Enter a number: 4
Enter a number: 12
Enter a number: 8
Enter a number: 3
Enter a number: 13
Enter a number: 11
Enter a number: 0
The average of those numbers is 8.50
The largest is 13 and the smallest is  3.
```

3.6. Write a program that reads strings over and over until it sees the word "quit", at which point it prints "Goodbye" and stops. While it is reading strings, each time it sees the string "bob" it replies "BOB RULES!" It igores all other words except "quit".

```
Enter a string: foobar
Enter a string: Oberlin
Enter a string: Bob
Enter a string: bob
BOB RULES!
Enter a string: Harry Potter
Enter a string: quit
Goodbye
```

3.7. Here is a harder one. Write a program that inputs a string and says if it is a palindrome: a word that is the same forwards and backwards, like "bob" or "anna" or "amanaplanacanalpanama". One way to do this is to have a counter start at the front of the string and another start and the end. At each step you compare the letter at each counter; if these are the same you increment the front counter and decrement (subtract 1 from) the back counter. Continue on until you find a difference or the counters cross.

3.8. Write a program that finds all of the perfect numbers less than 100,000. A number is called *perfect* if its factors (including 1 but not the number itself), sum to the number. For example, 6 is perfect because its factors are 1,2, 3 and 6 and $1 + 2 + 3 = 6$.

3.9. Write a program that computes the factorial of integer `n`. This is the product of the numbers from 1 to `n`. For example, the factorial of 5 is $1 \times 2 \times 3 \times 4 \times 5 = 120$. Factorials quickly become very large; even 10 factorial, which is 3628800, has 7 digits. See what happens if you try to compute 100 factorial. Add a feature to your program to print the number of digits in the factorials in addition to their values.

Chapter 4

Functions

It is theoretically possible to write any program using only loops, conditional and assignment statements. In practice, it is difficult to write any program much longer than a page using only these statements. We need a tool that will allow us to assemble a large program from smaller pieces. That is the role of functions in programming. Once you have mastered functions and the art of programming with them you can write programs to implement almost any algorithm you can think of.

4.1 Concepts

A *function* is a block of code that has a name attached to it. We execute this code by *calling* the function. In Python we call a function by giving its name, followed by open and closed parentheses. For example, if a function is named foo, we call it with foo(). We may put values inside the parentheses; this is the way inputs are given to the function. We call values put in the parentheses *arguments* to the function. In the expression

```
bar(3, 5)
```

a function named bar is being called with arguments 3 and 5.

There are two kinds of functions: functions that return values and functions that do not return values. These two kinds of functions are used in very different ways. When a function *returns* a value, you can use a call to this function as a placeholder for the value the function returns. For example, we might have a function addOne that takes a numeric argument and returns that argument plus 1. This means that addOne(5) returns the value 6. We might have a line of a program x = addOne(5). The effect of this line is to set variable x to 6. The call to function addOne produces a value and this call can be used anywhere we can use that value. We could even use it as an argument for another call to addOne: y = addOne(addOne(5)) sets variable y to 7. When a function does not return a value we use a call to it as a statement by itself. For example, we might have a function printCalendar that takes a month and a year as arguments and prints a calendar for that month. We could call this function as printCalendar (9, 2011) to print a calendar for September, 2011. We would not say x = printCalendar(9, 2011) because this function does not return a value.

When writing about functions it is customary to put parentheses after the name of the function to emphasize the fact that this is a function. Unless there is a reason to discuss the arguments, we often omit them from the name. In the examples above, we might discuss the functions addOne() and printCalendar(). We have already used a number of standard Python functions. **input**() is a function that takes an optional string argument and returns a string typed by the user. **range**() is a function that takes one, two, or three integer arguments and returns a list of numbers. **len**() is a function that takes a sequence and returns its length. We have seen a function **print**() that prints its argument (and doesnt return anything). Note that in all of these examples we can use the function without knowing how it is written. All that matters is what inputs the function takes and what value, if any, it returns. One great advantage of functions is that once they are written we can use them without thinking about their details, only about their inputs and outputs. This is what allows us to write long programs. Functions *encapsulate* a process so that we may concentrate on what it does rather than how it works.

4.2 Function definitions – the basics

There are three questions you must answer before you can write a function definition:

- What will the function do?
- What inputs does the function need?
- What, if anything, will the function return?

The answers to all of these should be clear from the way you use the function. It is often easiest to write the function call before you write its definition. This might seem backwards the definitions usually appear before the calls in the program code. However, modern editors make it easy to move around in the code; there is no reason that we need to write the program in the same order in which it is read.

A function definition has the form:

```
def <name>( <parameters> ):
    <body>
```

In other words, the definition

- starts with the word **def**;
- has the name of the function. This should be a unique name in your program. Python will allow you to give a function the same name as a variable, but the result might not work the way you expect. To emphasize the difference, some people capitalize the first letter of function names and have variable names start with lower-case letters;
- has a list of parameters inside parentheses. There might not be any parameters, but there must be parentheses in any case;
- has a body indented underneath the function header. This body is the code that is executed when the function is called.

There are a number of issues with parameters, so we will start with functions that have no parameters. One way to get data without inputs to the function is to ask the user for it. Here is a function that asks the user to input a string and prints it back out:

```
def ReadAndReport():
    text = input( "Enter a string: " )
    print( "You typed '%s'." % text )
```

The name of this function is ReadAndReport. It has no parameters. When it is called the two lines of the body: a **raw_input** statement and a **print** statement, will be executed. There must be a match between the parameters of a function and the arguments with which it is called. Since this function has no parameters, we call it with no arguments, as in

```
ReadAndReport()
```

Here is a complete program that makes use of function ReadAndReport():

```
def ReadAndReport():
    text = input( "Enter a string: " )
    print( "You typed '%s'." % text )

def main():
    ReadAndReport()
    ReadAndReport()

main()
```

Program 4.2.1: Coding with a function

The last line of this program, outside any of the definitions, calls function main(), so this is where execution starts. Inside function main(), function ReadAndReport() is called twice. Here is a typical run of this program:

```
Enter a string: blah
You typed 'blah'.
Enter a string: La de dah
You typed 'La de dah'.
```

It might be nice to call function ReadAndReport() in a loop, but this function doesn't give any way to signal the loop to quit. We need to return something if we want to exit the loop based on the input. Here is another version of our program that does this:

```
def ReadReportAndSignal():
    text = input( "Enter a string: " )
    if text == "":
        return "No input"
    else:
        print( "You typed '%s'." % text )
        return "Got input"

def main():
    done = False
    while not done:
        response = ReadReportAndSignal()
        if response == "No input":
            done = True
main()
```

Program 4.2.2: Returning a value from a function

Our function ReadReportAndSignal() still reads a string from the user and prints out a message about it. Unlike function ReadAndReport(), the new function returns a message about the string it has read: either the message `"No input"` or the message `"Got input"`. Function main() saves this message in variable response and uses this to decide when to exit from the **while**-loop.

The syntax of a **return**-statement is

```
return <expression>
```

When this is executed the expression is evaluated and control immediately goes back to the point where the function was called and the value of the expression is used as the value of the function call. If the expression is omitted and the statement is just

```
return
```

no value is returned but control goes back to the point of the call.

Here is a third version of our program. This time we incorporate the loop inside the function and use a return statement to break out of the loop and the function:

```
def ReadAndReportLoop():
    while True:
        text = input( "Enter a string: " )
        if text == "":
            return
        else:
            print( "You typed '%s'." % text )

def main():
    ReadAndReportLoop()

main()
```

Program 4.2.3: Using return to break out of a function

Since function ReadAndReportLoop() does not return any values, we call it as a statement, not as an expression. Note that the return statement exits from the function even when it is nested inside a loop.

As a final example, here is a function that simulates the toss of a coin. It gets a random value that is either 0 or 1, and returns a corresponding value "Heads" or "Tails". This makes use of the randint() function that we introduced in Section 2.4. This function requires us to include the line:

```
from random import *
```

at the top of the program. Recall that randint(a, b) returns a random integer between a and b, so randint(0, 1) returns either a 0 or a 1. Of course, we could replace a call to function CoinToss() with one to randint(0, 1) but that would lose one of the great advantages of writing our own functions: abstraction. When we see the name CoinToss we know that the function represents the tossing of a coin. This is very useful information for a human trying to understand the program, information that is missing from the expression randint(0, 1).

```
from random import *

def Toss():
    x = randint(0, 1)
    if x == 0:
        return "Heads"
    else:
        return "Tails"

def main():
    for i in range(10):
        print( Toss() )

main()
```

Program 4.2.4: Simulating coin tosses

A function call in Python always gives a value, which is None (a formal value in Python) if the function itself does not return anything. You should design functions so that they either return a value, and in that case they should always return something for any input, or else to not return a value in any situation. The following program will print `"small"` if the user enters a number less than 10, and it will print the word `"None"` if the user enters a value 10 or larger. This is very confusing for the user of the program. The problem comes from function BadSize(), which only returns a value in some cases. Design your programs so this doesn't happen.

```
def BadSize(x):
    if x < 10:
        return "small"

def main():
    value = eval(input("Enter a number: "))
    print( BadSize(value) )

main()
```

Program 4.2.4: Function BadSize sometimes returns None

The problem comes from function BadSize(), which only returns a value in some cases. Design your programs so this doesn't happen.

Scoping Rules

People have been writing programs for over 50 years and for most of that time functions have been the main building blocks of programs. One thing we have learned about the art of programming is that code is much more likely to be correct if the functions are as small as possible and independent of each other. It might sound convenient to allow one function to modify the variables of another, but that turns out to be a very bad idea one that is the cause of many, many mysterious program failures. Python does not allow this to happen. Unless you explicitly state that a variable is **global** (something we will discuss later), all of the variables of a function are hidden from all other functions they are accessible only within the function in which they are created.

Consider the following simple program.

```
def Changer():
    x = 23

def main():
    x = 5
    Changer()
    print( x )

main()
```

At first glance it might seem that function Changer() will modify the value of x to 23 and so this program might print 23. That is not correct the program actually prints 5. The reason for this is that function Changer() and function main() each have their own variables named x. Neither function can see the other function's variables. So main() changes its x to 5 and Changer() sets its to 23; these variables are completely unrelated. The **print**-statement occurs in main(), so it is main's x that is printed, with the value of 5.

If you want a function to change the value of a variable in another function you should have the first function return the new value. Here is this program rewritten correctly.

```
def Changer():
    return 23

def main():
    x = 5
    x = Changer()
    print( x )

main()
```

Program 4.2.5: Changing variables with a function call

Now function Changer() returns the new value to main(), which assigns this value to x. Only function main() can see x, so only main() can change its value. Program 4.2.5 prints 23.

Similarly, the following program gives an error message saying that variable x in the line **if** x > 9: of function Printer() is not defined.

```
def Printer():
    if x > 9:
        print( "big" )
    else:
        print( "small" )

def main():
    x = 5
    Printer()

main()
```

No variable x is created for function Printer(), and this function can't see the variable x inside main().

The correct way to get data into a function is to pass it in through a parameter. That is the topic of our next section.

4.3 Parameters and Arguments

Functions represent processes. A function is a named block of code that can be invoked from any point in the program. Much of the power of functions comes from our ability to supply inputs to these processes. The inputs to a function are called its *arguments* and they are represented inside the function definition by variables called *parameters*. In this section we will describe parameters and their relationships to arguments.

As we said in Section 4.2, the parameters are listed inside the parentheses in the function header. For example, we might have the following header to a function:

```
def foo(x, y):
```

This is the start of the definition of a function named foo. The parameters for this function are named x and y. These are variables, just like any other variables used in the function with one exception: variables x and y are given initial values when the function is called. The *call* of foo has the form

```
foo(<first_arg >, <second_arg >)
```

When this is executed < first_arg > is evaluated and its value becomes the starting value for parameter x. Similarly, <second_arg> is evaluated for the initial value of y. The body of foo is then executed until it ends or returns.

As a first example, we will write a function to calculate a 20% tip to add to the bill for a meal. To keep our arithmetic simple at mealtime, in cases where the tip is not an exact dollar amount we will round up to the next dollar. The tip calculation is easy: we get 20% of a value by multiplying by 0.2, and we round up with the ceiling function from the math library (see Section 2.4). For a given cost we compute the tip as ceil (cost*0.2)l our function returns this value:

```
def Tip(mealCost):
    tipAmount = ceil(mealCost*0.2)
    return tipAmount
```

Here is a full program that makes use of this:

```
def Tip(mealCost):
    tipAmount = ceil(mealCost*0.2)
    return tipAmount

def main():
    done = False
    while not done:
        amount=eval(input("Enter a meal cost, or 0 to exit: "))
        if amount == 0:
            done = True
        else:
            print( "Your tip should be $%d."%Tip(amount) )
            print( "The total meal cost is $%.2f."%
                        (amount+Tip(amount)))

main()
```

Program 4.3.1: Computing a tip

Note that the argument to function Tip() has a different name than the function's parameter: the argument is called amount while the parameter is mealCost. This is typical; we name the function's parameter something that is internally descriptive within the function. The argument can be anything we wish to apply the function to. At the time of the call the argument is evaluated and it is this value that is given to the function as an initial value for the parameter; any name present in the argument is irrelevant once its value is retrieved. For example, the following calls to function Tip are all valid:

- Tip(23)
- Tip(3*5+8)

-
  ```
  x = 23
  Tip(x)
  ```

It is generally not a good idea to give the parameters of a function the same names as variables elsewhere in the program, since this gives the impression that these are the same variable. The next program prints 2 as the value of myVariable, even though function Change() seems to change this value to 33.

```
def Change(myVariable):
    myVariable = 33

def main():
    myVariable = 2
```

```
    Change(myVariable)
    print( myVariable )

main()
```

What is happening here is that function Change() and function main() each have their own variables called myVariable and these variables are unrelated; a change to one does not affect the other.

If a function has multiple parameters, the arguments in a call to the function must match the parameters: there must be one argument for each parameter and they must appear in the same order. For example, function PrintMultiples () in the next program has two parameters: the first is a string and the second an integer. We could call this with PrintMultiples ("bob", 5) to print the string "bob bob bob bob bob ", but we couldn't call it with PrintMultiples (5, "bob") or with PrintMultiples ("bob").

```
def PrintMultiples(string, count):
    print( (string+" ")*count )

def main():
    PrintMultiples( "bob", 5)

main()
```

A parameter is just like any other variable of a function, with the one exception that it gets an initial value from the argument at the time the function is called. Like all other variables it can be modified within the function, and like all other variables it is invisible outside the function. It might be tempting to write something like the following, but the function call SetTo23(x) does not change the value of x, in spite of its name.

```
def SetTo23(x):
    x = 23

def main():
    x = 5
    SetTo23(x)
    print x

main()
```

Function SetTo23() changes its own variable x and has no affect on the variable x inside function main(). This program prints 5, not 23.

Examples

We will now give several examples of the way programs can be designed with functions.

First, think back to the prime number programs we wrote in Section 3.4. Each of these contained a block of code that determined whether a given number was prime. This task is clearer if it is performed by a function such as the following:

```
def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
                return False
    return True
```

Note that this function returns a Boolean value that just says if its argument is prime or not.

We can put this into a program that checks all the numbers between 2 and some upper limit and prints the primes. The result is Program 4.3.2:

```
# This prints all of the prime numbers up to
# a limit supplied by the user

def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
                return False
    return True

def main():
    max = eval(input("Enter the largest number of check: "))
    for x in range(2, max+1):
        if IsPrime(x):
            print( "%d is prime." % x )

main()
```

Program 4.3.2: Improved version of Program 3.4.4

Note how much easier to read this program is than Program 3.4.4. The `main()` function consists of an **input**-statement, and a loop with a conditional **print**-statement. The condition on the **print**-statement is almost grammatical English: "`if IsPrime(x)`" is easy to understand. In general, the more natural language we can work into a program the easier it is to read, and the more likely we are to write it correctly in the first place.

Similarly, here is a version of Program 3.4.5, which prints a table of the first N prime numbers. It is tempting to think of adding another function to handle the printing, but this is more complex than it might seem. All of the variables of a function go away after the function call, so a function could not easily keep track of the number of values that are on the current line. One way to handle that might be to build up a list of all of the values for the current line and send them to a function to print; that will need to wait until we have a more thorough treatment of lists in Chapter 5.

```
# This prints the first N prime numbers,
# where the value of N is supplied by the user.
# The output is printed in C columns.

def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
                return False
    return True

def main():
    N = eval(input("How many prime numbers do you want? "))
    C = eval(input("How many columns of output do you want? "))
    x = 2
    primeCount = 0
    lineCount = 0
    while primeCount < N:
        if IsPrime(x):
            print( "%7d " % x, end='''' )
            primeCount = primeCount + 1
            lineCount = lineCount + 1
            if lineCount == C:
                print( )
                lineCount = 0

        x = x + 1
    if lineCount != 0:
        print( )

main()
```

Program 4.3.3: Improved version of Program 3.4.5

Here is a third example with prime numbers. This time we will find *twin primes*: pairs of consecutive odd numbers that are both prime, such as 11 and 13. There is an unproven conjecture in Mathematics claiming that there are infinitely many twin prime pairs. Our program would require some contorted code if we only used loops, but with our IsPrime() function it is quite simple. Our condition for printing a pair (x, x+2) is just that both x and x+2 are prime: **if** IsPrime(x) **and** IsPrime(x+2)

```
# This program looks through the numbers up to a limit
# for pairs of twin primes.

def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
                return False
    return True

def main():
    print( "This program looks for twin primes." )
    max = eval(input("Enter the largest number of check: "))
    for x in range(3, max+1):
        if IsPrime(x) and IsPrime(x+2):
            print("(%d, %d) are twin primes." % (x, x+2))

main()
```

Program 4.3.4: Finding twin primes

Note that the function IsPrime() is identical in each of these programs. Because we write functions as stand-alone objects in programs, function definitions can frequently be copied from one program to another. Once a function is written and debugged, you can use it in any programming situation where it is appropriate. This makes the job of the programmer easier, and makes our programs much more likely to be correct.

As a final example for this section we will write a program that inputs strings and determines whether they are palindromes the same forwards as backwards, such as "bob" or "racecar". To make this more interesting, palindromists usually ignore white space, punctuation and capitalization. Probably the most famous palindrome in English is "A man, a plan, a canal: Panama." Here are some others: "Dammit, I'm mad!", "Step on no pets.", "Was it a rat I saw?" and "Are we not drawn onward, we few, drawn onward to new era?"

Our strategy in developing this program is to write it one step at a time.

Whenever we need a new activity in the program we will create a function to handle it, first writing the call to the function and then writing the function itself. The steps are:

a. A loop that prompts the user for input, reads strings, and decides if each string is a palindrome. This should also print the result for the user.

b. To determine whether a string is a palindrome, we can reverse it and compare that to the original.

c. To handle punctuation and white space, we can walk through the letters of the string and remove anything that isn't a letter. Since strings are immutable, we build up a new string consisting of just the letters of the original.

d. To handle capitalization, as we build up the new string we insert the lower-case form of each letter. The string method `lower()` is useful for this: if `s` is a string, `s.lower()` is a string with the same letters as `s`, all converted to lower-case.

The first step is a straightforward application of the input loop model from Section 3.7:

```
def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if <the phrase is a palindrome>:
                print( "'%s' is a palindrome."%
                         phrase)
            else:
                print( "'%s' is not a palindrome."%
                         phrase )
```

Program 4.3.5: Palindromes - First Version

This has pseudo-code for the condition on the print statement:

```
if <the phrase is a palindrome>:
```

We can define a function to handle this check. To keep its name similar to its usage in English, we will call the function `IsPalindrome()`. Its job is to determine whether the variable `phrase` holds a palindrome; `phrase` is the only input it needs. This means we can state our condition as

```
if IsPalindrome(phrase):
```

Our function definition needs a parameter as a placeholder for the argument. Since this is a string we will just use the letter s:

```
def IsPalindrome(s):
```

To conclude the first step, we will make this function return True for all strings. This allows us to make sure the initial part of our program is working correctly. The following program isn't complete, but it is a working program. It allows the user to enter strings until a blank string is given. It claims that each string is a palindrome.

```
def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    return True

def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if IsPalindrome(phrase):
                print("'%s' is a palindrome."%phrase)
            else:
                print("'%s' is not a palindrome."%phrase)

main()
```

Program 4.3.5: Palindromes continued

Our next step is to make the IsPalindrome() function more useful. We defined a palindrome as a string that reads the same forwards and backwards. This is easy to code into a function:

```
def IsPalindrome(s):
    if s == Reverse(s):
        return True
    else:
        return False
```

This involves another function: Reverse(s) is a function we will write that returns the reversal of string vars. How do we do that? It is easy to use a **for**-loop to walk through the letters of a string. We want to build up a new string, which we will call answer, that is the reversal of s. Variable answer starts off as the empty string. At each step we add the new letter onto answer. If we add it onto the end of answer, we will make an exact copy of string s. If we add the new letter onto the beginning of answer, we get the reversal of string s. If you don't see this, note that the last letter of string s will become the first letter of the answer, the next-to-last letter of s will be the second letter of the answer, and so forth.

```
def Reverse(s):
    answer = ""
    for c in s:
        answer = c + answer
    return answer
```

Adding this to our program gives the first version of our palindrome checker:

```
def Reverse(s):
    # This returns the reversal of string s:
    # if s is 'abc' this returns 'cba'.
    answer = ""
    for c in s:
        answer = c + answer
    return answer

def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    s = StripPunctuation(s)
    if s == Reverse(s):
        return True
    else:
        return False

def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if IsPalindrome(phrase):
                print("'%s' is a palindrome."%phrase)
            else:
                print("'%s' is not a palindrome."%phrase)

main()
```

Program 4.3.5: Palindromes Step 2

For the last steps, we need to eliminate the punctuation. A new function StripPunctuation() will handle that; this function will take a string s and return a new string consisting of just the alphabetic letters of s, all in lowercase. We should think carefully about where to call this function. We could call it in main(), as soon as we read the string and see that it is not empty:

```
phrase = input( "Enter a string: " )
if phrase == "":
        done = True
else:
        phrase = StripPunctuation(phrase)
        if IsPalindrome(phrase):
```

⋮

This is not good. The user will enter a phrase such as "Dammitit I'm mad!" and the program will respond that `'dammitimmad' is a palindrome.` This is confusing because 'dammitimmad' is not what the user entered. A more appropriate placement for the call to StripPalindrome() is within the IsPalindrome() function. It is this function that is trying to decide if a string is a palindrome. Anything it does to the string will be invisible to the main() function. Accordingly, we will keep main() as it stands and change the IsPalindrome() function to:

```
def IsPalindrome(s):
    s = StripPunctuation(s)
    if s == Reverse(s):
        return True
    else:
        return False
```

All that remains is to write StripPunctuation(). As with Reverse(), this walks through the letters of string s and places them in an answer string. This time we add the new letters onto the end of answer, so that we are making a copy of s. We add only the alphabetic letters to delete the punctuation and white space elements. How do we tell which are the alphabetic characters? One way to test of variable c contains a lower-case letter is to determine if it is between 'a' and 'z':

```
if 'a' <= c and c <= 'z':
```

We could do the same thing with upper-case letters and get code that starts

```
if ('a' <= c and c <= 'z) or ('A' <= c and c <= 'Z'):
```

Another way is to use a string method that tests if all of the elements of the string are alphabetic::

```
if c.isalpha():
```

Finally, we use the string method lower() to insure that we add the lower-case version of each letter to answer. Here is the code for function StripPunctuation():

```
def StripPunctuation(s):
    answer = ""
    for c in s:
        if c.isalpha():
            answer = answer + c.lower()
    return answer
```

This completes the program. Here is complete code for our palindrome program, including comments to make it more readable:

```
# This program reads strings from the user and
# says if they are palindromes: the same when
# read backwards as when read forwards

def StripPunctuation(s):
    # This returns a string just like s only all
    # of the non-letters are removed and the letters
    # are all changed to lower-case.
    answer = ""
    for c in s:
        if c.isalpha():
            answer = answer + c.lower()
    return answer

def Reverse(s):
    # This returns the reversal of string s:
    # if s is 'abc' this returns 'cba'.
    answer = ""
    for c in s:
        answer = c + answer
    return answer

def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    s = StripPunctuation(s)
    if s == Reverse(s):
        return True
    else:
        return False

def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if IsPalindrome(phrase):
                print("'%s' is a palindrome." % phrase)
            else:
                print("'%s' is not a palindrome." % phrase)

main()
```

Program 4.3.5: Palindromes: final version

4.4 The Calendar program

To illustrate the power of functions, in this section we will develop a useful program that allows the user to input a date or a month or a year. For a specific date the program will respond with the day of the week on which this date occurred or will occur. For example, 2 16 1952 was a Saturday (and my birthday) while 12 7 1941 was a Sunday (and the day on which Pearl Harbor was attacked). If the user inputs a month the program will respond with a calendar for that month. For a year the program responds with 12 calendars for the 12 months of that year. This is a substantial program with 14 different functions, but its development is rather straightforward.

Our program is based on one fact and a simple idea. The fact is that January 1, 1800 was a Wednesday. We use this as our starting date because it comes after the adoption of the Gregorian calendar by England and its colonies in 1752. Wednesday, September 2 1752 was followed by Thursday, September 14 1752 to bring the civil calendar more in line with the seasons. There have been no significant changes to the Western calendar since.

The idea we will use is to notice that if the number of days between two dates is divisible by 7 then these dates must occur on the same day of the week. For example, August 12 and August 26 differ by 14 days; in 2009 they both occur on a Wednesday while in 2005 they both are on a Friday. For our purposes "days between" will mean all days including the first date but not including the second, so the 7 days between August 12 and August 19 are numbered 12, 13, 14, 15, 16, 17, and 18. Our algorithm comes down to counting the number of days between January 1, 1800 and the date we are interested in, and taking this count mod 7. If the result is 0 the date occurs on a Wednesday, if it is 1 the date occurs on a Thursday, and so forth.

The first step is to get input from the user. We know the form of an input loop; we'll read dates and respond to them until we get a signal that the user is done with the program. But how do we read a date? There are lots of possible formats. We could ask the user for a year, then a month in that year, then a date in that month, but this seems very awkward and slow. A more intuitive format from the user's point of view would be to enter all of the information on one line: a user could input

```
9 16 2009
```

to learn that September 16, 2009 occurs on a Wednesday. We will store the input values in a list, where the individual entries are numbers are numbers. Lists are written inside parentheses, as in [9, 16, 2009]. The algorithm for reading the date is similar to a lot of Python input algorithms. We will first read the entire line of input, because reading a line of text as a string is easy. There is a string method called `split ()` that divides a string into "words", using white space as the delimiter between words. If `s` is the line of input, `s. split ()` is a list of the words (which in our case are the numbers) of `s`. A **for**-loop can run through this list; we can use the **int**() function to convert each word into its numeric value. Altogether, we get the following function for reading a date:

```
def ReadDate():
    s = input( "Enter a date in the form m d y, \
or a blank to exit: " )
    t = []
    for x in s.split():
        t = t + [int(x)]
    return t
```

This function always returns variable t, which starts as an empty list, and has numbers added to it. If there is a no input, as will happen when the user enters a blank line, function readDate() returns an empty list. We can use this as the exit condition for our program.

Our main() function is our usual input loop, with format:

```
def main():
    done = False
    while not done:
        date = ReadDate()
        if  len(date) == 0:
           done = True
        else:
            <print the day for this date>
```

We will assign a function PrintDay() to the task of printing the day of the week for a particular date. The information that needs to be sent to this function are the month, day, and year of the date: these are the first, second, and third fields that function ReadDate() returns:

```
def main():
    done = False
    while not done:
        date = ReadDate()
        if  len(date) == 0:
            done = True
        elif len(date) == 3:
            PrintDay(date[0], date[1], date[2])
        else:
            print( "Bad date." )
```

Function PrintDay() holds the basic activity of this program: it takes as arguments a month, day, and year, and prints the day of the week corresponding to this date. We will divide this into two steps. The hard part is computing the numeric day of the week: 0 for Sunday, 1 for Monday, and so forth. We'll create a function FindDay() for this. Once we have that number, we will give it to a simple function DayName() to print as a string. This makes function PrintDay() particularly easy:

```
def PrintDay(m, d, y):
    # This prints the day of the week (m,d,y) falls on
    day = FindDay(m, d, y)
    print( "That was a %s" % DayName(day) )
```

The DayName() function is also easy:

```
def DayName(d):
    #This returns a string for the name of day d
    if d == 0:
        return "Sunday"
    elif d == 1:
        return "Monday"
    elif d == 2
        ...
        <and so forth>
```

The real work is in writing function FindDay(). This starts by finding the number of days between January 1, 1800 and day (m, d, y); we will us a function CountDays() to compute this. We will store this count in variable days. We will include January 1, 1800 in this count but not day (m, d, y). Recall that January, 1, 1800 fell on a Wednesday; every 7 days after this we are back to a Wednesday. This means that if days%7 is 0, then (m, d, y) also falls on a Wednesday. On the other hand, if days % 7 is 1, then (m, d, y) falls on a Thursday; if it is 2 then (m, d, y) is a Friday, and so forth. Here is the code that comes from this analysis.

```
def FindDay(m, d, y):
    # This counts the number of days between (1,1,1800)
    # and (m,d,y), takes the remainder %7, and
    # returns 0 for Sundays, 1 for Mondays, etc.
    days = CountDays(m, d, y)
    day = days%7
    if day == 0:
        return 3  # Wednesday is day 3 of the week
    elif day == 1:
        return 4 # Thursday
        ...
    elif day == 3:
         return 6   # Saturday
    elif day == 4:
        return 0   # Sunday
       ...
<and so forth>
```

Function CountDays() is easier than you might think. Remember that this function counts the number of days between January 1, 1800 and day (m, d, y). We can break this into three steps:

- The sum of all the days of all the years starting with 1800, up to but not including year y (we don't want to include all of year y). We keep a running total of these, adding 365 for non-leap years and 366 for leap years.

- The sum of all the days of all the months of year y starting with January, up to but not including month m. We add each of these onto the total.

- The number of days of month m up to, but not including day d.

The first two of these we accomplish with a loop; the third by adding d−1 onto our total. The following code does this; we use helper-functions to compute the number of days in a year (365 if it isn't a leap year, 366 if it is) and the number of days in a month (31 for January, 28 for non-leap year Februarys, 29 for leap year Februarys, 31 for March, etc.):

```
def CountDays(m, d, y):
    # This returns the number of days between
    # (1, 1, 1800) and (m,d,y).
    days = 0
    for year in range(1800, y):
        days = days + DaysInYear( year )
    for month in range(1, m):
        days = days + DaysInMonth(month, y)
    days = days + d-1
    return days
```

Notice how we can use the ability of a computer to execute many small steps quickly to solve a problem in a way that human would think of solving it. Part of the art of programming is learning to make use of this ability.

The rest of the program is completely straightforward. The complete program is given below. This looks like a lot of code, but the development of it was not difficult. At each step we write the code for one small portion of the program. If this is more than just a few lines of code, we break it into pieces and assign a function to write each piece. Note that we write the call of the function, and then go back and write the function itself. As long as we have a clear vision of what each function should do (comments help us to keep that straight), the individual pieces will all fit together this way we designed them precisely to do that.

```
# This program finds the day of the week corresponding to
# any specific (m,d,y) date after January 1, 1800.

def IsLeapYear(y):
    # This returns True if y is a leap year, False otherwise
    if y % 400 == 0:
        return True
    elif y % 100 == 0:
        return False
    elif y % 4 == 0:
        return True
    else:
        return False

def DaysInMonth(m, y):
    # This returns the number of days in month m of year y.
    # Of course, the year only matters for February
    if m == 1:
        return 31
    elif m == 2:
        if isLeapYear(y):
            return 29
        else:
            return 28
    elif m == 3:
        return 31
    elif m == 4:
        return 30
    elif m == 5:
        return 31
    elif m == 6:
        return 30
    elif m == 7:
        return 31
    elif m == 8:
        return 31
    elif m == 9:
        return 30
    elif m == 10:
        return 31
    elif m == 11:
        return 30
    elif m == 12:
        return 31
    else:
        print( "daysInMonth: No such month %d" % m )
```

Program 4.4.1: The Calendar Program - beginning

```
def DayName(d):
    #This returns a string for the name of day d
    if d == 0:
        return "Sunday"
    elif d == 1:
        return "Monday"
    elif d == 2:
        return "Tuesday"
    elif d == 3:
        return "Wednesday"
    elif d == 4:
        return "Thursday"
    elif d == 5:
        return "Friday"
    elif d == 6:
        return "Saturday"

def DaysInYear(year):
    # This returns the number of days in the given year.
    if IsLeapYear(year):
        return 366
    else:
        return 365

def CountDays(m, d, y):
    # This returns the number of days between
    # (1, 1, 1800) and (m,d,y)
    days = 0
    for year in range(1800, y):
        days = days + DaysInYear( year )
    for month in range(1, m):
        days = days + DaysInMonth(month, y)
    days = days + d-1
    return days
```

Program 4.4.1: The Calendar Program continued

```
def FindDay(m, d, y):
    # This counts the number of days between (1,1,1800)
    # and (m,d,y), takes the remainder %7, and returns
    # 0 for Sundays, 1 for Mondays, etc.
    days = CountDays(m, d, y)
    day = days%7
    if day == 0:
        return 3
    elif day == 1:
        return 4
    elif day == 2:
        return 5
    elif day == 3:
        return 6
    elif day == 4:
        return 0
    elif day == 5:
        return 1
    elif day == 6:
        return 2
    else:
        print( "FindDay: bad day %d" % day )

def PrintDay(m, d, y):
    # This prints the day of the week (m,d,y) falls on
    day = FindDay(m, d, y)
    print( "That was a %s" % DayName(day) )

def ReadDate():
    # This asks the user for a date, then makes a tuple
    # of numbers out of the user's response.
    print( "Either enter a date in the form m d y" )
    s = input( "or enter a blank to exit: " )
    t = []
    for x in s.split():
        t = t + [int(x)]
    return t

def main():
    done = False
    while not done:
        date = ReadDate()
        if  len(date) == 0:
            done = True
        elif len(date) == 3:
            PrintDay(date[0], date[1], date[2])
        else:
            print( "Bad date." )

main()
```

Program 4.4.1: The Calendar Program continued

4.5 Recursion

One of the most powerful programming techniques involves a function calling itself; this is called *recursion*. It is not immediately obvious that this is useful; take that on faith for now and concentrate first on how to make recursion work. The utility should become apparent as you see it in action.

Here is a complete program that will serve as our first example of recursion:

```
def Factorial(n):
    if n <= 1:
        return 1
    else:
        return n*Factorial(n-1)

def main():
    print( Factorial(4) )

main()
```

Program 4.5.1: The recursive Factorial function

Before we think about what the program does, let's concentrate on the Factorial () function:

```
def Factorial(n):
    if n == 1:
        return 1
    else:
        return n*Factorial(n-1)
```

Can we see what this function will return for Factorial (1)? Sure; if n is 1 then the first line: if n == 1 is True so Factorial (1) returns 1.

What about Factorial (2)? Well, if n is 2 then the test n==1 fails and the function returns 2∗ Factorial (1). But we just saw that Factorial (1) returns 1, so Factorial (2) must return 2.

Factorial (3)? If n is 3 then again n==1 is False and the function returns 3∗ Factorial (2). We just saw that Factorial (2) returns 2, so Factorial (3) must return 6.

One more time. If n is 4 then n==1 is False and the function returns 4∗ Factorial (3). We just saw that Factorial (3) returns 6, so Factorial (4) must return 24. If we now look at our program, in main() it prints Factorial (4), so the program prints the number 24 and halts.

Of course, you have surely noticed that Factorial (2) is 2, which is 2∗1, Factorial (3) is 6, which is 3∗2∗1, and Factorial (4) is 24, which is 4∗3∗2∗1. Our function is indeed computing what mathematicians call the Factorial () function. Factorial (n) is the product of the first n positive integers. This should make sense: if Factorial (n−1) is the product of the first n−1 positive integers, we have defined the function so that Factorial (n) is n times the product of the first n−1 positive integers, and that is the product of the first n positive integers.

Notice that the code for our Factorial () starts with an **if** statement one one of whose branches the function returns without calling itself. The values for which a function returns without recursing are called *base cases*. We computed Factorial (4) by starting at the base case Factorial (1) and working up until we reached the Factorial (4) case. Can we go the other way – computing Factorial(4) the way the computer does?

We start with a definition straight from the function:

```
Factorial(4)=4*Factorial(3)
```

We don't yet know the value of Factorial (3), so we remember where we are and start a new computation:

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
```

We don't know the value of Factorial (2), so again we remember where we are and start a new computation:

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
                              Factorial(2)=2*Factorial(1)
```

Finally, we do know the value of Factorial (1):

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
                              Factorial(2)=2*Factorial(1)
                                             Factorial(1)=1
```

We can now work our way back up. Since we know Factorial (1)=1 we can compute Factorial (2):

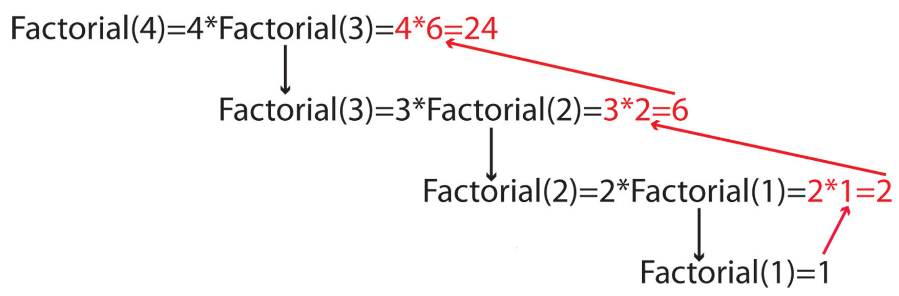
```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
                              Factorial(2)=2*Factorial(1)=2*1=2
                                             Factorial(1)=1
```

and now we can compute Factorial (3):

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)=3*2=6
                              Factorial(2)=2*Factorial(1)=2*1=2
                                             Factorial(1)=1
```

and finally

```
Factorial(4)=4*Factorial(3)=4*6=24
               Factorial(3)=3*Factorial(2)=3*2=6
                              Factorial(2)=2*Factorial(1)=2*1=2
                                             Factorial(1)=1
```


Factorial(4)=4*Factorial(3)=4*6=24
Factorial(3)=3*Factorial(2)=3*2=6
Factorial(2)=2*Factorial(1)=2*1=2
Factorial(1)=1

Follow the black arrows down into the recursion, then the red arrows back out.

You can imitate the way a computer executes any recursive function in this way. It becomes a bookkeeping problem – the difficulty is just in keeping track of what has been evaluated and where you are in the computation.

Notice that our recursive factorial function started with an if–statement. This is true of almost all recursive functions. The first question to ask is whether the argument to the function is one of the base cases. If so we can just return the answer; it is only when the argument is not a base case that we need to recurse. Notice also that in order for the recursive function call to terminate, the argument in the recursive call must be closer to a base can than the initial argument was. To compute `Factorial(n)` we recurse on `Factorial(n−1)`. If `n` is larger than 1, `n−1` is closer to the base case of 1 than `n` is. On the other hand, if `n` is not positive then `n−1` moves farther away from 1, and as a result the `Factorial()` function never halts if you give it a non-positive argument.

The aspect of recursion that students find hardest is writing recursive functions. This almost seems like magic. Here is a way to think about the process:

- Find the base case. This usually tells you what the parameter of the function should be. Start the recursive function with an if-statement that describes how to perform the computation for the base case.

- Handle the recursive case. Describe how to perform the computation if the parameter is not the base case, in terms of parameters that would be closer to the base case.

- Be sure that the recursion always works down to one of the base cases.

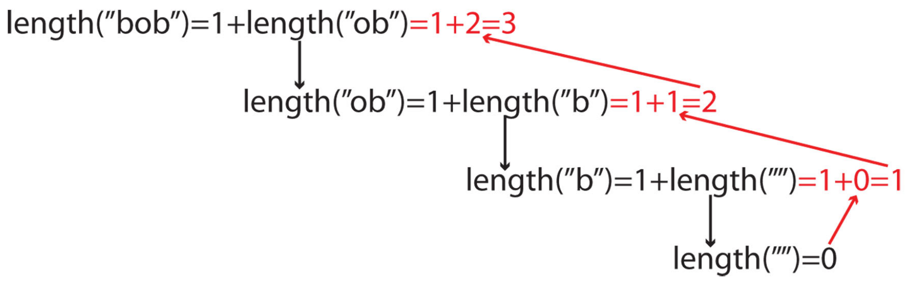
Example 1: *Write a recursive function that returns the length of a string.* Strings can be long: "Marvin Krislov is President of Oberlin College", or short: "bob". Short ones seem easier to deal with than long ones. The shortest possible string is "", the *empty* string. We'll take that as our base case. If *the string is empty* is our base case, our recursive parameter must be the string and our function starts

```
def length(string):
    if string == "":
        return 0
    else:
```

Now we need to handle the recursive cases: the cases where the base case does not apply. This means we have a string whose length is not 0. The length can't be negative, so our string must have positive length. To make use of recursion, we want to find its length in terms of the lengths of strings closer to the base case: i.e., string with shorter length. With a little thought you can see that the length of the string is 1 more than it would be if we removed one letter. One way to remove a letter is to make use of our string indexing operations: string [1:] consists of all of the letters of string after the first. We add this to our code with one line:

```
def length(string):
    if string == "":
        return 0
    else:
        return 1 + length(string[1:]
```

That is our function. Does it always lead us to a base case? Sure; for any starting string we remove one letter at a time until we get to the empty string:


length("bob")=1+length("ob")=1+2=3
length("ob")=1+length("b")=1+1=2
length("b")=1+length("")=1+0=1
length("")=0

Example 2: *Write a recursive function to make the Fibonacci sequence 0,1,1,2,3,5,8,13.....* The Fibonacci sequence is easy to describe in English: it starts with 0 and 1, and then every subsequent number is the sum of the two previous numbers. Since "starts with 0 and 1" is a statement about position in the sequence, we will write a function Fib(n), where n refers to position in the sequence: Fib(0) is the first number in the sequence, Fib(1) is the second, and so forth. Our English description of the problem gives us two base cases:

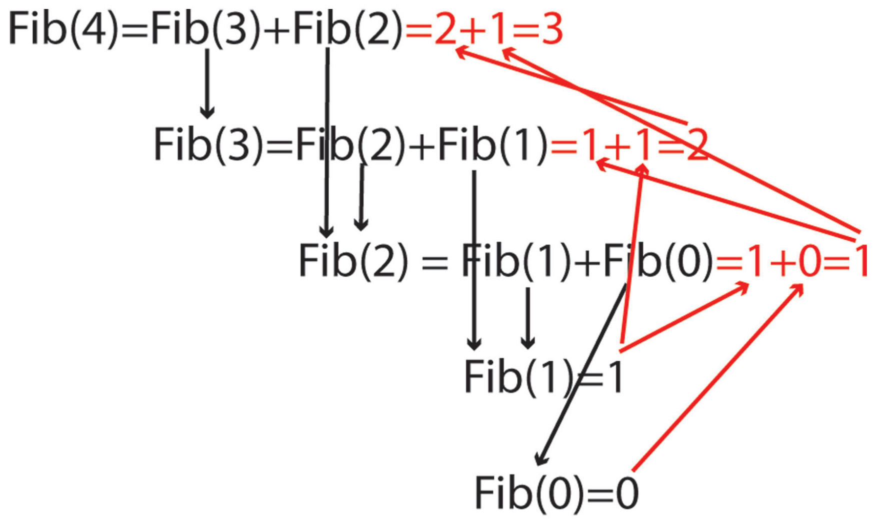
```
def Fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
```

We will look later at whether we actually need both of the base cases. Our English description is easy to translate into the recursive case of our function. The description says "every subsequent number is the sum of the two previous numbers." If n is the index of the number we are currently calculating, the indexes of the two previous numbers are n−1 and n−2. In other words, our description says Fib(n)=Fib(n−1)+Fib(n−2). We translate this into code by making the right hand side the value that we return:

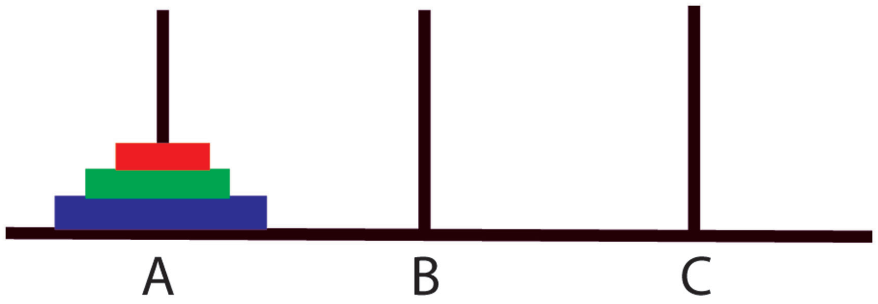
```
def Fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return Fib(n-1)+Fib(n-2}
```

It is easy to see that we need both of our base cases. The calculation of Fib(2 uses both of them: Fib(2)=Fib(1)+Fib(0). If we omitted one of them as a base case and tried to calculate it recursively we would get into trouble. For example, if we tried to calculate Fib(1) recursively we would get Fib(1)=Fib(0)+Fib(−1)=0+Fib(−2)+Fib(−3)=... and this clearly will never terminate. Are our two base cases enough? Yes; the calculation works down through even numbers and odd numbers and our base cases of 0 and 1 terminate both the evens and the odds.

Here is a sample calculation with our function


Fib(4)=Fib(3)+Fib(2)=2+1=3
Fib(3)=Fib(2)+Fib(1)=1+1=2
Fib(2) = Fib(1)+Fib(0)=1+0=1
Fib(1)=1
Fib(0)=0

Example 3: *Write a recursive function to solve the Towers of Hanoi puzzle* *Towers of Hanoi* is a puzzle game invented in1883 by the French mathematician Edouard Lucas. In this game there are three "towers" or vertical sticks that can hold a set of concentric disks. At the start of the game the disks are all on one tower, in order of decreasing size with the largest disk at the bottom and the smallest at the top:



The goal of the game is to move all of the disks from their starting tower to another tower, using the following three rules:

1. Only one disk can be moved at a time.

2. Only the top disk on a tower can be moved.

3. A disk can only be placed on an empty tower or on a larger disk, never on a smaller disk.

Lucas's game included a story, supposedly taken from the writings of an eminent Chinese monk who lived in Hanoi, that there is a temple in India where the Brahmin monks are working on a version of this puzzle with 64 golden disks. A prophecy claims that when the monks finish solving the puzzle the world will come to an end.

Just for fun, here is the cover of the box containing Lucas's first version of the puzzle:

Now we need to find a solution. As usual we'll start thinking about base cases. An easy setup to solve is when we have only one disk on the starting tower; we can move that to either of the other towers in one step. However, an even easier setup is when we have no disks at all; we can solve that without doing anything! Both of these cases refer to the number of disks on the starting tower. If you think about it, the only difference in two distinct starting states is the number of disks they contain. So we'll include the number of disks as a parameter on our Hanoi() function. Since our case with 0 disks requires no work, we can start the function as follows

```
def Hanoi(n):
    #Solves Towers of Hanoi with n disks on the starting tower
    if n>0:
```

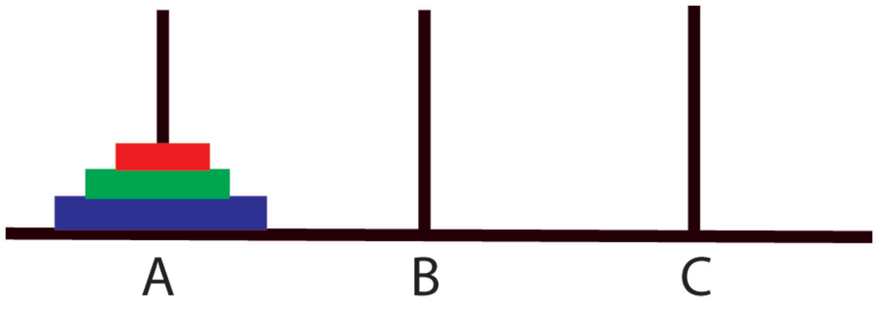
This, however, is not enough. We need to know not just how many disks are on the starting tower, but the situations with the other towers as well. Once we are into the recursion we need to move disks not just from the starting tower, but from all three towers. So we will include all three towers as parameters:

```
def Hanoi(n, source, destination, free):
    # Moves n disks from the "source" tower to the
    # "destination" tower, making use of the "free" tower.
    if n>0:
```

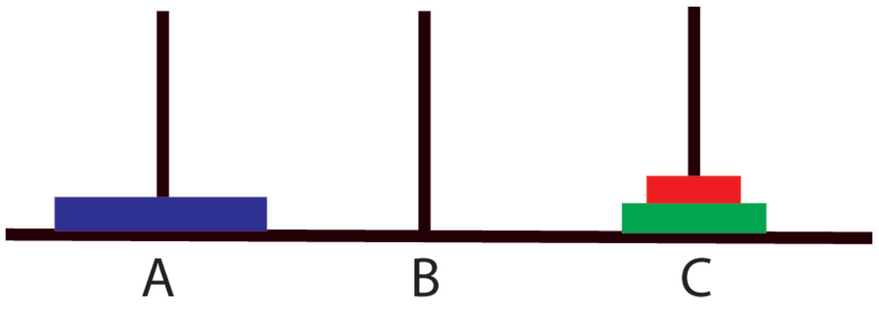
Now we are rolling. We need to express how to move `n` disks from the source to the destination in terms of moving small numbers of disks from one tower to another. This is easy:

1. Move `n−1` disks from the source tower to the free tower.
2. Move 1 disk, the largest, from the source tower to the destination.
3. Move `n−1` disks from the free tower to the destination.

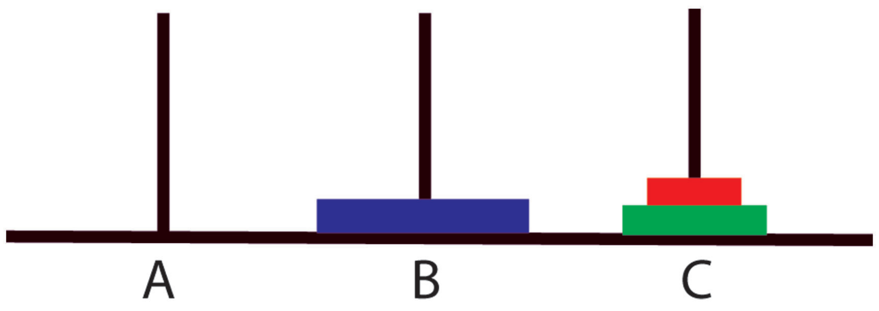
For example, suppose we start with 3 disks on tower A, the other towers empty:



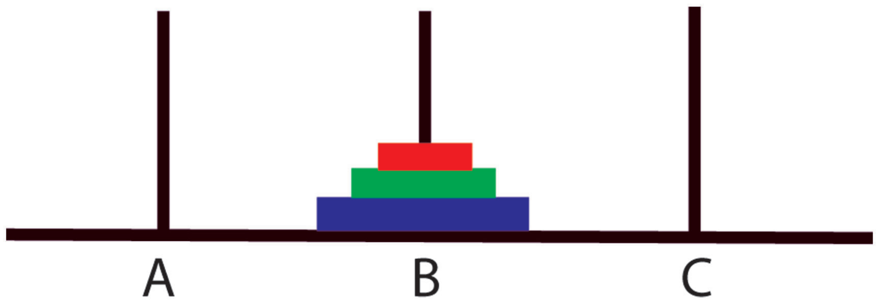
We will think of A as our source tower, B as our destination and C as free. We first recursively move 2 disks from A to C:



Next we move the largest disk from A to B:



Finally, we move those 2 disks that we placed onto the "free" tower C to the destination tower:



To turn this into code, we just need to realize that the only actual disk moves are occurring on step (2), when we move one disk from the source tower to the destination tower. Everything else is just organization for the recursive calls. We will handle step (2) with a print statement. Here is the complete code for our solution, along with a typical main() function that solves the puzzle by moving 3 disks from tower "A" to tower "B":

```
def Hanoi(n, source, destination, free):
    # Moves n disks from the "source" tower to the
    # "destination" tower, making use of the "free" tower.
    if n > 0:
        Hanoi(n-1, source, free, destination)
        print( "%s ---> %s"%(source, destination))
        Hanoi(n-1, free, destination, source)

def main():
    Hanoi(3, "A", "B", "C")

main()
```

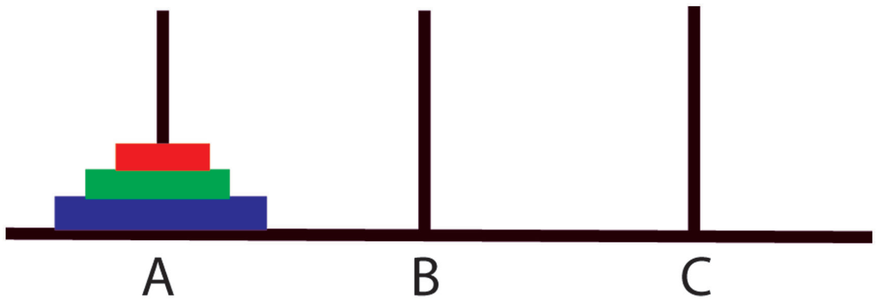
Here is a trace of this program. Because the Hanoi() function doesn't return anything, we need to use a slightly different style of trace. This time we indent to show levels of recursion. To make it a little easier to read we'll leave the quotes off the strings. At the top level Hanoi(3,A,B,C) is performed with 3 calls:

```
Hanoi(2,A,C,B)
print A ---> B
Hanoi(2,C,B,A)
```

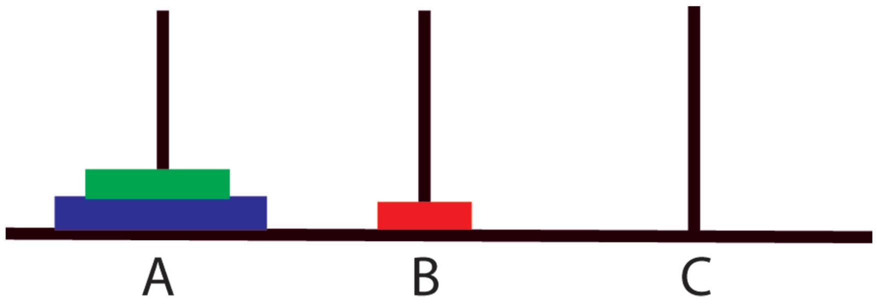
We write those three at the same indentation level, then go in a further level for the recursive calls for Hanoi(2,A,C,B) and Hanoi(2,C,B,A). Here is the resulting trace:

```
Hanoi(3,A, B, C)
        Hanoi(2,A,C,B)
                Hanoi(1,A,B,C)
                        print A--->B
                print A--->C
                Hanoi(1,B,C,A)
                        print B--->C
        print A--->B
        Hanoi(2,C,B,A)
                Hanoi(1,C,A,B)
                        print C--->
                print C--->B
                Hanoi(1,A,B,C)
                        print A--->B
```

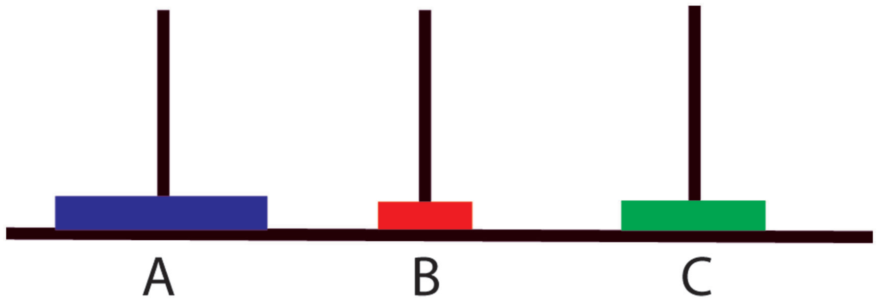
We will walk through the 7 steps. We start with



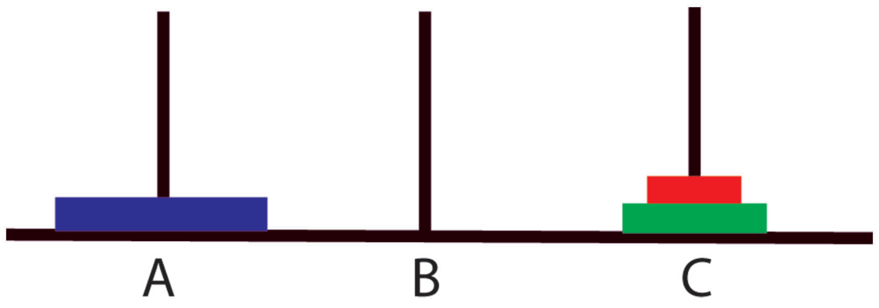
The first move is A—>B:



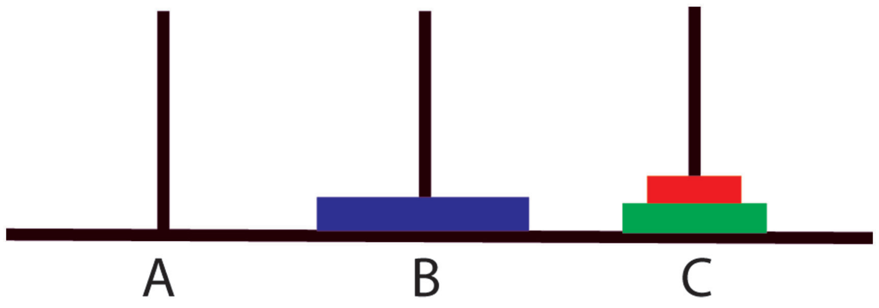
Then A—>C:



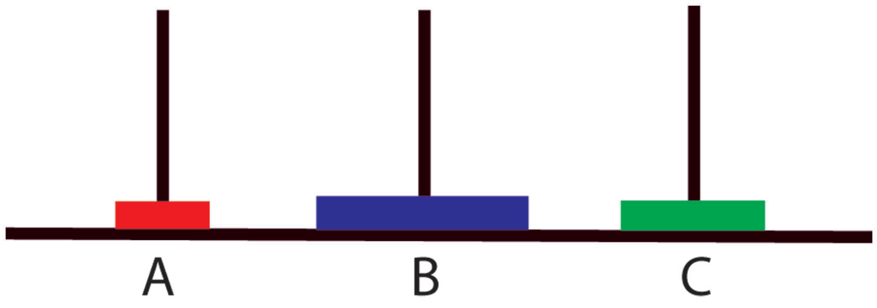
B—>C:



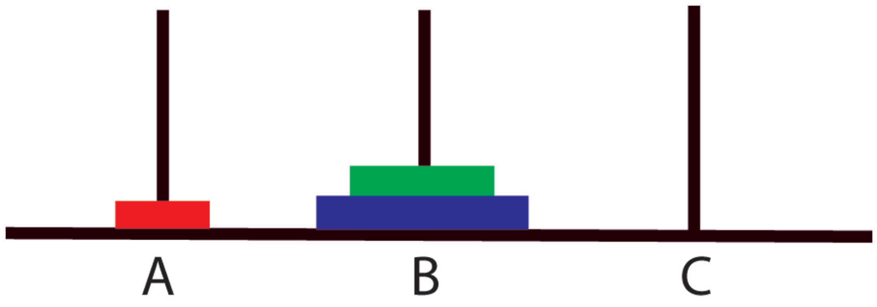
A–>B:



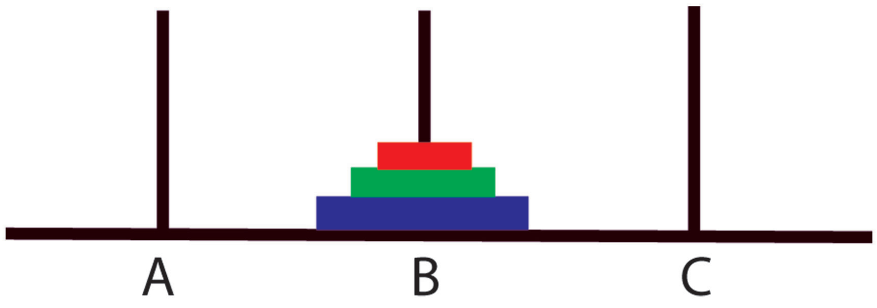
C—>A:



C—>B:



And finally A–>B:



It is not hard to show that solving the puzzle with `n` disks takes $2^n - 1$ moves. Those 64 disks the monks are supposedly moving need about 10^{19} individual moves. There are about $3 * 10^7$ seconds in a year, so if the monks were moving one disk per second it would take roughly $3 * 10^{11}$ years to solve the puzzle. Sadly, astrophysicists currently estimate that the sun will burn out in on $5 * 10^9$ years, so the monks really need to speed up.

4.6 Exercises

Write a complete Python program to solve each of the following problems.

4.1. Write a program with a function **next**(n) that returns the number after n (i.e, it returns n+1). Give your program the following main() function:

```
def main():
    print(next(35))
    print(next( next(23) ))
```

Your program should print the value 36, then 25.

4.2. A *perfect* number is one whose factors (including 1 but not the number) sum to the number itself. For example, the factors of 6 are 1, 2, and 3, which sum to 6. The factors of 28 are 1, 2, 4, 7, which sum to 28. The factors of 24, however, are 1, 2, 3, 4, 6, 8, and 12; these sum to 36, so 24 is not perfect. Write a function isPerfect (x) that returns True if x is a perfect number, then incorporate this in a program that finds all of the perfect numbers less than 10,000.

4.3. Write a function printTime(minutes) that inputs a number of minutes and prints this in terms of hours and minutes. For example, printTime(325) should print "6 hours and 25 minutes."

4.4. Write a program with area functions for squares, circles and rectangles. You might call these functions AreaSquare(side), AreaCircle(radius) and AreaRectangle(length, width). If you call these functions within the main() function:

```
def main():
    print( AreaSquare(4) )
    print( AreaCircle(10) )
    print( AreaRectangle(6, 7) )
```

the program should print 16, then 314.16, then 42

4.5. Write a function decade(year) that takes as argument a year and returns the start of the decade containing this year. For example, for argument 1968 the function should return 1960; for 1999 it should return 1900, and so forth. There are several ways to do this: you can either use the % operator (1968 % 10 is 8, so 1968 − (1968%10) is 1960) or you can have a loop that counts down by 10's from some upper limit (say 3000), and stops when it gets a decade smaller than or equal to the year. Alternatively, you could count down from the year you start, looking for a year that is divisible by 10.

Once you have function decade(year) written you can add the following main() function:

```
def main():
    y = 1
    while y != 0:
        y = eval(input( "Enter a year, 0 to exit: " ))
        if y != 0:
            print("%d was part of the %d's" %(y,decade(y)))
```

On an input like 1968 the program will print `"1968 was part of the 1960's."` Of course, for the 1960's it should add `"Now that was a groovy decade."`

4.6. Write a function deleteB(string) that removes all the instances of the letter `"b"` from a string and returns the result. Here are two hints. First, you can use a **for**-loop to run through the letters of a string:

```
for letter in "blah":
    print( letter )
```

will print first `"b"`, then `"l"`, then `"a"`, then `"h"`. Second, you can use concatenation to build up strings. Start variable result as the empty string:

```
result = ""
```

Then use the +-operator to add letters onto result. For example, you can copy a string with

```
result = ""
for letter in "blah":
    result = result + letter
```

Finally, give your program a main() function to test this out:

```
def main():
    print( deleteB("bob is a blob") )
```

will print `"o is a lo"`, a far nicer string.

Chapter 5

Testing and Debugging

You've written it so it must work, right? By now you know that is not necessarily true. We all make mistakes. To be a successful programmer you need to be able to reliably find and fix your mistakes. This chapter deals with two related efforts: writing test cases to determine if your program is running correctly, and finding the source of the problem when testing reveals that something is wrong.

Python has two built-in systems to help you with debugging. You should not need either of these for short programs, but they come in handy as your programs become longer and more complex. Knowing how to use a debugger is a good skill for any programmer. Just don't fall into the trap of relying on a debugger instead of writing your code carefully.

5.1 Testing Your Code

Probably the most common mistake made by beginning programmers is insufficiently testing their code. Most of us need to believe what we are doing is correct or we'll never do anything. It is hard to surrender that certainty when it comes time for testing. Too many beginners are satisfied with one or two test cases; sometimes students even hand in code without ever running it and are surprised to hear that it doesn't work correctly. If you want to be a successful programmer, integrate a simple testing methodology into your coding practice. Far from slowing you down; you'll find that this helps you to develop good code faster than you were previously writing buggy code.

When To Test

There are three issues with testing: when to test, which test cases to use, and what to do with the information if a test comes back with a different response than you expect. The *when* question is easy: test continually as you develop. Beginners tend to write an entire program and then test it. If you do that, when a test fails every line of the program is a potential source of the problem. If you test as you develop and are confident that the program is correct before you add a new function, if errors appear when you test after that addition you know the likely source of those errors.

For example, think back to Program 4.3.5, which asks the user to enter strings and says whether each string is a palindrome. To test this program we need a more precise statement of what it should do. Here is such a statement:

Write a program that repeatedly asks the user to input a string; the input loop is terminated by an empty string. For each non-empty string the program should strip off any punctuation characters and determine if what is left forms a palindrome. The program's response to a non-empty input $< s >$ should be "$< s >$ is a palindrome." or "$< s >$ is not a palindrome."

What follows is an outline of the program we wrote to solve this problem.

```
# This program reads strings from the user and
# says if they are palindromes: the same when
# read backwards as when read forwards

def StripPunctuation(s):
    # This returns a string just like s only all
    # of the non-letters are removed and the letters
    # are all changed to lower-case.

def Reverse(s):
    # This returns the reversal of string s:
    # if s is 'abc' this returns 'cba'.

def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    # It calls StripPunctuation to remove all of the
    # non-letters from s

def main():
    # This has the input loop.  It reads strings from
    # the user, tests if they are palindromes, and
    # prints the answer. The loop terminates when it
    # gets an empty string.

main()
```

The skeleton of Program 4.3.5

When we developed this program in section 4.3 we first wrote main() with a *stub*, or dummy function, for IsPalindrome(), then we wrote both IsPalindrome() and Reverse(), then finally StripPunctuation(). We would test the program in exactly these pieces: one set of tests to ensure that our main() loop is working correctly, one set of tests after completing IsPalindrome() and Reverse(), to ensure that the main functionality of our program is correct, and a final set of tests on the completed program.

Test Cases

Picking out test cases for a new component of your program involves thinking about what it does. There are three broad categories of test cases:

- Typical cases

- Boundary cases
- Extreme cases

Typical cases are test cases that exercise the basic functionality of your code. You should use enough to cover all of the possibilities several times. For example, if you are testing a function that determines if a number is prime, you should test with several numbers that are prime and several numbers that are not. If you are testing a function that decomposes a string into a list of English words (such as turning ”thisisatest” into [”this”, ”is”, ”a”, ”test”], you might use strings that are single English words, such as ”bob”, strings with two words, such as ”testtwo”, strings with a larger number of words, such as ”onetwothreefourfive”, and strings that can be decomposed in multiple ways, such as ”onestone” (which could be [”one”, ”stone”] or [”ones”, ”tone”]). You should also include negative test cases: strings that don’t contain any word, such as ”pxq” and strings that contain some words but can’t be completely decomposed into words, such as ”testfailswxq”.

Boundary cases apply in situations where there is a range of possible inputs. Bugs frequently hide at the edge of the range of possibilities. For example, with a function that tests for prime numbers, the bottom end of the range would be in the numbers 1 and 2; 1 is generally not considered to be prime while 2 is definitely prime. You should test both. For the program that decomposes strings, you should test both the empty string and strings of one letter. If the unit you are testing is based on a loop, make sure it is both starting and stopping at the right point. A common error results when a *for*-loop goes one step too far, or stops one step earlier than it should. Try to find test cases that check for this.

Extreme cases test input you might not ordinary think about. For example, with the prime number tester, what will it respond if you give it a negative number? There is no *a priori* right answer, but you should code your programs to handle smoothly unexpected input. Programs should not crash, regardless of the input they are given. For the prime number function you should include 0 and a negative number as test cases. For the string decomposition function you should test the empty string. You should also test very large inputs to ensure that you haven’t unintentionally placed a limit to the size of the input your program can handle. There is no upper end to the range of prime numbers, but you might test a very large number to ensure that you haven’t accidentally put a limit on the size of the numbers your function can handle: 15458863 is the millionth prime number. As with numbers there is no upper boundary on the size of strings but you should test a very large input, such as ”IreadthenewstodayohboyAboutaluckymanwhomadethegradeAndthoughthenews-wasrathersadWellIjusthadtolaughIsawthephotographHeblewhismindoutinacarHe-didn’tnoticethatthelightshadchangedAcrowdofpeoplestoodandstaredThey’dseen-hisfacebeforeNobodywasreallysureIfhewasfromtheHouseofLords” Depending on the algorithms you are using, large inputs sometimes are processed very, very slowly; when testing you need to take this into account and use large, but still practical, test cases.

Lets' consider the palindrome program we wrote in section 4.3 and outlined above. We test this in three stages.

Phase one We test just the main() function, with a stub for the IsPalindrome() function. Here we just want to be sure the main loop is working.

Typical cases "test", "this is a test"

Boundary cases the empty string

Extreme cases "fox socks box knox knox in box fox in socks knox on fox in socks in box"

Phase two We test the Reverse() and IsPalindrome() functions. The program should now recognize palindromes. We haven't yet written the StripPunctuation() function, so we will use test strings that have only alphabetic letters.

Typical cases "madam", "maddam", "maday"

Boundary cases the empty string, "a"

Extreme cases "amanaplanacanalpanama", "abcdefghijklmnopqrstuvwxyzyxwvut-srqponmlkjihgfedcba"

Phase three Now we test the whole program, with special emphasis on the StripPunctuation() function:

Typical cases "A man, a plan, a canal: Panama!", "Drat such custard", "abc;a"

Boundary cases the empty string, "a"

Extreme cases "A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal-Panama!"

How to test

One simple way to manage testing, which works in any programming language, is to develop your program in such a what that it can always be run. That is how we developed program **??**. We even implemented a stub for the IsPalindrome() function so that main() would run when we hadn't written the rest of the code. This leads to a program design strategy known as *Top-Down Design* – at each step we implement one part of the program (typically one function), breaking that part down into steps that are each represented by functions. We give stubs for the new functions. This continues until the steps that we need to implement are so simple that we can code them without calling any new function. Some languages, though not Python, require function definitions to be present in the code before the calls to those functions. If you do this even though Python doesn't require it (in larger programs it does help you to find function definitions), Top-Down Design results in you writing the code file backwards – from the end to the start. As long as you use a good program editor, such as Idle, that doesn't present any difficulties. If you use this strategy in each phase of testing you only need to run the program and type in the test cases.

An alternative for testing Python code makes use of Python's interactivity. When we run a program in Idle, the entire program is loaded into memory. When the program stops running, it stay's in Idle's memory. Individual functions can still be called as long as you give them appropriate arguments. For example, if you run program **??** and immediately enter the empty string, at the prompts you can directly call the Reverse(), IsPalindrome(), and StripPunctuation() functions. For example, here is a typical interaction:

When I run the program it types the program's prompt for input:

```
Enter a string:
```

I press the Return key to enter the empty string and the program halts. Idle then gives me a prompt:

```
>>>
```

I want to see the result of reversing a string, so at the prompt I type

```
>>> Reverse("abc")
```

Idle responds by printing the result of calling Reverse("abc") and then another prompt:

```
'cba'
>>>
```

I next check out IsPalindrome() by typing at the prompt

```
>>> IsPalindrome( "bob" )
```

to which Idle replies

```
True
>>>
```

Finally, I try calling StripPunctuation():

```
>>> StripPunctuation( "Naomi did I moan?" )
```

and Idle responds

```
'naomididimoan'
>>>
```

In this way we can quickly and easily test a variety of functions.

Debugging – making use of testing results

No amount of testing can show that code is bug-free. Testing only tells us about the presence of errors, not their absence. Most students don't want to see it this way, but a successful test is one that reveals a problem. So celebrate when a test comes back with the wrong answer: you found a bug! But now, what do you do? Fortunately, 99% of all bugs can be fixed by just carefully reading the code. If you have followed our methodology for developing programs the test that failed should involve only a small amount of new code. All of the prior code should have been thoroughly tested in earlier testing phases, so the potential source of the bug should be easy to localize. First, take a minute to find the simplest example you can of input that fails. If the problem is a function like StripPunctuation(), it is a lot easier to to work with a string of 2 or 3 characters than one of 15. Now it is just a matter of thinking carefully about your code. Check your logic; does the algorithm you are using really do what you think? Try stepping through the guilty function on paper, working through all of the steps of the example that failed. If done carefully, this almost always finds the problem. In tricky situations it helps if you can find two examples, only one of which runs correctly. If you can find why one succeeds and one fails you will probably find the bug. Don't give in to the temptation of monkey-coding – a cycle of random changes to the code followed by quick tests. This usually introduces more new bugs than it fixes. Stay in control of your code and don't make a change until you are confident it is correct. Any program you are asked to write in an introductory course will be short enough that you can keep it all in your head at one time. It is much faster to reason your way through the program than to complete it through arbitrary modifications.

The next two sections discuss *debuggers*, which are tools designed to help with the debugging process. These can be very helpful for longer programs. You probably won't need them for a first programming course, but some people find them helpful. Use them if you wish, but don't rely on them as a substitute for careful reading.

5.2 The Shell Debugger

A *debugger* is a software tool that helps programmers find bugs in complex programs. This and the next section look at two simple debuggers that are standard parts of Python. A word of caution before we start: you should not let yourself become too dependent on a debugger. When you are starting out as a programmer, it is much more important to learn to read your code carefully than to become proficient with debugging tools. Good programmers write code that is easy to read; once you know how the various statements of Python work, errors should jump out at you. There is no program in all of these notes for which a debugger is a necessary part of the program development. Still debuggers are handy tools that can come to your rescue in those rare times when you just don't see what the problem is with your code. So use a debugger sparingly, but don't hesitate to use it when it will help.

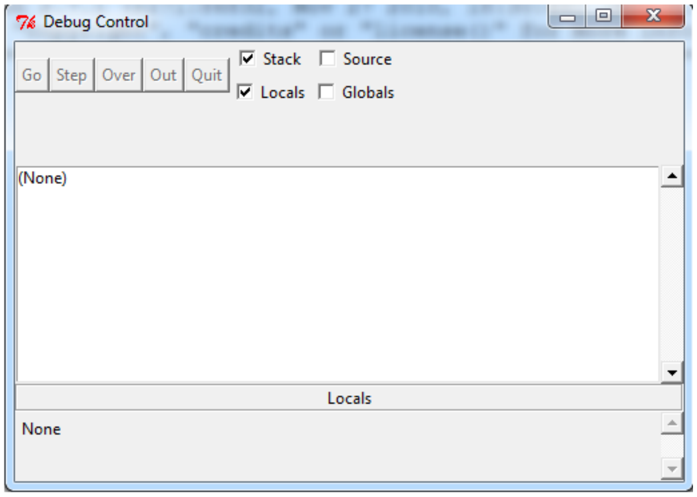
There are two standard debuggers for the Python. One of these is built into IDLE and the Python shell. The other is a module called "Pdb". The shell debugger is simpler to use and has fewer features, so we will look at it first. The next section discusses Pdb.

Suppose you are working with the following program:

```
def IsPrime(x):
    for d in range(2, x):
        if x%d == 0:
            return False
    return True

def PrintPrimes(N):
    for num in range(2,N+1):
        if IsPrime(num):
            print( num )

def main():
    PrintPrimes(100)

main()
```

Program 5.2.1: Finding Prime Numbers

To use the shell debugger your program must be loaded into the system's memory. The easiest way to achieve this is to run the program from IDLE: just select Run Module from the Run menu. You need to run the program through to completion, so if your program expects input from the user, give it input until the program stops executing. Then in the Debug menu of the shell select the Debugger option. A *Debug Control* window will pop up:



```
(None)
```

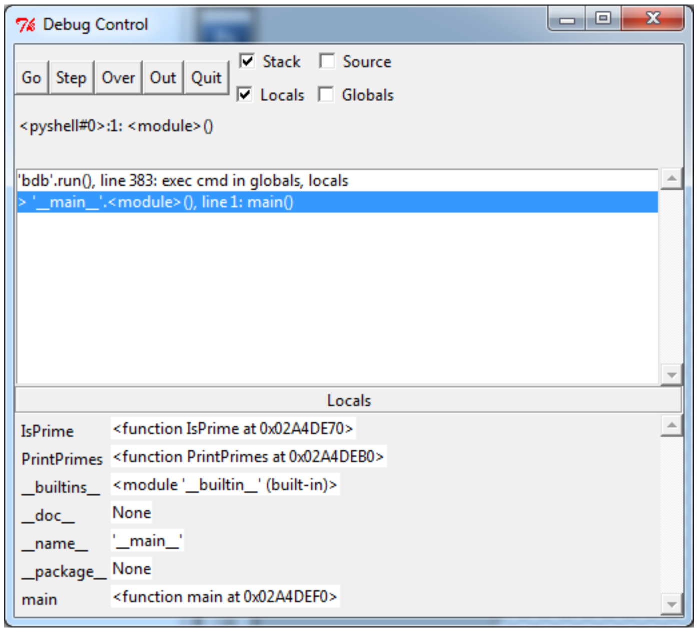
Locals

```
None
```

To start debugging you need to restart your program. The program starts wtih a call to main() so at the debugger prompt in the shell we type

```
>>> main()
```

The Debug Controls window changes to include some information about the program.



```
<pyshell#0>:1: <module>()

'bdb'.run(), line 383: exec cmd in globals, locals
> '__main__'.<module>(), line 1: main()
```

Locals

```
IsPrime       <function IsPrime at 0x02A4DE70>
PrintPrimes   <function PrintPrimes at 0x02A4DEB0>
__builtins__  <module '__builtin__' (built-in)>
__doc__       None
__name__      '__main__'
__package__   None
main          <function main at 0x02A4DEF0>
```

There are five buttons at the top of this window that control the actions of the debugger. They are

Go This causes execution of the program to resume and run until the program finishes.

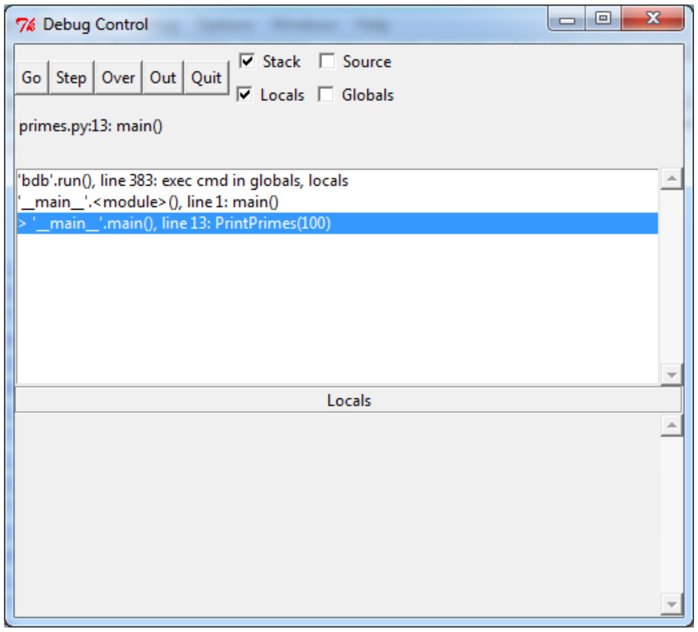
Step This causes the next instruction in the code to be executed. If that instruction is a function call, control changes to the first line of that function.

Over This is similar to Step, only function calls are evaluated rather than stepped into.

Out This causes the rest of the current function to be executed. The program halts as soon as it returns from the current function.

Quit This stops the execution of the program and halts the debugger.

We will now go through a long series of steps examining our prime-generating program in the debugger. When we Step into main() we see:

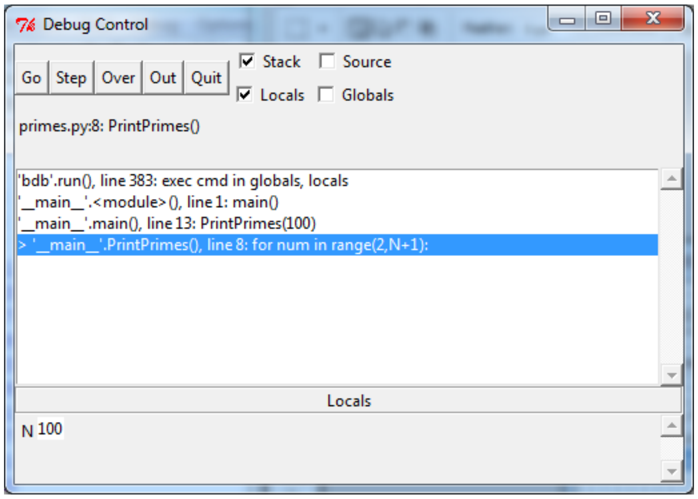


Stack [✓], Source, Locals [✓], Globals

```
primes.py:13: main()
```

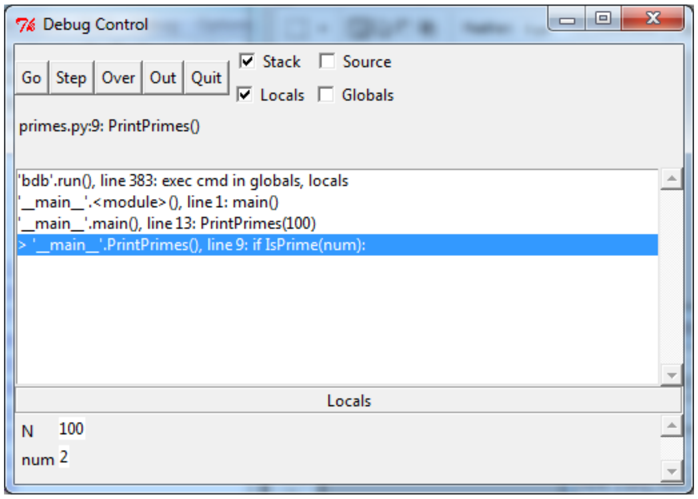
```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
> '__main__'.main(), line 13: PrintPrimes(100)
```

We now Step into PrintPrimes with argument 100:



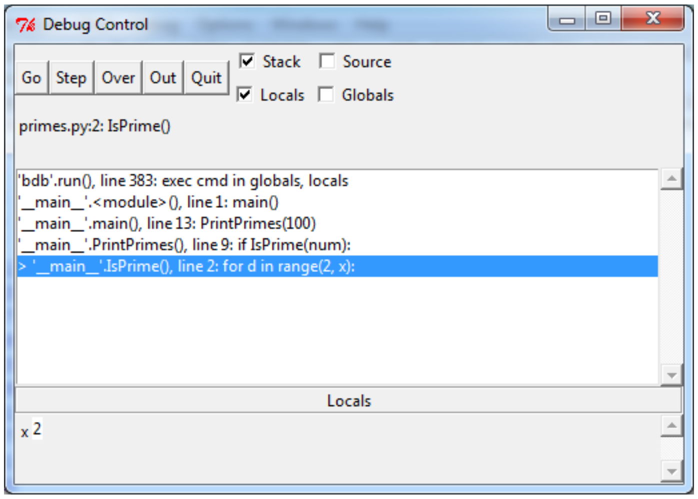
```
primes.py:8: PrintPrimes()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 8: for num in range(2,N+1):

Locals
N 100
```

You can see the value of N at the bottom of the screen. The first line of PrintPrimes is a **for**-loop. Initially the loop variable num has no value. After we Step again:



```
primes.py:9: PrintPrimes()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 9: if IsPrime(num):

Locals
N    100
num  2
```

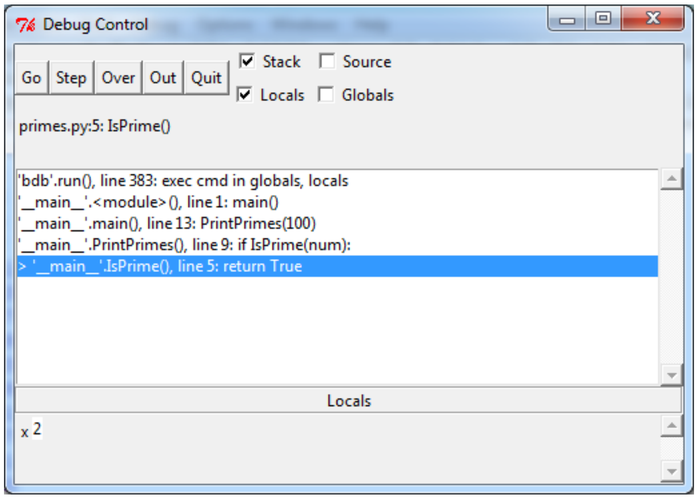
we see that num has the value 2 and we call IsPrime(num). The next Step takes us inside funciton IsPrime



[✓ Stack] Source
[✓ Locals] Globals

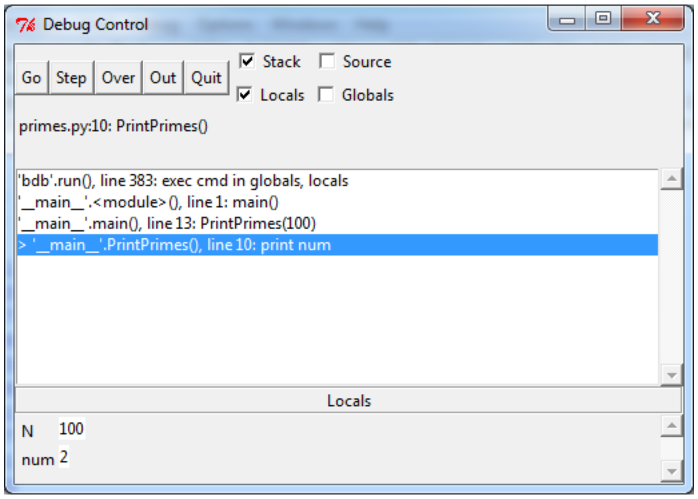
```
primes.py:2: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 2: for d in range(2, x):

x 2
```

The **for**-loop is vacuous since the **range**(2,2) is the empty list, so IsPrime returns True and we go back to function PrintPrimes:



[✓ Stack] Source
[✓ Locals] Globals

```
primes.py:5: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 5: return True

x 2
```



☑ Stack ☐ Source
☑ Locals ☐ Globals

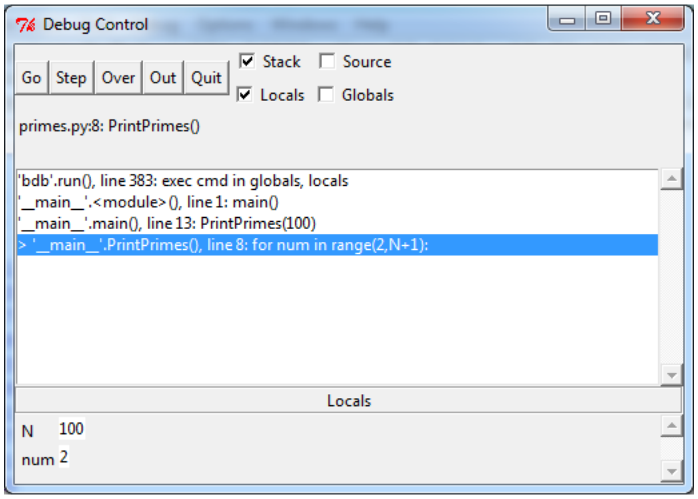
primes.py:10: PrintPrimes()

```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 10: print num
```

Locals

```
N    100
num  2
```

When we execute the **print** instruction the value of `num`, 2, is printed in the shell and we go to the next instruction:



☑ Stack ☐ Source
☑ Locals ☐ Globals

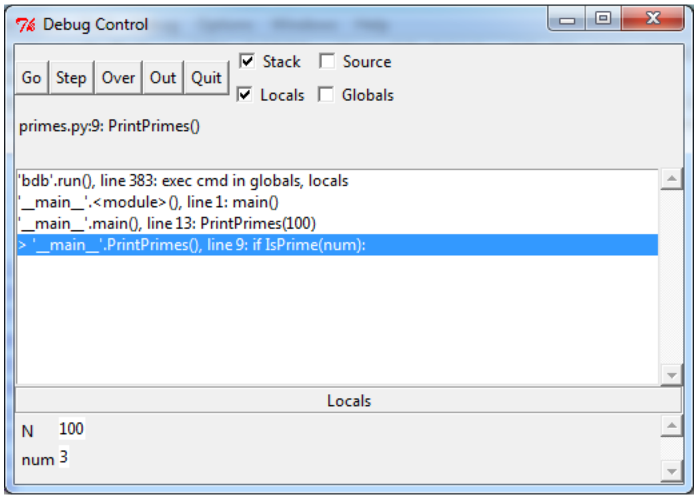
primes.py:8: PrintPrimes()

```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 8: for num in range(2,N+1):
```

Locals

```
N    100
num  2
```

The next value of `num` is 3:



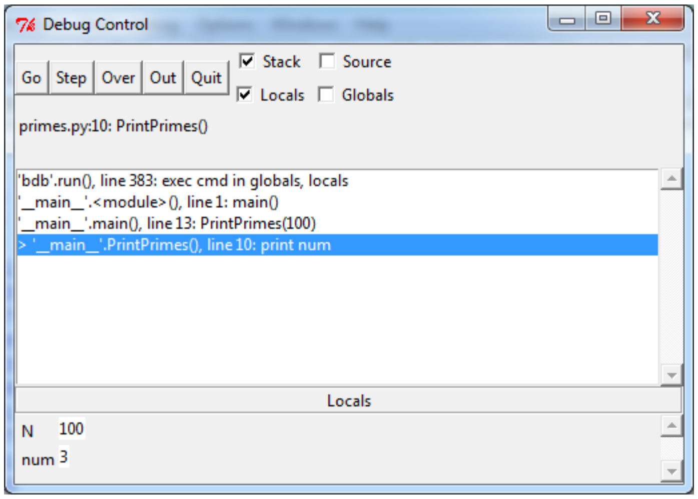
primes.py:9: PrintPrimes()

```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 9: if IsPrime(num):
```

Locals

```
N   100
num 3
```

This time we will step Over the call to IsPrime(num). This returns True since 3 is prime and we come to:



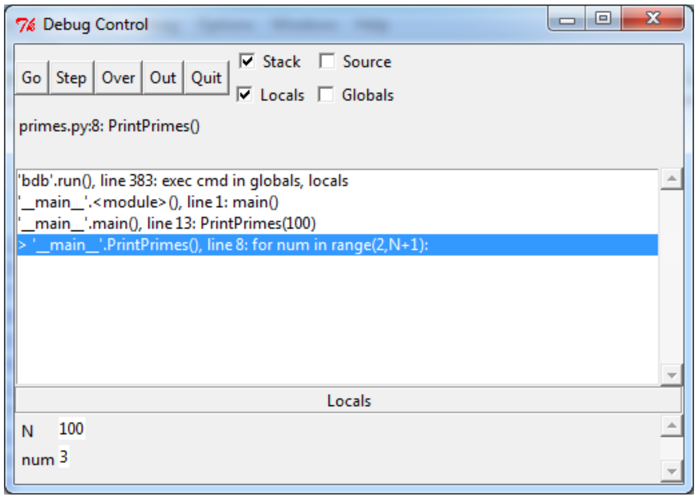
primes.py:10: PrintPrimes()

```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 10: print num
```

Locals

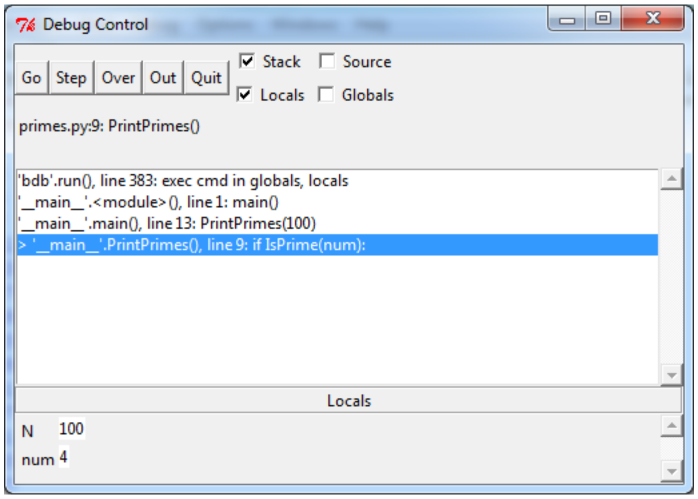
```
N   100
num 3
```

When we Step again num is printed, and we are back to our main **for**-loop:

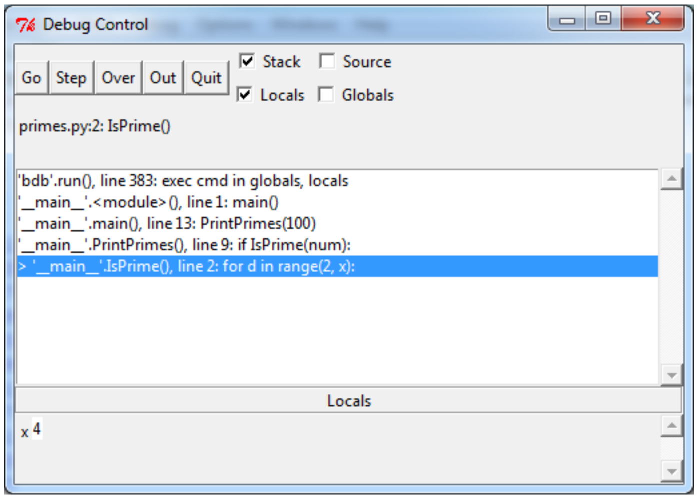


```
primes.py:8: PrintPrimes()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 8: for num in range(2,N+1):

N   100
num 3
```

We now have a sequence of Steps with the value of num at 4:



```
primes.py:9: PrintPrimes()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 9: if IsPrime(num):

N   100
num 4
```

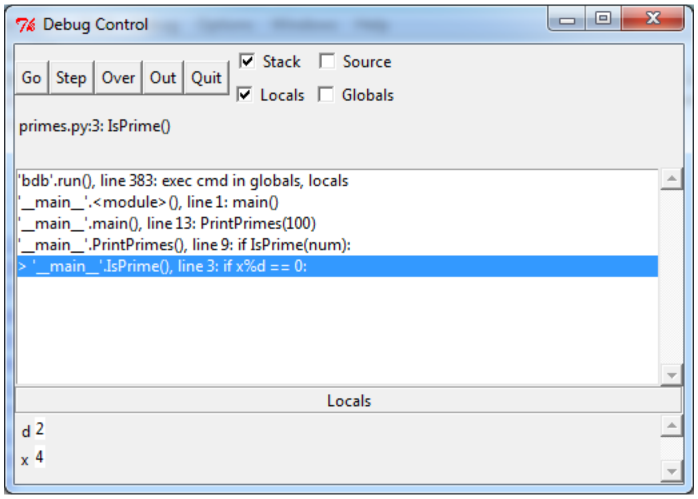


primes.py:2: IsPrime()

```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 2: for d in range(2, x):
```

Locals

```
x 4
```



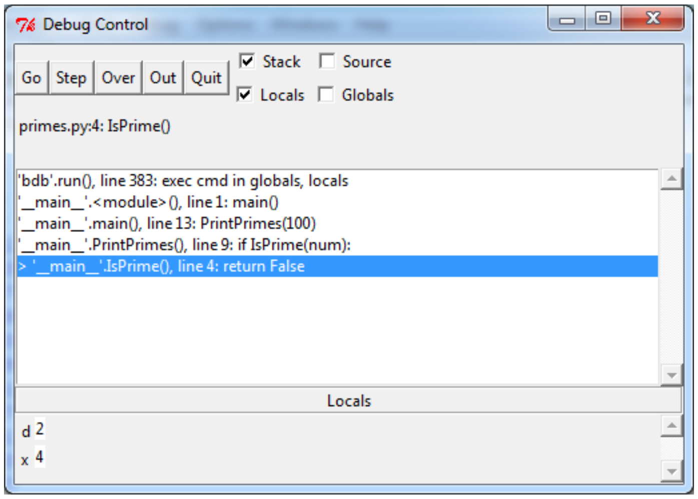
primes.py:3: IsPrime()

```
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 3: if x%d == 0:
```

Locals

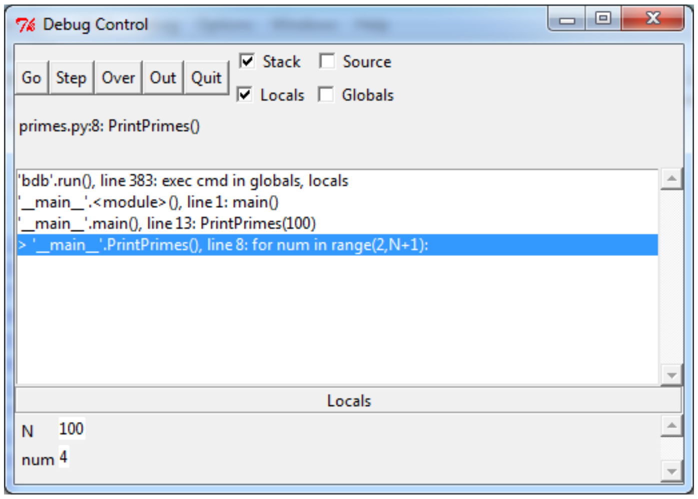
```
d 2
x 4
```

When `x` is 4 and `d` is 2, `x%d==0` is `True`, so the function returns `False`



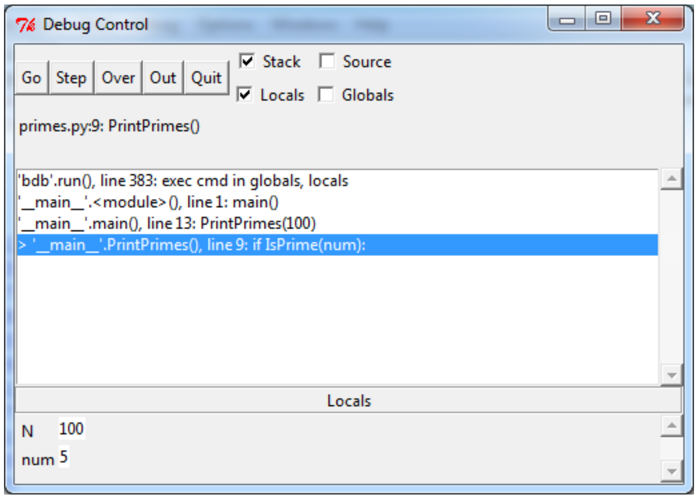
```
primes.py:4: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 4: return False

Locals
d 2
x 4
```

and we are back to the main **for**-loop of function PrintPrimes:

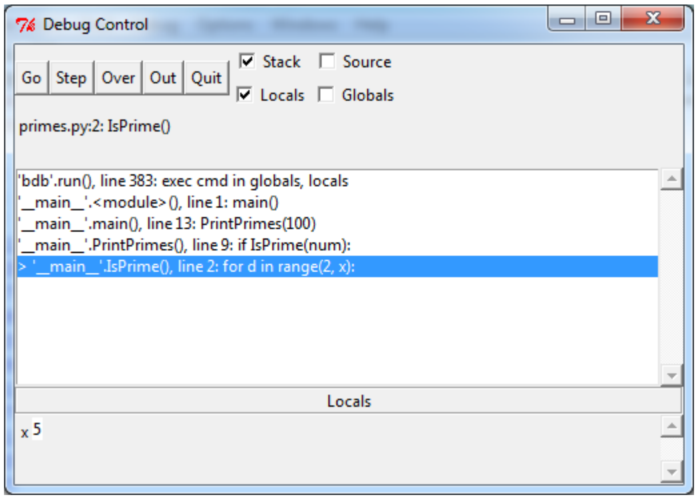


```
primes.py:8: PrintPrimes()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 8: for num in range(2,N+1):

Locals
N   100
num 4
```

Now we have another sequence of steps to determine if 5 is prime:

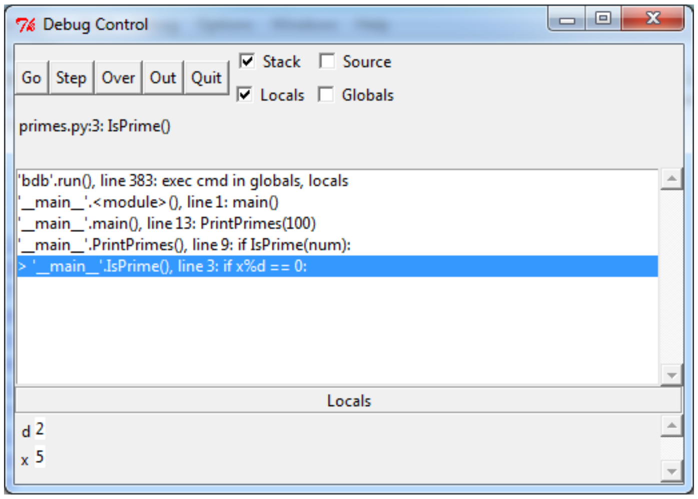


```
primes.py:9: PrintPrimes()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 9: if IsPrime(num):

Locals
N    100
num 5
```

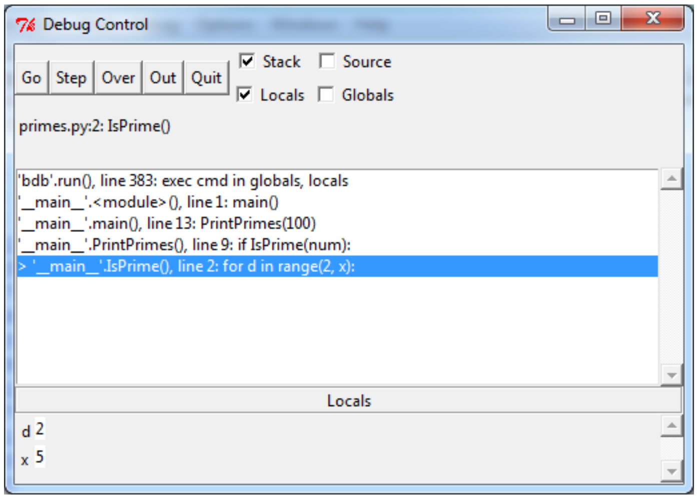


```
primes.py:2: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 2: for d in range(2, x):

Locals
x 5
```

First we try to divide 2 into 5:

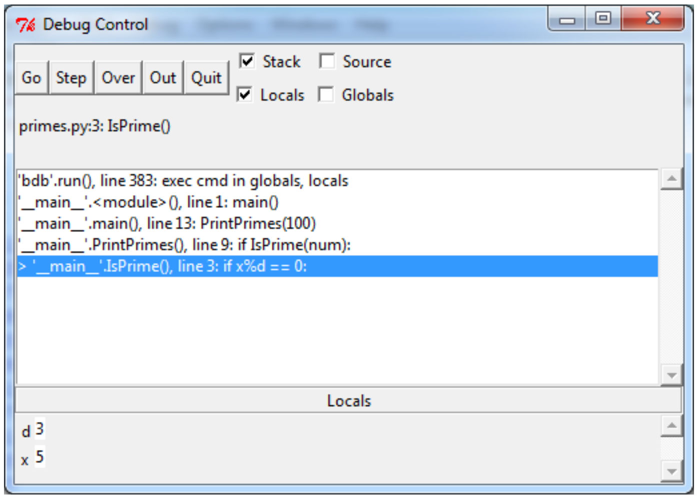


```
primes.py:3: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 3: if x%d == 0:

d 2
x 5
```



```
primes.py:2: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 2: for d in range(2, x):

d 2
x 5
```

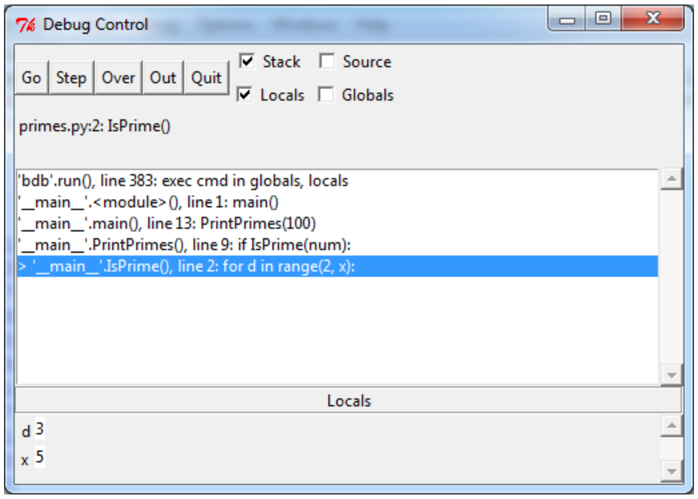
Then we try 3:



☑ Stack ☐ Source
☑ Locals ☐ Globals

```
primes.py:3: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 3: if x%d == 0:
```

d	3
x	5

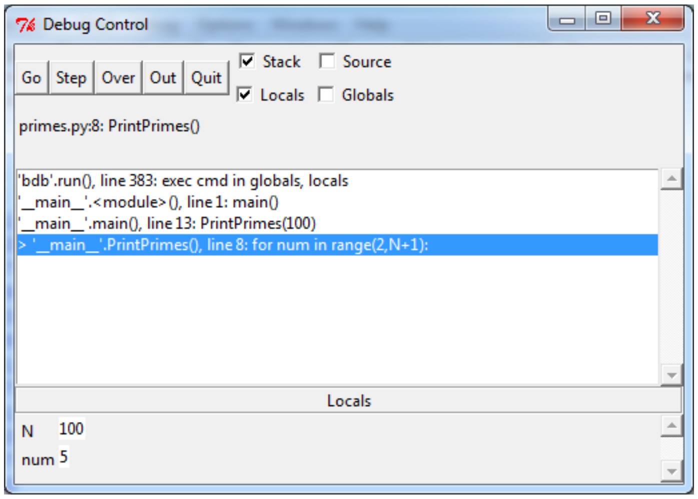


☑ Stack ☐ Source
☑ Locals ☐ Globals

```
primes.py:2: IsPrime()

'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 2: for d in range(2, x):
```

d	3
x	5

We could continue in this way, but we know that 5 is prime and all of these divisions will fail. Let's step Out of the call to IsPrime This function completes its execution and returns True, so we are back in function PrintPrimes:



```
primes.py:8: PrintPrimes()
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
> '__main__'.PrintPrimes(), line 8: for num in range(2,N+1):
```

Locals

```
N   100
num 5
```

At this point we have seen enough. We click on the Go button to continue the rest of the program without interruption, and the remaining prime numbers are printed to the shell. We could run the program again in the debugger by typing

```
>>> main()
```

at the shell prompt. To exit from the debugger, either uncheck Debugger in the Debug menu or click on the close-box in the upper right corner of the Debug Control window.

5.3 Pdb

The Pdb module gives a richer debugging environment than the shell debugger that we discussed in section 5.2. Again, you select Run Module from the Run menu of IDLE to load your program into the shell's memory. Then at the shell prompt you need two lines:

```
>>> import pdb
>>> pdb.run("main()")
```

Don't forget the parentheses after main; you want to call function main, not just evaluate it. The system responds with

```
> <string>(1)<module>()->None
(Pdb)
```

At this point you have access to a number of pdb commands. You type these commands into the shell at the (Pdb) prompt; the system responds in the shell. Here is a list of the most useful pdb commands; most can be abbreviated to their first letter:

s(tep) Execute the current line, stepping into function calls. This is like the Step button in the shell debugger.

n(ext) Execute the current line, stepping over function calls. This is like the Over button in the shell debugger.

r(eturn) Execute the rest of the current function call, continuing until it returns. This is like the Out button in the shell debugger.

c(ontinue) Continue to execute the rest of the program, not stopping unless a breakpoint is encountered. This is like the Go button in the shell debugger.

l(ist) List the portion of the source code around the current instruction.

list first, last List the portion of the source code between lines first and last.

p expression The expression is evaluated and its value is printed. Note that the command is *p*, not *print*. The command p (x, y,z) is the usual way to get the current values of variables x, y and z.

b(reak) function Set a *breakpoint* at the first line of the named function.

b(reak) n Set a *breakpoint* at line number n.

b(reak) function, condition Set a conditional breakpoint at the start of the named function.

b(reak) n, condition Set a conditional breakpoint at line number n.

disable n Disable breakpoint number n (not the breakpoint at line number n).

enable n Re-enable the disabled breakpoint number n

condition n Removes any condition on breakpoint n

condition n c Sets the condition on breakpoint n to c.

cl(ear) Removes all breakpoints.

cl(ear) n Removes breakpoint number n

Return-key On a Return the system re-runs the last pdb command it executed. So to step through a program you need to give one s or step command, followed by a series of <Return> keys.

The only thing really new here is the notion of a *breakpoint*. If you are executing more than one line of code with **next**, **return**, or **continue** commands, execution will always stop when a breakpoint is reached. Listings of the code will indicate breakpoints with a B. Note that the breakpoints are numbered from 1 as you create them. Some of the commands require you to know the number of a breakpoint; you can always look back in the shell to find the number that was given when you created it.

As in section 1.5, we will run the following program through the pdb debugger:

```
def IsPrime(x):
    for d in range(2, x):
        if x%d == 0:
            return False
    return True

def PrintPrimes(N):
    for num in range(2,N+1):
        if IsPrime(num):
            print( num )

def main():
    PrintPrimes(100)

main()
```

Program 5.3.1: Finding Prime Numbers

After running the program in the shell, we type

```
>>> import pdb
>>> pdb.run("main()")
```

and the system responds:

```
> <string>(1)<module>()
(Pdb)
```

You need to actually step into the program before doing anything with it. So we type

```
(Pdb) step
```

and the system responds with

```
--Call--
> c:\users\bob geitz\...\primes.py(12)main()
-> def main():
(Pdb)
```

The program has 15 lines (to see this, put the cursor on the last line of the program in IDLE; the line number appears at the bottom right of the IDLE window). To get a listingof it in the shell, so we can avoid bouncing back and forth between the shell and IDLE, we ask the first 15 lines. We type:

```
(Pdb) list 1, 15
```

and the system responds:

```
  1          def IsPrime(x):
  2              for d in range(2, x):
  3                  if x%d == 0:
  4                      return False
  5              return True
  6
  7          def PrintPrimes(N):
  8              for num in range(2,N+1):
  9                  if IsPrime(num):
 10                      print num
 11
 12  ->        def main():
 13              PrintPrimes(100)
 14
 15          main()
(Pdb)
```

Note the arrow at line 12, which indicates the current instruction, in this case the start of function main. Let's do a few more steps. We type

```
(Pdb) s
```

and the system responds

```
> c:\users\bob geitz\...\primes.py(13)main()
-> PrintPrimes(100)
(Pdb)
```

We could step again by just pressing the <Return>-key, but the output is easier to read if we at least use the s of step:

```
(Pdb) s
```

The system responds:

```
--Call--
> c:\users\bob geitz\...\primes.py(7)PrintPrimes()
-> def PrintPrimes(N):
(Pdb)
```

One more time: we type

```
(Pdb) s
```

and the system responds

```
> c:\users\bob geitz\...\primes.py(8)PrintPrimes()
-> for num in range(2,N+1):
(Pdb)
```

To see the current value of N we type

```
(Pdb) p N
```

and the system responds

```
100
```

If we try to find the value of variable num from PrintPrimes:

```
(Pdb) p num
```

we get an error message from the debugger because the **for**-loop in which num is defined has not yet started executing.

We could continue stepping through the program, but let's take advantage of the fact that pdb has breakpoints. To watch the program determine if the number 65 is prime, we can set a breakpoint just inside the **for**-loop in PrintPrimes when the value of num becomes 65. We type:

```
(Pdb) break 9, num==65
```

and the system responds:

```
Breakpoint 1 at c:\users\bob geitz\...\primes.py:9
(Pdb)
```

We can run the execution of the program to this point with

```
(Pdb) continue
```

The program runs, printing out prime numbers as it finds them and stops at the breakpoint when our condition is met:

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
> c:\users\bob geitz\...\primes.py(9)PrintPrimes()
-> if IsPrime(num):
(Pdb)
```

We can see the current values of N and num with

```
(Pdb) p (N, num)
```

and the system prints them:

```
(100, 65)
(Pdb)
```

We will now step 4 times and watch the action of the code. To the first step the system responds:

```
--Call--
> c:\users\bob geitz\...\primes.py(1)IsPrime()
-> def IsPrime(x):
(Pdb)
```

On the next step we go inside IsPrime():

```
> c:\users\bob geitz\...\primes.py(2)IsPrime()
-> for d in range(2, x):
(Pdb)
```

Our first value of d is 2, so when we step

```
> c:\users\bob geitz\...\primes.py(3)IsPrime()
-> if x%d == 0:
(Pdb)
```

the result is False and we go back and increment d.

```
> c:\users\bob geitz\...\primes.py(2)IsPrime()
-> for d in range(2, x):
(Pdb)
```

We know that when d gets to 5 it will divide evenly into x and IsPrime will return False. To see if this happens, we tell use the Return command, which tells the system to continue until the current function returns. We type:

```
(Pdb) return
```

and the system responds:

```
--Return--
> c:\users\bob geitz\...\primes.py(4)IsPrime()->False
-> return False
(Pdb)
```

The first line of this says it is executing a Return command. The next line says that the current function, IsPrime, returns False. The third line is the next instruction to execute, the actual return instruction from line 4 of the code. If we step again we are back in function PrintPrimes:

```
> c:\users\bob geitz\...\primes.py(8)PrintPrimes()
-> for num in range(2,N+1):
(Pdb)
```

We now finish the exeuction of the program with a *Continue* statement. We type:

```
(Pdb) continue
```

and the system responds with the remaining prime numbers less than 100, and stops the program execution:

```
67
71
73
79
83
89
97
>>>
```

Note that the final prompt we get from the system is the usual shell prompt, not the (Pdb) prompt.

Using Pdb we can quickly get to any portion of a long program. With some patience, stepping through instructions and examining the values of variables, you can use this tool to find some very subtle errors in your code. This only works if you know exactly what your code is supposed to be doing; you must compare actual values to your expectations for those values. Debuggers are no substitute for good programming practices, but in some cases they can very helpful in the development of long programs.

Chapter 6

Data Structures

Up to this point we have restricted our data to simple values: numbers and strings. In the real world data is far more complex than this. Consider a program that represents people. The program might need to keep track of physical characteristics, like height and hair color. Or it might need to track personal data, like name, age and social security number. Or it might need student id-numbers and transcripts (which are themselves lists of courses and grades). Most programs that deal with objects in the world store multiple items of simple data for each object. *Data structures* are program constructs that hold multiple items of related data, where each item can be of any type. Python has a rich, easy-to-use set of data structures that will enable you to write sophisticated programs with relatively little effort. In this chapter we will look at three important data structures in Python: lists, tuples and dictionaries.

6.1 Concepts

Data structures allow us to package together multiple items of simple data. For example, we might have a list of numbers: `[2, 3, 5]`, or a list with a name, age and serial number triple: `["Bob", 59, "123−45−6789"]`, or even a list of lists: `[ [1, 2], [3, 4, 5], [6] ]`. You can think of the concept of a list as a template for any sequence of data, of any type whatsoever. All of the structures we will look at in this chapter are templates for assembling simpler data into packages so that one element of the structure might be composed of many individual data items. These structures differ in the mechanisms for how they are created, how data is inserted into them, and how that data can be accessed. Your job as a programmer is to choose the data structure that fits the situation you are coding in the most natural way possible.

There are several concepts that are common to all three of the structures we will look at. One of these concepts is the notion of *indexing* the structure. This is the mechanism through which data can be retrieved from a structure. We have seen indexing before, with strings: if `s` is a string, then `s[i]` is the *ith* letter making up `s`. All three of our new structures are indexed in the same way, using square brackets and index values: `s[i]` is the value in the structure associated with index `i`. What differs from structure to structure is the kind of index the structure allows: some use numeric indexes and some use other kinds of data for indexes.

Another important concept in Python concerns whether a structure is *mutable* or *immutable.* Once immutable structures are created, the values in them cannot be changed. For example, strings are immutable. Suppose we create a string and store it in variable `x` with the statement

```
x = "Hi, Mom"
```

Then `x[1]` refers to the second element of this string, the letter `'i'`, but we are not allowed to write `x[1] = 'o'`; an error will result. We could write

```
x = x + '!'
```

to extend the string, but this creates a new string `"Hi, Mom!"` and stores that in `x`; the old string was not modified.

Lists, on the other hand, are mutable structures. Suppose we start with the list `L = [1, 3, 5]`. Then `L[1]` is the value 3. This time we are allowed to modify the contents of the list:

```
L[1] = 6
```

changes the list to `[1, 6, 5]`. Lists come with an `append()` method that inserts a value onto the end of the list. If `L` is this list `[1, 6, 5]` then

```
L.append(3)
```

turns `L` into `[1, 6, 5, 3]`. There is no `append` operation for our immutable strings. The closest thing to `append` for strings is the concatenation operator +, which creates a new list from the content of two old lists.

One place where this matters is passing a structure to a function. Suppose we want a function `StringAppend(string, tail)` that concatenates the tail on to the string. We would need to write this:

```
def StringAppend(string, tail):
        return string + tail
```

and call it in the following way:

```
s = StringAppend("bob", "by")
```

On the other hand, we could write a list function

```
def AddToEnd(list, x):
        list.append(x)
```

We might call this as follows:

```
L = [1, 3]
AddToEnd(L,5)
```

This changes `L` to the list `[1, 3, 5]`. There is no way for a function to modify a string because strings are immutable, but a function can modify the contents of a list – lists are mutable.

6.2 Lists

Lists are the most important type of structured data. They are used in every branch of programming. Lists appear in one form or another in almost all programming laguages. Python in particular makes much use of lists. Python also has some very useful tools for working with lists, which makes this the natural data structure to use in many situations.

Lists and strings share many common features. As with strings, we can use numeric indexes to retrieve the individual elements of a list. Consider, for example, the list

```
L = ["first", "second", "third"]
```

Then L[0] is the initial element of L, the string "first", while L[2] is the string "third". We can use the + operator to concatenate two lists, and the ∗ operator to multiply a list times an integer. As with strings, we can ask if object x is in L:

```
if x in L:
```

and we can find its index with L.index(x).

There is one significant difference between lists and strings: lists are mutable structures and strings are immutable. This means that we can change the contents of a list after it is created; this is not true of strings. Lists have an append() method that adds its argument onto the end of the list. For example, we might say:

```
L = []
L.append( "John" )
L.append( "George" )
L.append( "Paul" )
L.append( "Ringo" )
```

This turns L into the list ["John", "George", "Paul", "Ringo"].

Similarly, we can have a function modify the contents of a list. Think carefully about the next example. We know that the following program with integer variables won't work:

```
def Change(x):
    x = 24

def main():
    num = 4
    Change(num)
    print(num)
```

This prints 4; in spite of the function name, variable num isn't changed by the call Change(num).

Contrast this with the following program, which is similar but uses a list instead of an integer argument for the function call:

```
def ChangeList( L ):
    L.append(24)

def main():
    L = [4]
    ChangeList(L)
    print(L)

main()
```

This time the change function, here called ChangeList( ), does modify something. If we run this program it prints the list [4, 24]. The difference is that function ChangeList( ) changes the contents of list L, it doesn't change the list value, which is the address in memory where the list is stored. This value is what is sent to the ChangeList( ) function. At every point while this program is running the list variables L, both the L in main() and the L in ChangeList(), all contain the locations of the same list.

Here is a third related example. At first glance it looks as though this program will print [24] as the value of L, but this is not correct.

```
def ChangeListBadly( L ):
    L = [ ]
    L.append(24)

def main():
    L = [4]
    ChangeListBadly(L)
    print(L)

main()
```

Remember that each function's variables are visible only to that function. The variable L in function ChangeListBadly( ) and the variable L in main() are two completely different variables that happen to have the same name. When we call ChangeListBadly(L) these variables initially have the same value (i.e., they refer to the same memory location where a list is stored), but as soon as function ChangeListBadly( ) says

```
L = [ ]
```

this is no longer the case. This assignment statement attaches to variable L in ChangeListBadly( ) a completely different list from that in variable L of main( ). After this, the append statement appends onto this new list rather than list L of main( ). This program ends up printing [4], and the call to ChangeListBadly(L) has no effect. When a list is passed to a function, it is seldom useful to have the function re-assign to the list variable; that destroys the connection between the list argument to the function and lists that occur anywhere else in the program.

Here are some of the common operations with lists:

A. Making lists:
  L = [] (the empty list, which is the list with no elements)
  L = [”abc”, ”de”, ”fghij”, 1, [2, 3]]: this list has 5 elements:
    three strings, one integer and one list.
  L = L1 + L2, whereL1 andL2 are lists. This concatenatesL1 andL2
    into a new listL.
  L = L1 * 3, whereL1 is a list. This makes a new listL, which is the
    concatenation ofL1 3 times, as inL1 + L1 + L1.

B. Indexing:
  L[0]: the first element in listL
  L[1]: the second element in listL
  L[2:5]: a *slice* of listL, which is a new list consisting of the elements
    at positions 2, 3, and 4 (but not 5) ofL.

C. Changing the contents of the list, without changing the list itself:
  L[i] = a changes the value of theith entry ofL toa
  L.append(x): addsx to the end of the listL
  L.extend(L1): whereL1 is a list. This adds all the entries ofL1 ontoL
  L.sort(): sorts, or arranges in order, the entries ofL
  L.sort(compare): again, this sorts the entries ofL, usingcompare
    as a function to compare two entries.compare(a, b) should return
    -1 ifa ¡ b, 0 ifa == b, and 1 ifa ¿ b
  L.reverse(): reverses the order of the entries of L
  del L[i]: deletes theith element ofL
  L[i:j] = [] deletes the indexi throughj slice ofL

D. Other methods
  len(L): the length, or number of entries, ofL
  for x in L: iterates a loop over all entries ofL
  x in L: returnsTrue ifL has an entry whose value isx
  L.index(v): returns the index of the first entry ofL that equalsv;
    generates an *exception* (i.e. crashes) ifL does not containv

Here is an example that illustrates the power and ease of use of Python's list structures. We will write a that allows the user to enter a list of names, edit it, put it in alphabetical order, and then print it out. Each of these steps can be handled by its own function since the list contents can be modified by a function.

The main( ) function for this program is quite easy. For the first version we will dispense with editing and just read and print the list. For this version main( ) is just:

```
def main():
    L = [ ]
    ReadNames(L)
    PrintNames(L)
```

Function ReadNames( ) is our usual input loop for strings; each time we get a non-blank string we append it onto the list:

```
def ReadNames(L):
   done = False
   while not done:
      name=raw_input("Enter a name, or a blank to quit: ")
      if name == "":
         done = True
      else:
         L.append(name)
```

A simple **for**-loop is all that is needed to print the list:

```
def PrintNames(L):
    print "Here is the list: "
    for name in L:
        print(name)
```

Putting main( ), ReadNames( ) and PrintNames( ) together yields the first version of our program. If we add one more line to main( ) we can get the names to print in alphabetical order:

```
def ReadNames(L):
    done = False
    print("Enter names, and a blank to quit.")
    while not done:
        name=input("Enter a name: ")
        if name == "":
            done = True
        else:
            L.append(name)

def PrintNames(L):
    print("Here is the list: ")
    for name in L:
        print(name)

def main():
    L = [ ]
    ReadNames(L)
    L.sort()
    PrintNames(L)
```

Program 6.2.1: Reading and printing names, first version

Here is a typical user interaction with this program; note that the output is alphabetized:

```
Enter names, and a blank to quit.
Enter a name: john
Enter a name: paul
Enter a name: george
Enter a name: ringo
Enter a name:
Here is the list:
george
john
paul
ringo
```

Editing the list is a little more interesting. While more elaborate edits are possible, we will keep this simple by just allowing the user to delete names. A **for**-loop can run through the list and ask the user if each name should be deleted. However, modifying a list while it is being used to sequence a **for**-loop is a very bad idea that can lead to mysterious bugs that are hard to detect. Rather than this, we will build up a "delete list" of names to be removed, then in a second loop perform the deletions. Python makes it easy to create and process lists, so this is a common solution to programming problems in Python.

To delete an individual name from a list we need to know the index of this item in the list. Lists have an index( ) method that returns the position of any item in the list. It is important to insure that the argument to this method is actually an element of the list; your program will crash if you ask for the index of something that isn't in the list. Fortunately, we will only ask for the index of names in the delete list, which all come from the original list. Here is all that is needed to delete the entry stored in variable name from list L:

```
i = L.index(name)
del L[i]
```

Putting these ideas together, here is the DeleteNames( ) function:

```
def DeleteNames(L):
    print "Which of these should be deleted?"
    deleteList = []
    for name in L:
        print(name)
        print("Delete?")
        response=input("Enter 'y' to delete: ")
        if response == 'y':
            deleteList.append(name)
    for name in deleteList:
        i = L.index(name)
        del L[i]
```

Here is the complete code for the final version of our program:

```
# This programs allows the user to read,
# a list is names, edit that list, alphabetize it,
# and then print it.
def ReadNames(L):
    # This reads names and stores them in a list
    done = False
    print("Enter names, and a blank to quit.")
    while not done:
        name=input("Enter a name: ")
        if name == "":
            done = True
        else:
            L.append(name)

def DeleteNames(L):
    # This runs through L and asks if each
    # entry should be deleted.
    print("Which of these should be deleted?")
    deleteList = []
    for name in L:
        print(name)
        print("Delete?")
        response=input("Enter 'y' to delete: ")
        if response == 'y':
           deleteList.append(name)
    for name in deleteList:
        i = L.index(name)
        del L[i]

def PrintNames(L):
    # This prints all the entries of L
    print "Here is the list: "
    for name in L:
        print(name)

def main():
    L = [ ]
    ReadNames(L)
    DeleteNames(L)
    PrintNames(L)

main()
```

Program 6.2.1: Reading and printing names, final version

6.3 Dictionaries

Lists hold sequences of data of any type. Sometimes we want to do more than that. In many situations it is useful to associate related values. For example, a gradebook associates individual students with lists of grades. A concordance associates a word with places where it occurs. An inventory associates items with counts of how many there are. An associative data structure and operations to manipulate it are built into Python. This structure is called a *dictionary* and it is very useful.

Because lists are sequential, we can use numeric indexes to refer to the individual elements of a list. L[0] is the first element of listL, L[1] is the second element, and so forth. A list with 5 elements always uses the indexes 0, 1, 2, 3, 4. Dictionaries use a similar notation, with slightly different terminology. We say that a dictionary associates *values* with *keys*. The keys play the role of the indexes of a list, only the keys do not need to be numeric. For example, if "bob" is a key of dictionary D, then D["bob"] is the value associated with "bob". We can create this association by assigning a value to D["bob"], such as D["bob"]= 57 or D["bob"]= [1, 2, 3]. We retrieve the value associated with "bob" by referring to D["bob"], as in **print**(D["bob"]).

Dictionaries can have any immutable type for their keys. The immutability requirement comes from the way dictionaries are implemented internally. In practice, this requirement means that dictionaries cannot have lists as their keys. There are situations where it would be nice to have lists as keys. For example, we might want to represent a date by the list [month, day, year] (such as [9, 30, 2009]) and use such dates as keys for a dictionary. For these situations there is an immutable type similar to lists called *tuples*. We will discuss tuples in the next section. Strings, integers, and tuples are the most common keys for dictionaries.

The *empty dictionary* is denoted by the symbol {}. Although there are mechanisms to construct a dictionary from lists, we usually start with the empty dictionary and add items to it one at a time. Since dictionaries aren't sequential, we need to do a little work to print the information from one. Here is an easy way to do this. If D is a dictionary, then D.keys( ) is a list of all of the keys of D. Since it is a list, we can process this with a **for**-loop, and then use D[key] to access the value associated with each key.

For example, if the values associated with each key are printable, we might use the following function to print the dictionary:

```
def PrintDictionary( D ):
    for key in D.keys():
        print(D[key])
```

As with lists, dictionaries are mutable structures, so we can pass them as arguments to functions that modify the contents of the dictionary. Perhaps the biggest issue with dictionaries is the need to check that a key exists before we try to find the value associated with it. This doesn't occur with lists; if a list has 4 entries we know they are indexed with 0, 1, 2, and 3. A dictionary could have almost anything as its keys, so knowing the size of the dictionary tells us nothing about its keys. Suppose, for example, that we have a program that reads words and records how many times each of them occurs. We store this information in a dictionary Counts, where the words are the keys and numbers of instances of the words are the values. When we read a word, we don't *a priori* know if it is one of the keys or not. We could write

```
Counts[word] = 1
```

but if word has been seen before this will destroy any prior information about the number of times it has already occurred. We could write

```
Counts[word] = Counts[word] + 1
```

but this won't work if word has not been seen before, as there won't be a value for the right-hand side of the assignment. The right way to address this is to use a conditional statement:

```
if word in Counts.keys():
        Counts[word] = Counts[word] + 1
else:
        Counts[word] = 1
```

It is an error to index a dictionary with a something that is not currently a key of the dictionary; if you do this, your program will crash.

Example. We will now develop a simple inventory program that tracks items and counts. This program stores its information in a dictionary with strings (the item names) for the keys of a dictionary and numbers (the counts) for the values. This repeatedly asks the user for an item and a count. The first time we see an item we create a record for it. If the item comes up again, we add its count onto the existing count for that item. For example, we might have the following interaction with this program:

```
Enter an item or a blank to exit: hammer
Enter a count: 10
Enter an item or a blank to exit: screwdriver
Enter a count: 24
Enter an item or a blank to exit: saw
Enter a count: 5
Enter an item or a blank to exit: hammer
Enter a count: 15
Enter an item or a blank to exit: saw
Enter a count: 3
Enter an item or a blank to exit:
Here is the current inventory:
    hammer: 25
   saw: 8
   screwdriver: 24
```

Each iteration of the input gives us valuesa for item and count. This information is added to the dictionary in one of two ways. If the item is not already a key in the dictionary, it is added with

```
D[item] = count
```

On the other hand, if item is a key, then we increase its value by count:

```
D[item] = D[item] + count
```

This is very similar to the code above where we counted words; before updating the dictionary we check to see if the new index is already one of the dictionary's keys, and take appropriate action based on the answer to this question. We need such code in almost every program that makes use of dictionaries.

Here is the complete program:

```
# This keeps track of an inventory of items
# and counts.

def AddToDictionary(D, s, n):
# This adds todictionary D the information that n units
# of item s were found.
    if s in D.keys():
        D[s] = D[s] + n
    else:
        D[s] = n

def PrintDictionary(D):
# This prints the keys and values of the dictionary
#  with the keys in alphabetical order
    names = D.keys()
    names.sort()
    for item in names:
        print( "%s: %d" %(item, D[item]) )

def main():
    inventory = {}
    done = False
    while not done:
        item = input("Enter an item or a blank to exit: " )
        if item == "":
            done = True
            print()
        else:
            count = eval( input("Enter a count: " ) )
            AddToDictionary(inventory, item, count)
    PrintDictionary(inventory)

main()
```

Program 6.3.1: Inventory Program

6.4 Tuples

Old programming languages such as C have strict rules for the programmer for the ways in which data must be handled. The burden of showing that a program is consistent in its data usage is put on the programmer. As a result, programs in these languages can run very efficiently; little needs to be checked during the execution of a program. Python reverses this. It places little burden on the programmer, and this means that things that are verified in other languages when the program is written need to be checked during run-time on Python programs. For the most part this is a good trade-off; it enables Python programs to be written by many people who are not professional programmers, and processors have become so fast that we seldom notice the increased time for the execution. Occasionally, however, this tradeoff does not work well for us; we would like to get faster execution, even at the cost of more elaboration at programming time. One way to achieve this is to use data structures with less flexibility.

We have seen that the list data type in Python is useful in many situations because of its mutability. We can create an empty list and gradually add members to it as we have a need to do so. We can insert members, delete members and alter members of a list whenever we need to. This gives great power to the programmer, but it also requires that every list be created with the ability to expand without limits. Python has a similar data type called a *tuple* that has many of the same properties as lists, but tuples are immutable. Once a tuple is created, its contents cannot be altered. Tuples place fewer demands on the system, so programs that use them might run faster than similar programs that use lists. More important for our purposes, there are things that Python will let you do with tuples that you cannot do with lists. Among the most important of these is serving as keys for dictionaries. Tuples can be keys, lists cannot.

Tuples are written as comma-separated sequences of values inside parentheses, as in `(3, 4, 5)`. The parentheses can be eliminated if you choose: the expression `2, 3` is interpreted by Python as the tuple whose first element is 2 and whose second is 3; this is more commonly written `(2, 3)`. If you need a tuple with only one element 2, it is written `(2,)`. Tuples are made differently than lists: we often start with an empty list and build it up one element at a time, but since tuples are immutable we have to start tuples with their complete values.

Tuples are indexed the same way as lists and dictionaries, with square brackets. If `T` is a tuple, then `T[0]` is its first element, `T[1]` is the second element, and so forth.

One thing that you can do in Python that is not allowed in older languages is assigning to a structure in which the entries are variables. For example, Python allows the statement

```
(x, y) = (3, 5)
```

and the effect of this statement is to assign 3 to `x` and 5 to `y`. This works with both the tuple `(x, y)` and the list `[x, y]`, but since we would be unlikely to alter a structure containing several variables, tuples are the more natural structure to use for such an assignment.

The following program illustrates the way we could use this to have a function return multiple values. Here function MinMax( ) takes as an argument a list of numeric values and returns a tuple containing the smallest and largest values in the list. Note that in main( ) we can assign a tuple of variables to the value returned by a call to MinMax( )

```
# This finds the largest and smallest values of a list.
def MinMax(L):
    # This returns the largest and smallest values of list L.
    # We assume that all the values in L are between 0
    # and 100.
    min = 101
    max = -1
    for x in L:
        if x < min:
            min = x
        if x > max:
            max = x
    return (min, max)

def main():
    L = [2, 9, 1, 3, 5, 4, 2, 7, 5]
    (m, M) = MinMax(L)
    print( "The smallest of those values is %d" % m )
    print( "The largest of those values is %d" % M )

main( )
```

Program 6.4.1: Largest and smallest values of a list

. In the next program we store and edit a list of names, where names are represented by (givenNames, familyName) tuples. Since we do not plan to edit the names after they are created, tuples are the natural structure here. Since tuples, like strings, are immutable, we could use such a structure any place where we use strings, such as for the keys of a dictionary.

Our program needs to take a number of possibilities into account. Some people have only one name, as in "Madonna" or "Prince". We will ultimately sort the names by the family name field, so we will put this single name as the family name, which is the second entry in the tuple, and for these use an empty string in the given name field. Some people have more than one given name, as in "Barack Hussein Obama" or "Charles Philip Arthur George Windsor". We will assume that the last of the words in such a string is the family name and put everything else into one long given name.

```
# This reads a list of names and sorts it.
# The names are stored as (givenName, familyName)
def ReadName():
    response = input( "Enter a name: " )
    names = response.split()
    if len(names) == 0:
        return ("", "")
    else:
        lastName = names[ len(names) - 1 ]
        if len(names) == 1:
            firstName = ""
        else:
            firstName = names[ 0 ]
        return (firstName, lastName )

def PrintList(L):
    # This prints list L
    for entry in L:
        print( "%s %s" % (entry[0], entry[1]) )

def compare(name1, name2):
    # This returns -1 if name1 < name2,
    # 0 if name1 == name2,
    # and 1 if name1 > name2
    if name1[1] < name2[1]:
        return -1
    elif name1[1] > name2[1]:
        return 1
    elif name1[0] < name2[0]:
        return -1
    elif name1[0] > name2[0]:
        return 1
    else:
        return 0

def main():
    PersonList = []
    done = False
    while not done:
        p = ReadName()
        if p[1] == "":
            done = True
        else:
            PersonList.append(p)
    PersonList.sort (compare)  # Note this use of compare
    PrintList( PersonList )

main()
```

Program 6.4.2: Sorting (givenName, familyName) tuples

6.5 Sets

A *set* is a data structure that holds single copies of a number of elements. The most important property of a set is that it has no repeats: if an item is in the set and we try to add it again, the set is unchanged. Note that adding an element to the set that is already present in the set is not an error, it just doesn't change the change the set. You have encountered sets in mathematics classes where they seem fairly obvious and not very interesting, just an excuse for more terminology. They play a more interesting role in programming. Sets have some useful operations. For example, if word1 and word2 are two words, then **set**(word1)−**set**(word2) is the set of letters that are in word1 but not in word2, while **set**(word1)&**set**(word2) is the set of letters in both. Sets are also efficient ways to store data for lookups. If a set s has n elements, you can determine if an item is in s or not in time proportional to log(n). For a list this operation takes time proportional to n. The difference doesn't matter when n is small, but if n is large it might take 20 steps to find if something is in a set and one million steps to find if the item is in a list with the same elements as the set. That is enough of a difference to show up in the user experience of running your program.

We create an empty string with quotes, an empty list with square brackets, an empty tuple with parentheses, and an empty dictionary with curly brackets. We are out of delimiters. To make an empty set, we use the function **set**(). There are actually similar constructors for making the other structures: **list** (), **tuple**() and **dict**() make empty lists, tuples and dictionaries, respectively. If we have elements to put in the set at the time it is created we can write it with curly brackets, as in {1, 2, 3}. Note that the elements of a set must be *hashable*. It is hard to describe what constitutes being hashable without going into the details of how sets are implemented, but numbers and the immutable structures (strings and tuples) are all hashable. Sets themselves are mutable.

Here are some of the common operations with sets:

A. Making sets:
  S = set() (the empty set, which is the set with no elements)
  S = {3, 6, 9} makes S be a set consisting of the numbers 3, 6, and 9.
  S = set(L), where L is a list, string, or tuple: makes S be a set whose
    elements are the elements of L
    S = set(”abc”) is the same as S = {”a”, ”b”, ”c”}
  S.copy() returns a new set that is a copy of set S

B. Operations:
  A+B is the union of sets A and B. This can also be written A.union(B)
  A&B is the intersection of sets A and B.
    This can also be written A.intersect(B)
  A-B is the difference of sets A and B: the elements of A that are not in B.
    This can also be written A.difference(B)
  A<=B is True if A is a subset of B: all of the elements of A are also in B
  A<B is True if A is a *proper* subset of B:
    A is a subset of B and not equal to B

C. Changing the contents of a set:
  S.add(x) adds element x to set S if x is not already an element of S
  S.remove(x) deletes element x from set S;
    this causes an error if x is not an element of S
  S.discard(x) deletes element x from set S
    but does not cause an error if x is not an element of S
  S.clear() removes all elements of set S

D. Other methods
  len(S): the length, or number of entries, of set S
  for x in S: iterates a loop over all entries of set S
  x in S: returnsTrue ifx is an element of set S
  x not in S: returnsFalse ifx is an element of set S

We will now look at two examples of sets. The first example is a short program that builds up a set of the user's favorite colors. We make an empty set called favorites with the line

```
favorites=set()
```

and then go into our usual input loop that reads strings as variable color until we get to an empty string. We exit the loop on an empty string; when variable color is not empty we add it to the set with

```
favorites.add(color)
```

At the end we print the contents of the set with

```
for x in favorites:
    print(x)
```

Here is the complete program:

```
def main():
    # This builds up and then prints a set of
    # favorite colors
    done = False
    favorites = set()
    while not done:
        color = input( "Favorite colors? " )
        if color == "":
            done = True
        else:
            favorites.add(color)

    for x in favorites:
        print(x)

main()
```

Program 6.5.1: Creating and printing a set of favorite colors

Our second example of sets is more elaborate. Here is a statement of the problem:

Write a program that inputs a single string from the user. The program should print every possible rearrangement of the letters of that string, with each rearrangement printed only once.

One way we will use sets in this program is for finding unique rearrangements. Each time we generate a rearrangement we'll add it to a set; this way duplicates will be ignored. At the end we'll just print the elements of this set. A more interesting use of sets occurs in creating the rearrangements. We will find all of the rearrangements by first finding all of the possible permutations of the numbers (0, 1, 2, ... n-1), where n is the length of our string, and using these permuations to make the rearrangements.

An example will help clarify this algorithm. Suppose the string we start with is "abc"; we'll call this string word. We start by finding the set of all permutations of the numbers (0, 1, 2): {(0,1,2),(0,2,1),(1,2,0),(1,0,2),(2,0,1),(2,1,0)} Each of these makes one rearrangement of word. For example, with permutation (1,2,0) our rearrangement is word[1]+word[2]+word[0], which is "bca". Here is code for a function that takes variable word and rearranges it according to permutation t:

```
def reArrange(word, t):
    newWord = ""
    for i in t:
        newWord = newWord + word[i]
    return newWord
```

So far we have reduced our problem to the problem of finding all permutations of the numbers (0,1,2,..., n−1). There are a number of ways to do this. An easy, though not particularly efficient way makes another use of sets. We will maintain a set of all of the permutations we have found so far; initially this contains only the single tuple (0,1,2,..., n−1). At each step we take one of the permutations from this set and run through all of the possible indexes i and j of this permutation. We interchange the elements at index i and index j, forming a new permutation, which we add to our set. This keeps going until we have found all of the permutations. How many are there? There are n numbers in (0,1,2,..., n−1), so there are n choices for the first entry. The one chosen can't be reused, so there are n−1 choices for the second entry, n−2 choices for the third entry, and so forth. Altogether there are n∗(n−1)∗(n−2)∗...∗1, or factorial (n) possible permutations.

This algorithm makes the following code:

```
permutations = set()  #makes an empty set
permutations.add(start) #start is the initial tuple (0,1,2,...N-1)
while len(permutations) <  maxSize:  #maxSize is factorial(N)
    for t in permutations:
        for i in range(0, N-1):
            for j in range(i+1, N):
                newT = switch(t, i, j) #switches i and j elements
                permutations.add(newT)
```

There is one problem with this code. We have a **for**-loop: **for** t **in** permutations: and in the body of this loop we modify the permutations set. Python doesn't allow this. To fix this we change the loop to use a *copy* of the current permutations set, and we add newT to the *actual* permutations set:

```
permutations = set()  #makes an empty set
permutations.add(start) #start is the initial tuple (0,1,2,...N-1)
while len(permutations) <  maxSize:  #maxSize is factorial(N)
    for t in permutations.copy():
        for i in range(0, N-1):
            for j in range(i+1, N):
                newT = switch(t, i, j) #switches i and j elements
                permutations.add(newT)
```

Now, suppose the size of our string is N=4. Factorial(4) is 24; suppose we have found 23 of the 24 permutations. If we loop through every one of these permutations and every possible pair of indices i and j, we might find the only remaining permutation quickly and then be stuck doing a lot of extra, unnecessary work. We can avoid this by placing a break statement inside the **for**-loop on t:

```
    if len(permutations) == maxSize:
        break
```

Our code so far calls a function switch(t, i, j) to interchange two elements of tuple t. Here is the algorithm for this. We have chosen indices i and j in such a way that we know i<j. So we first copy the elements of t with indices less than i, followed by the entry at index j, followed by the elements of t with indices from i+1 to j−1, followed by the entry at index i, and that followed by the portion of t with indices greater than j:

```
def switch(t, i, j):
    newT = t[0:i]+(t[j],)+t[i+1:j]+(t[i],)+t[j+1:]
    return newT
```

The following program puts all of this together:

```
# This program inputs a word and prints all of the possible
# anagrams of that word.

def factorial(n):
    # This returns the product of the numbers from 1 to n
    prod = 1
    for x in range(2, n+1):
        prod = prod*x
    return prod

def switch(t, i, j):
    # This assumes t is a tuple and
    # i and j are indexes into t, with i<j
    # This returns the result of interchanging
    # t[i] and t[j]
    newT = t[0:i]+(t[j],)+t[i+1:j]+(t[i],)+t[j+1:]
    return newT

def reArrange(word, t):
    # This arranges the letters of word according to the
    # order specified by permutation t.
    newWord = ""
    for i in t:
        newWord = newWord + word[i]
    return newWord

def main():
    word = input( "word? " )
    N = len(word)
    maxSize = factorial(N)
    start = tuple(range(N)) #(0,1,,2,...,N-1)
    # First we find all of the permutations of start
    permutations = set()
    permutations.add(start)
    while len(permutations) <  maxSize:
        for t in permutations.copy():
            if len(permutations) == maxSize:
                break
            for i in range(0, N-1):
                 for j in range(i+1, N):
                    newT = switch(t, i, j)
                    permutations.add(newT)
    # Next we make a set of the anagrams of word
    anagrams = set()
    for t in permutations:
        anagrams.add( reArrange(word, t))
    # Finally we print that set.
    for w in anagrams:
        print(w)

main()
```

Program 6.5.2: Anagrams of a word

If we run this with word="read" we get the 24 permutations of those 4 letters:

```
dear , dare , aedr , ared , erad , eard , aerd , reda ,
eadr , ader , daer , drae , adre , raed , drea , erda ,
dera , arde , edra , rdea , read , rdae , edar , rade
```

However, if we run it with word="bob" we only get 3 since other permutations would result in duplicates:

```
obb , bob , bbo
```

6.6 Exercises

Write a complete Python program to solve each of the following problems.

6.1. Write a program that reads a list of numbers from the user; the list terminates when the user enters 0. After all of theinput the program should print the list.

6.2. Change the program from problem 6.1 so the list is printed in reverse order.

6.3. Change the program from problem 6.1 to use a tuple instead of a list.

6.4. Change the program from problem 6.1 to print the average value of the list as well as each of the values in the list.

6.5. Write an inventory program for a store, similar to program 6.3.1. This should let the user enter objects and the number of each in the inventory. The input terminates when the user gives the empty string as an object name. After this the program prints the entire inventory, alphabetized by the names of the objects. For example, the input might be

```
Object: widget
Number of widgets: 24

Object: thingamabob
Number of thingamabobs: 3

Object: gizmo
Number of gizmos: 15

Object:
```

The output will then be:

```
gizmos:       15
thingamabobs: 3
widgets:       24
```

Make the user interaction as natural as you can. Note the use of singular and plural objects in this example.

6.6. Change the program from problem **??** so that the output is in order from smallest number to largest number. For the example given it will be

```
thingamabobs: 3
gizmos:       15
widgets:      24
```

6.7. Write a program that inputs a list of values (it doesn't matter if they are numbers or strings) and then prints all of the values that appear in the list more than once.

6.8. Write a program that inputs a list of numbers and outputs basic statistical information about the list: minimum and maximum values, average value, standard deviation (the square root of variance; you can find this in any book on statistics), and median (middle value).

6.9. Write a program that stores in a dictionary a list of people's birthdays. To make the input simple, read in the three fields of the birthday (month, day, year) on separate lines. Your input might look like this:

Enter name: `bob`
Enter month: `2`
Enter day: `16`
Enter year: `1952`
Enter name:

It this could keep up until you get a blank name.With each name you should enter the name and date in a dictionary, with the name as the key, and a tuple for the date as a value. After all of the input, have your program print all of the birthdays in the dictionary.

6.10. Add a `LookupName` feature to the program from 6.9. This should ask the user for a name, and then print the birthday for that person. Make sure your program doesn't crash if you give it the name of someone who isn't in the dictionary.

6.11. Add a `LookupBirthday` feature to the program from 6.9. This should ask for a birthday, and then print all of the people who had that birthday.

6.12. Try rewriting the program from problem 6.9 so that the keys of the dictionary are the dates (use a tuple `(m, d, y)` for this) and the values are lists of all people with that date as their birthday.

Chapter 7

Files

Files provide permanent storage of data. Any program that retains data from one execution of the program to another stores that data in files, whether the user can see those files or not. For example, any program that has a ”preferences” tab that allows you to change some features of the program must store those preferences in a file. In this chapter we will look at ways to incorporate files into our programs, using files both for input (reading data from a file) and output (storing data in a file).

7.1 Concepts

There are two basic types of files: *text* files, which consist of a sequence of characters, and *binary* files, which contain binary encodings of information. Microsoft Word, and many other commercial programs that format information, create binary files because data can usually be packed much more efficiently into binary files than into text files. A Python program can open up binary files, but in order to obtain useful information from a binary file you need to know the way it is encoded; this information may or may not be available to you. In this chapter we will only look at text files. You can create a text file with the Python editor. Most word processing programs, such as Microsoft Word, also enable you to save files in text format.

Since files exist apart from your programs, an extra step is needed to make a file available to your program. This is called *opening* the file. You must always open a file before using it. This provides a link between the physical file and your program. When you open a file, you must say how you are going to use it: whether you want to read data out of it or write data into it. You may want to do both, but only one of these can be done at any given time.

The **open** function is

```
open(<file name>, <mode> )
```

where both the <**file** name> and <mode> are strings. This returns a *file object* that you need to reference each time you use the file, so you should save it in a variable, as in

```
F = open( <file name>, <mode> )
```

In this open statement the mode can be one of three strings:

- "r" means you want to open an existing file to read information from it. Your program will crash if the file is not found.
- "w" means you want to create a new file so you can write information into it. Any previous file with this file name will be destroyed.
- "a" means you want to append information at the end of a new or existing file.

For example, we could open for reading a file whose name is "data.txt" with the statement

```
F = open("data.txt", "r")
```

Note that to open a file you need to know the full name, including any file extension such as ".txt" Many modern software packages hide the file extension; you need to know it in order to write a program that opens a file with this extenstion. If you create a text file to use in a Python program, use a standard, explicit file extension, such as ".txt".

It is possible to specify a full path to a file, such as

```
F=open("C:\Documents and Settings\Bob\My Documents\data.txt","r")
```

If the file is in the same folder as the Python program that uses it, you only need to give the file name rather than its full path. For example, we might use

```
F=open("data.txt","r")
```

if the "data.txt" file is in the same folder as the program. We will later see a graphical way to obtain the path to a file; for now we will just work with files in the same folder as our programs.

With other data structures there are simple ways to move around in the structures. We call these "random access" structures because we can move directly to arbitrary positions in the structure. With files random access is much more difficult. We generally read files from beginning to end. When writing files we also write from beginning to end, rather than inserting characters into the middle of a file. This makes some file manipulations awkward. It is usually easier to work with lists and other structures internal to Python than with files. A simple paradigm for handling files is:

- Read the data from your file into a list or other structure at the start of your program.
- Manipulate the list or other structures.
- At the end of the program write the data from your structures back into the file.

7.2 Writing Files

To open a file for writing you must use either

```
F = open(<file name>, "w")
```

or

```
F = open(<file name>, "a")
```

The value returned by the **open( )** statement is a *file object*; we need this object in order to manipulate the file. The "w" mode causes a new file to be created. If a file with this name already exists in the current folder, it will be destroyed and replaced by the new file. The "a" mode will append onto the end of an existing file, or create a new one if no file with the given name already exists. In either case each write statement writes onto the end of the file.

You can write a string to file object F with

```
F.write( <string> )
```

Note that this does not pad the string with any spaces, and it does not terminate any line in the file. The statements

```
F.write("26")
F.write("35")
```

will put "2635" into the file. You need to add any separators that you want as you write the data. If you wish to have the file organized into lines of text, as most text files are, you need to directly write the line breaks: "\n" is the symbol we use for this. So to write the string "This is a line." into file object F we would say

```
F.write("This is a line.\n")
```

Finally, note that you can only write a string into a text file. If you want to output numbers, you can write them with formatted strings, as in

```
F.write("%d\n"%value)
```

or you can use **str()**, as in

```
F.write( str(value)+"\n" )
```

Here is a simple program that reads a list of names from the user and writes these names into a file called "names.txt".

```
def main():
    F = open( "names.txt", "w" )
    done = False
    while not done:
        name=input("Enter a name, or a blank to exit: ")
        if name == "":
            done = True
        else:
            F.write(name+"\n")
main()
```

Program 7.2.1: Saving strings to a file

The next program builds a file that contains a table of prime numbers. This time we build up the data in a list, and print the list to a file at the end. Rather than trying to divide all numbers into a given candidate, we will use the fact that all smaller primes have already been entered into our list; these are the only potential divisors that we need to check. The resulting program finds all prime numbers up to one million within a few seconds.

The important function here is `WriteToFile( L )`, which writes list `L` to the file `"primes.txt"`. This starts by opening the file, then writing the entries one at a time into the file. We use a formatted string:

```
F.write( "%10d" % entry )
```

to guarantee that there will be white space around each number: this uses 10 spaces per entry, and the entries have no more than 6 digits. Since we use multiple write statements per line, the formatted print statement also guarantees that our numbers come out in neat columns.

Unless we want thousands of numbers to be printed on the same line, at some point we need to give line breaks. Eight entries per line makes for a nice table. We have a variable `lineCount` that keeps track of how many entries are on the current line. When this gets up to 8 we terminate the line with

```
F.write( "\n" )
```

and reset lineCount to 0.

```
# This builds a table of primes numbers.

def isPrime(x, L):
    # This returns True is x is prime,
    # where L is the list of all primes
    # less than x.
    for d in L:
        if x%d == 0:
            return False
        elif d*d > x:
            return True
    return True

def FindPrimes( L ):
    # This puts all primes less than 1000000 into L
    for x in range(2, 1000000):
        if isPrime(x, L):
            L.append(x)

def WriteToFile( L ):
    # This writes the entries of L to a file "primes.txt",
    # printing 8 entries per line
    F = open( "primes.txt", "w")
    lineCount = 0
    for entry in L:
        F.write( "%10d" % entry )
        lineCount = lineCount + 1
        if lineCount == 8:
            F.write( "\n" )
            lineCount = 0

def main():
    Primes = []
    FindPrimes( Primes )
    WriteToFile( Primes )

main()
```

Program 7.2.2: Saving prime numbers to a file

7.3 Reading Files

To open a file for reading you must use

```
F = open(<file name>, "r")
```

The file must exist and be in the right folder; it is an error to try to open a file that doesn't exist in the folder where the system expects to find it. For what we are doing now, the system looks for the file in the folder that contains the program. So until we talk about moving around in a file system, you should place any file your program will read in the same folder as the program itself. If you don't, your program will crash with a "File not found" exception.

If F is an open file – i.e., if it is the value returned by a call to **open**, there are several ways to read data from F:

F.read() returns the entire file as one long string, using \n characters for line breaks

F.read(n) returns the next **n** characters from the file as a string.

F. readline () returns the next line of the file as a string.

F. readlines () returns all of the file as a list of strings, each string representing one line of the file.

You can also read the entire file with a for-loop:

```
for line in F:
    ...
```

In most situations it is easiest to read through the whole file with a **for**-loop. Text files have a special end-of-line character that marks the end of each line; this is usually denoted "\n". When you iterate through a text file with a **for**-loop, this end-of-line character remains attached to the lines; your first step is usually to eliminate it:

```
for line in F:
        line = line.strip("\n")
        ...
```

After that, the line is a standard string that you can treat the way you would treat any string.

Here, for example, is a program that will print its own text to the screen.

```
# This program prints the file "Program7.3.1.py",
# which happens to contain the text of this
# program.

def main():
    F = open( "Program7.3.1.py", "r" )
    for line in F:
        line = line.strip("\n")
        print(line)

main()
```

Program 7.3.1: This prints its own text

Next, we have a program that reads a column of numbers and prints its average. As with many situations when working with files, we dump the file of numbers into a list, and then use our previous techniques to find the average of the list. Note that the lines of the file are strings, so we need to convert these lines into numbers before we average them.

```
def Average(L):
    sum = 0
    count = 0
    for x in L:
        count = count + 1
        sum = sum + x
    return sum/count

def main():
    F = open( "nums.txt", "r")
    numbers = []
    for line in F:
        x = int(line)
        numbers.append(x)
    print( "I count %d numbers." % len(numbers) )
    print( "Their average is %.2f" % Average(numbers) )

main()
```

Program 7.3.2: This averages the numbers in a data file.

If this program sees a file "nums.txt" containing the numbers

```
2
9
4
3
5
6
```

it outputs:

```
I count 6 numbers.
Their average is 4.83
```

As a final example, here is a similar program that reads a data file with two columns of numbers, such as

```
3   29
4   45
7   52
5   43
```

```
def Avg(L):
    sum = 0
    count = 0
    for x in L:
        count = count + 1
        sum = sum + x
    return sum/count

def main():
    F = open( "2cols.txt", "r")
    column1 = []
    column2 = []
    for line in F:
        nums = line.split()
        x = int(nums[0])
        y = int(nums[1])
        column1.append(x)
        column2.append(y)
    avg1 = Avg(column1)
    avg2 = Avg(column2)
    print( "I count %d rows." % len(column1) )
    print( "The averages were %.2f and %.2f" % (avg1,avg2) )

main()
```

Program 7.3.3: This averages two columns of data from a file.

7.4 Try-Except Statements

In programming terminology an *exception* is a terminal event, typically caused either by an error in the code or by an unexpected input from the user. The usual behavior of a program in the event of an exception is to crash. The **try−except** statement allows you to catch exceptions before they cause your program to crash. The form of this statement is

```
try:
        <the try-block of code>
except:
        <the except-block of code>
```

When this statement is executed, the system executes the **try**-block. If this block of code causes no problems the **except**-block is ignored. However, if the **try**-block causes an error the system executes the **except**-block. The **except**-block should make up in some way for the fact that the **try**-block did not execute correctly.

For example, consider the following program:

```
def main():
    done = False
    while not done:
            x = eval(input( "Enter a number: " ))
            if x < 0:
                done = True
            else:
                print("100/%d = %d" %(x, 100/x))
main()
```

Program 7.4.1: A program that generates an exception

If the user enters a number such as 4 the program outputs "100/4=25". If the user enters a negative number the program terminates. However, if the user enters 0 the program crashes. We can use a **try−except** statement to avoid this as follows:

```
def main():
    done = False
    while not done:
        try:
            x = eval(input( "Enter a number: " ))
            if x < 0:
                done = True
            else:
                print("100/%d = %d" %(x, 100/x))
        except:
            print("I can't divide by that.")

main()
```

Program 7.4.2: A fix for the previous program

Now if the user enters a 0 the program responds with "I can't divide by that." The program goes back to the top of the input loop and asks for another number. There is no crash and the program continues to function.

It is possible to make **except**-clauses respond only to specific types of exceptions. For example, the exception that is raised when we divide by 0 is called a `ZeroDivisionError`. If we make the **except**-clause

```
except ZeroDivisionError:
```

then the block of code for this clause will be executed only for divisions by zero; any other type of exception will not be handled by this clause. Here is a third version of the program:

```
def main():
    done = False
    while not done:
        try:
            x = eval(input( "Enter a number: " ))
            if x < 0:
                done = True
            else:
                print("100/%d = %d" %(x, 100/x))
        except ZeroDivisionError:
            print("I can't divide by that.")
        except NameError:
            print("I said a NUMBER, doofus.")
        except:
            print("Now, that's  just dumb.")
main()
```

Program 7.4.3: Catching specific exceptions

Now, if the user enters 4 the program responds ”100/4=25” If the user enters 0 the program responds ”I can’t divide by that.” If the user enters the string ”ten” the program responds ”I said a NUMBER, doofus.” Finally, if the user enters something silly like ”23 skidoo” the program responds ”Now, that is just dumb.”

Chapter 8

Objects and Classes

The style of programming we have seen so far is called *procedural programming.* This was the first programming paradigm, developed in the 1950's and 1960's alongside the growth of hardware technology. In this chapter we look at *object-oriented programming*, a more modern style of coding that was developed in the 1970's and 1980's. The motivation for the development of object-oriented programming was the repeated failure of programmers to eliminate bugs from large programs written in the procedural style. Object-oriented programming is not more powerful than procedural programming any program that can be written in one style can be written in the other. However, in many situations it is easier to write correct programs in the object-oriented style. In Chapter 8 we will look at programs that use graphics, animation and simulations, and you will see that all of these lend themselves naturally to the object-oriented style.

8.1 Concepts

There is some terminology that you must understand to get started on writing object-oriented programs. The terminology is not difficult, but it is uses the building blocks we have already seen in a new way. Here are the basic pieces.

A *class* is a structure that holds both data and functions for manipulating that data. We have seen some classes before without calling them classes. For example, Python has a class called List. A list holds sequences of whatever data you give it, and it has functions, or *methods*, such as append( ) for manipulating that data. The class itself is a pattern. It says what kinds of data and what kinds of functionality instances of the class will have. You can construct specific instances of a class, which are called *objects*. For example, we might have a class Person. This would describe the data and functionality needed to represent people. Each object of this class would represent one individual person.

Like all data, the data in an object is stored in variables. There are two types of variables in classes: *instance variables* and*class variables*. Each object of a class has its own copy of each of the instance variables, so each object can have different data stored in the instance variables. For example, in our Person class there might be an instance variable to hold a persons name. Each object of the class will represent a different person, and so each will have its own name. The class variables are shared by all objects of the class; there is only one copy of the class variables, and it is visible to all objects of the class. Our Person class might have a class variable to hold the population size. The number of people in existence is the same regardless of which person we ask for this number. We refer to the instance variables of the class through objects of the class using a dot-notation: <object_name>.<variable_name>. For example, if x is an object of class Person we would refer to the name instance variable as x.name. We refer to the class variables through the class itself. Person.population might be the class variable that represent the size of the Person population.

Objects have their own functions for manipulating their data. We call the functions contained in an object its *methods*. If class Person has a GetOlder( ) method that adds one to the age of a person, and if x is an object of this class, we would tell object x to get older with the code

```
x.GetOlder( )
```

You might think of x.GetOlder() as a command to object x to increase its age variable.

As we said, each object of a class has its own copies of the instance variables and methods of the class. The class definition needs some way to refer to an individual object's data and methods. The word self is used for this. This word only appears in class definitions; it always refers to *the current object*. Every method of every class has self as its first argument. For example, the GetOlder( ) method has header

```
def GetOlder( self ):
```

When we call this method on a specific object, as in x.GetOlder( ), the object x is passed as the argument for self. We might have a method SetAge( ) that would set the persons age to a specific value. This would have header

```
def SetAge( self , myAge ):
```

and would be called as in x.SetAge(19). All methods have self as their first argument; you never explicitly pass a value for self, but the system substitutes the object whose method you are invoking for this argument.

The word self is used for similar reasons in references to instance variables within the methods of a class. self.age is the age variable for whatever object is being referenced. For example, here is the complete code for the SetAge( ) and GetOlder( ) methods:

```
def SetAge( self , myAge ):
        self.age = myAge

def GetOlder( self ):
        self.age = self.age + 1
```

If we call x.SetAge(19), then the object stored in variable x is passed for the self argument in method SetAge( ). Then self.age, which is the value of the age instance variable of object x, is set to 19. This notation probably seems cumbersome at first, but with a little practice it will begin to seem natural: x.foobar() runs method foobar() on object x.

Every class has a specific method called a *constructor* that is run when a new object of that class is created. The purpose of a constructor is to give initial values to the instance variables of the class. If the constructor has arguments, values for those arguments must be provided when a new object is constructed. Constructors in Python all have the name __init__( ). Of course, since constructors are methods they all have self as their first argument. For example, we might want the constructor for the Person class to give a name to the new person. The constructor that does this is

```
def __init__(self , myName ):
        self.name = myName
        self.age = 0
```

Remember that the constructor should initialize all of the instance variables of the class, even those that are not mentioned in the constructor's arguments. If we fail to initialize the variable then it doesn't exist. For example, if the constructor for class Person omits the line

```
self.age = 0
```

then new persons will not have an age variable. If other methods, such as a Print( ) method, refer to the age of the person and the age variable doesn't exist, our program will crash when these methods are called.

We construct objects of a class by using the class name as a function. The function that is actually called is the __init__( ) method of this class, so Python expects us to provide all of the arguments (except self) of this method. For example, the __init__( ) method above for class Person needs a string for the name, so we could construct a new person with

```
x = Person( "bob" )
```

Here, finally, is the complete code for a simple program that uses classes. This defines the class Person that we have been discussing. The main( ) function of the program makes use of this class definition to construct several persons, set the age variables, and print out their information. Note that the main( ) function is simple and intuitive. This is typically the case with object-oriented programs the hard work in such programs is implementing the classes; once that is accomplished they are usually easy to use. Note also that classes can be used in more than one program. One of the goals of object-oriented programming is to make reusable classes that can be written carefully and then used in a wide range of applications. A programmer doesn't need to know the details of how a class is implemented in order to use it. All the programmer needs to know is what data the class holds and what methods it offers. The List and Dictionary classes in Python are illustrations of this. We have used these classes to write many programs without any knowledge of how the classes are implemented. These classes are very powerful. It is the object-oriented paradigm that allows us to separate the implementation of the classes from their use.

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def SetAge(self, myAge):
        self.age = myAge

    def GetOlder(self):
        self.age = self.age + 1

    def Print(self):
        print("%s is %d years old."%(self.name,self.age))


def main():
    x = Person( "bob")
    y = Person( "suzie")
    z = Person( "joe")

    x.SetAge(57)
    x.GetOlder()
    x.GetOlder()
    x.Print()

    y.Print()

main()
```

Program 8.1.1: A complete object-oriented program

8.2 Examples of Classes

Don't be put off by the terminology of classes we presented in the preceding section. *Designing* classes that appropriately break down complex data is a skill that takes practice to learn. In most situations *implementing* classes is easy and straightforward. One nice thing about coding with classes is that you always have a place to start – the constructor (remember that in Python this is a function called `__init__ ( self ...)`) should assign a value to each of the instance variables of the class. Whatever description you have of the class should tell you what those instance variables are.

Example 1: *Write a class that will hold data about pets. The class should include variables for the pet's name, species (dog, cat, hamster, etc.), color and age.* To implement this we need to make a few decisions. One concerns changeability of the variables: once we set the species for a pet, there is probably no need to ever change it. The same is true of the pet's color. It may or may not be true for the name; for simplicity in this example we will assume that names are not modifiable. Age, on the other hand, is certainly something that changes. For all of the variables we will create *getter* methods – methods that return the value of the variables. For the changeable variables we also need to create *setter* methods – methods that allow us to set or change the value of the variable. The next decision regards the arguments to the constructor. All of these variables will be set in the constructor; some will get default values, some will get values specified in the arguments to the constructor. We don't want to make calls to the constructor too long, so it would be better to not have too many arguments. The pet's name and species seem like things we would want to name when we create a pet; age and color might be specified later, so we'll make name and species be arguments to the constructor. One could certainly justify other ways of doing this. Finally, we need to decide how to represent `name`, `species`, `color` and `age`. The pet's age is certainly a number; the other three can easily be respresented as strings.

All of these decisions lead to the following code for the constructor:

```
def __init__(self, name, species):
    self.name = name
    self.species = species
    self.color = "black"
    self.age = 0
```

As we said, this passes `name` and `species` as arguments and uses default values for the other two variables. Don't be confused by code such as

```
self.name=name
```

The left side of the = sign has the instance varible `self.name`. The right side of the = sign has the value we are assigning this variable, which is the value of the first argument to the constructor. If we construct a specific pet, such as

```
R = Pet( "Rosie", "cat")
```

the instance variable elf.name} will be et to "Rosie".

After the constructor we define any getter and setter for each of the instance variables. This is very easy code. For variable name we don't need need a setter and the getter is a one-line function:

```
def getName(self):
    return self.name
```

for variable color we also have a setter:

```
def getColor(self):
    return self.color

def setColor(self, c):
    self.color = c
```

For variable age we have a getter and setter and also an *increment* method that needs no arguments; it just adds 1 to the pet's age:

```
def getAge(self):
    return self.age

def setAge(self, x):
    self.age = x

def incrementAge(self):
    self.age = self.age + 1
```

Finally, we add a Print() method to format the instance data of an object:

```
def Print(self):
    print("%s is a %s %s."%(self.name, self.color, self.species))
```

Putting all of this together, here is our class definition:

```
class Pet:
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.color = "black"
        self.age = 0

    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def getColor(self):
        return self.color

    def setColor(self, c):
        self.color = c

    def getAge(self):
        return self.age

    def setAge(self, x):
        self.age = x

    def incrementAge(self):
        self.age = self.age + 1

    def Print(self):
        print("%s is a %s %s."
              %(self.name, self.color, self.species))
```

Program 8.2.1: A simple Pet class

Notice how simple and repetitive most of this code is. lt is not difficult to write code for a class once you understand what the class does; this is one of the great attractions of this style of coding.

Here is a small main() program that uses this class defintion:

```
def main():
     R = Pet("Rosie", "cat")
     R.setColor("black")

     N = Pet("Norton", "cat")
     N.setColor("orange")
     N.incrementAge()
     N.incrementAge()

     R.Print()
     N.Print()
main()
```

An application program for the Pet class

Example 2: *Write a class that represents people with names and ages and also models the ability of one person to marry another.* In the preceding section we gave a Person class:

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def SetAge(self, myAge):
        self.age = myAge

    def GetOlder(self):
        self.age = self.age + 1

    def Print(self):
        print("%s is %d years old."%(self.name,self.age))
```

The Person class from Program 8.1.1

We need to modify this class to include the idea of marriage. Actions in classes tend to take the form of methods – when we call a method of an object, we tell the object to perform that method on its data. For this class we need a method marry() that tells one object to marry another. In terms of data what does this do? After this method has called and returned, the object still needs to know who it married. This means we need an instance variable spouse. This will need to be initialized in the constructor; the easiest default value is None, which is the value Python uses for objects that are not yet constructed. For this example we won't make use of age, so we will eliminate to keep our program shorter. There would be no problem with keeping it. This makes our constructor

```
def __init__(self, myName):
    self.name = myName
    self.spouse = None
```

We will only let Persons marry other Persons, which means that we have access to the instance variables of the marriage partner. To enable self to marry Person x, we need to set self.spouse=x, but that is not sufficient. We also need to make self be x's spouse: x.spouse=self:

```
def Marry(self, x):
    # A person can only marry another Person
    self.spouse = x
    x.spouse = self
```

It is easy to tell if a Person is married. The following method returns True if the current person is married, and False if that person is single:

```
def IsMarried(self):
    return self.spouse != None
```

If x is married we can always find the name of x's spouse as x.spouse.name, but this is a bit unwieldy. Here is a method that returns the name of self's spouse. Of course there might not be a spouse, so we include an option for this case as well:

```
def SpouseName(self):
    if self.IsMarried():
        return self.spouse.name
    else:
        return "not married"
```

Note how this uses the IsMarried() method: we want to know if self is married, so we call the self.IsMarried() method.

Finally, here is a Print method that makes use of several of the methods we have implemented:

```
    def Print(self):
        if self.IsMarried():
            print( "%s is married to %s" %
                      (self.name, self.SpouseName()) )
        else:
            print( self.name )
```

Here is the complete code for our new class Person:

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.spouse = None

    def Marry(self, x):
        # A person can only marry another Person
        self.spouse = x
        x.spouse = self

    def IsMarried(self):
        return self.spouse != None

    def SpouseName(self):
        if self.IsMarried():
            return self.spouse.name
        else:
            return "not married"

    def Print(self):
        if self.IsMarried():
            print( "%s is married to %s" %
                       (self.name, self.SpouseName()))
        else:
            print( self.name )
```

Program 8.2.2: Person class with marriage

And here is a main() function that makes use of this class

```
def main():
    x = Person( "bob")
    y = Person( "suzie")
    z = Person( "joe")

    y.Marry(z)

    x.Print()
    y.Print()
    z.Print()

main()
```

An application program for the Person class

Example 3: *Write a class that represents names of people and can recognize family names and given names.* Our last example defines a Name class where the instance variables hold the given name (self.given) and the family name (self.family) of each object. The constructor takes a string, such as "Fred Flintstone" and separates it into a string holding the given name or names, and a string holding the family name. Note that there may be more than one given name, as in "Charles Philip Arthur George Windsor". We could represent this as a list of strings, but instead we will put all of the given names into one string and the family name in another. In most situations we want one of three things:

- The family name only
- The first name (the first of the given names) only
- The full name

All of those will be easily available in our representation.

The constructor is mostly a straightforward application of the String class strip() and split() methods. The idea is to split the argument into a list of strings, then pull of the last of these as the family name and concatenate the rest together, separated by spaces, as the given name. If **str** is the input string (such as "Fred Flintstone"), the following code does this:

```
names = str.split()
n = len(names)
self.family = names[n-1]
given = ""
for name in names[0:n-1]:
```

```
            given = given + name + " "
        self.given = given
```

We need to modify this slightly to account for anomalies: **str** might contain only one name, as in "Madonna", or it might even be the empty string. Single names are easy: we just put whatever we have into the family name, and make the given name be the empty string. An empty **str** is more of a problem. We can split an empty string, but the result is an empty list and our code above fails for an empty list. We fix this by first stripping **str** to eliminate any unnecessary spaces at the front and back, and then testing whether the result is an empty string. If so we assign the empty string to both the given name and family name and exit. If not, we use the code above. Here is the final code for the constructor:

```
    def __init__(self, str):
        str = str.strip()
        if str == "":
            self.family = ""
            self.given = ""
        else:
            names = str.split()
            n = len(names)
            self.family = names[n-1]
            given = ""
            for name in names[0:n-1]:
                given = given + name + " "
            self.given = given
```

The remaining methods handle the tree cases we mention above: family name only, given name only, first name only, or full name. Here is our class:

```
class Name:
    def __init__(self, str):
        str = str.strip()
        if str == "":
            self.family = ""
            self.given = ""
        else:
            names = str.split()
            n = len(names)
            self.family = names[n-1]
            given = ""
            for name in names[0:n-1]:
                given = given + name + " "
            self.given = given

    def GivenName(self):
        return self.given

    def LastName(self):
        return self.family

    def FirstName(self):
        if self.given == "":
            return ""
        else:
            names = self.given.split()
            return names[0]

    def FullName(self):
        if self.given == "":
            return self.family
        else:
            return self.given+self.family
```

The Name class

Here is a short application program,which just works with the names as names:

```
def main():
    L = []
    L.append( Name("Fred Flintstone") )
    L.append( Name("Barney Rubble") )
    L.append( Name("Wilma Flintstone") )
    L.append( Name("Dino") )
    L.append( Name( "") )
    L.append( Name("Bam Bam Rubble") )
    L.append( Name( "Mr. Slate" ) )

    for x in L:
        print(x.LastName())

main()
```

Program 8.2.3: Application program using the Name class

This will print:

```
Flintstone
Rubble
Flintstone
Dino

Rubble
Slate
```

Here is a more interesting application. For this example we put the class by itself in a file called "MyNameClass.py". In a more authentic situation we would probably call the file "Name.py", but I want to make a distinction between the name of the file and the name of the class. We write a separate application program, making use of the Name class. In this case our application is another Person program; this time the Person class uses Name to store the person's name. Since the Name class is not in the same file as the application, the latter program needs to import it. We use the syntax

```
from MyNameClass import *
```

and then refer directly to the Name class. Alternatively, we could use the following syntax for importing:

```
import MyNameClass
```

and then refer to the names as MyNameClass.Name. For example, the second line of the constructor would be

```
self.name = MyNameClass.Name(myName)
```

Here is the application program:

```
from MyNameClass import *

class Person:
    def __init__(self, myName):
        self.name = Name(myName)
        self.age = 0

    def SetAge( self, a ):
            self.age = a

    def GetOlder(self):
            self.age = self.age + 1

    def Print(self):
        print( "%s %s" %(self.name.FirstName(),
                         self.name.LastName()))

def main():
    L = []
    x = Person("Donald Ervin Knuth")
    x.SetAge( 77 )
    L.append(x)

    y = Person("Grace Brewster Murray Hopper")
    y.SetAge(109)
    L.append(y)

    z = Person("Alan Mathison Turing")
    z.SetAge(103)

    L.append(z)
    for person in L:
        person.Print()

main()
```

Program 8.2.4: Another application program using the Name class

8.3 Special Methods

There are a number of method names that have special significance in Python. One of these we have already seen: the constructor method is always named `__init__( )`. This method is called whenever a new object of the class is created; its purpose is to give initial values to the instance variables of the object. In this section we will see a number of similar methods that have pre-defined meanings. All of these have names that start and end with two underscores.

First, the method `__str__(self)` is called whenever the system needs to have a string representation of the object. This method should return the string representation. If x is an object of a class containing this method, the following statements will all result in calls to `__str__( )`:

```
print(x)
print("%s" % x)
y = str(x)
```

For example, the Person class from section 7.1 might have such a method:

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def __str__(self):
        return "%s is %d years old."% (self.name,self.age)
```

Program 8.3.1: A Person constructor and `__str__` method

This would eliminate the need for a separate `Print( )` method for this class; we could use the standard Python **print** statement to print objects of the class. Of course, nothing requires us to return a string containing all of the instance variables of the class. For some applications we might want the string representation of a person to consist of just the persons name:

```
def __str__(self):
        return self.name
```

We can also define methods that implement arithmetic operators in any class. The methods:

```
__add__(self, x)
__sub__(self, x)
__mul__(self, x)
__div__(self, x)
```

are called when the operators +, −, ∗,and / are used. Each of these methods should return a new object that is the result of the operation. For example, if a and b are objects of a class that defines these operators, we might use the statement

```
c = a+b
```

Variable c then gets the value that is returned from the call to the method __add__(a, b). Argument self refers to the object that is the left operand and argument x is the right operand.

In the next example we define a class Cents that represents money. Objects of this class have one instance variable, which holds the value of the object in pennies (so a value of 420 represents $4.20). We define a __str__( ) method to allow objects of the class to be printed, and an __add__( ) method to allow monetary values to be added.

```
class Cents:
    def __init__(self, x):
        self.value = x

    def __str__(self):
        dollars = self.value/100
        cents = self.value % 100
        if cents < 10:
            return "$%d.0%d" % (dollars, cents)
        else:
            return "$%d.%d" % (dollars, cents)

    def __add__(self, x):
        v = self.value + x.value
        return Cents( v )

def main():
    x = Cents(405)
    y = Cents(995)
    print("%s + %s = %s" % (x, y, x+y))

main()
```

Program 8.3.2: Adding elements of a class

We can also implement methods that allow us to use comparison operators between objects of a class. The method names

```
__lt__(self, x)
__le__(self, x)
```

```
__eq__(self, x)
__ne__(self, x)
__ge__(self, x)
__gt__(self, x)
```

refer to the operations <, <=, ==, !=, >=, >. In particular, if the `__lt__()` method is defined, then lists of objects of this can be sorted with the list `sort()` method.

Our last example adds comparison operators to the `Name` class we created in Section 8.2. We use the usual phone-book ordering for names: `a < b` if `a`'s last name comes before `b`'s in alphabetical ordering, or if the two last names are the same and `a`'s first name comes before `b`'s first name. In the name class the instance variable that holds the last name is `self.family`, and the variable that holds the first name is `self.given`. Our comparison operator is thus

```
def __lt__(self, x):
    if self.family < x.family:
        return True
    elif self.family > x.family:
        return False
    elif self.given < x.given:
        return True
    else:
        return False
```

This turns the class definition into

```
class Name:
    def __init__(self, str):
        str = str.strip()
        if str == "":
            self.family = ""
            self.given = ""
        else:
            names = str.split()
            n = len(names)
            self.family = names[n-1]
            given = ""
            for name in names[0:n-1]:
                given = given + name + " "
            self.given = given

    def GivenName(self):
        return self.given

    def LastName(self):
        return self.family

    def FirstName(self):
        if self.given == "":
            return ""
        else:
            names = self.given.split()
            return names[0]

    def FullName(self):
        if self.given == "":
            return self.family
        else:
            return self.given+self.family

    def __lt__(self, x):
        if self.family < x.family:
            return True
        elif self.family > x.family:
            return False
        elif self.given < x.given:
            return True
        else:
            return False
```

The Name class with a comparison operation

We now return to the Person class that uses Name. If we add a simple comparison function to this, just comparing the names, we can then sort lists of Persons:

```
import MyNameClass

class Person:
    def __init__(self, myName):
        self.name = MyNameClass.Name(myName)
        self.age = 0

    def SetAge( self , a ):
            self.age = a

    def GetOlder(self):
            self.age = self.age + 1

    def Print(self):
        print( "%s %s" %(self.name.FirstName(),
                          self.name.LastName()))
    def __lt__(self, x):
        return self.name < x.name

def main():
    L = []
    L.append( Person("Harry Potter") )
    L.append( Person("Hermione Granger") )
    L.append( Person("Ron Weasley") )
    L.append( Person("Albus Dumbledore") )
    L.append( Person("Severus Snape") )
    L.append( Person("Draco Malfoy") )

    L.sort()
    for person in L:
        person.Print()


main()
```

Program 8.3.3: Sorting elements of the Person class

This outputs

```
Albus Dumbledore
Hermione Granger
Draco Malfoy
Harry Potter
Severus Snape
Ron Weasley
```

8.4 Subclasses

There are many natural situations where some elements of a class have more or different properties than the other elements. For example, all people have names and addresses. Some people, the students, also have majors and gpas. Other people have job titles and annual salaries. Subclasses give a way to model these situations. A subclass refines a class by adding more instance variables or new methods.

There is a variety of terminology for classes and subclasses. Some people call the class being refined the "super class". Some people call the subclass a "child class" and the class it refines the "parent class". Whatever terminology you prefer, the refining class is generally smaller than the class it refines. Every element of the subclass is also an element of the parent class. The converse of this is false. For example, every student is a person, but not every person is a student. This means that all of the instance variables and all of the methods of the parent class are available to the subclass. We say the subclass *inherits* the data and methods of the parent class. The subclass may alter the definition of the methods of the parent class, or it may use them unaltered.

Why do we use subclasses? One reason is that they allow us to avoid re-implementing the methods of the parent class. Anything that allows us to re-use code helps to avoid errors in the code. Another reason is that this is a good organizational tool. Suppose we have classes that represent various geometric shapes: Square, Circle, and Polygon. If they are all subclasses of a basic Shape class, and if this Shape class has a method Moveto( ) then we know they all have such a method. This means we could put any of these shapes into a list and move them by processing the list with a **for**-loop without worrying which class a particular shape is from.

A subclass definition looks just like a class definition, only it contains the name of the parent class in parentheses, as in

```
class Circle(Shape):
        ...
```

Python allow a class to inherit from several parent classes. Setting up multiple inheritance correctly is not easy and we generally will avoid the problems this creates by only giving classes a single parent.

Subclasses inherit the methods of their parent classes. If the subclass provides a new definition of a method of its parent, the subclass has access to both version of the method, with the parents version accessed as <**class** name>.<method name>(self,) For example, subclasses usually overwrite the constructor method __init__ ( ) of the parent. If class B is a subclass of class A, then inside B we can refer to As constructor as A. __init__ ( )

Here is an example of two classes: Person and Student, with Student a subclass of Person. Note that the Student constructor calls the Person constructor, and also the method SetAge( ) method of class Person. Class Student does not give a new version of the __str__ ( ) method of class Person; it just uses the version inherited from Person.

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def SetAge(self, a):
        self.age = a

    def __str__(self):
        return "%s, who is %d years old"%(self.name,self.age)

class Student(Person):
    def __init__(self, myName):
        Person.__init__(self, myName)
        self.SetAge(18)
        self.major = " "

    def SetMajor(self, subject):
        self.major = subject

    def Major(self):
        return self.major

def main():
    x = Student("joe")
    x.SetAge(21)
    x.SetMajor("Computer Science" )
    print(x)

main()
```

Program 8.4.1: Class Student is a subclass of Person.

In the next example we model the payroll office of a company that has three kinds of workers: hourly workers, who have an hourly wage and work 40 hours per week, salaried workers who have an annual salary and are paid every fourth week, and interns,who arent paid at all. Each class of workers has a method Pay( ) that takes as argument a week number (the weeks are numbered 0, 1, 2, and so forth). The hourly workers are paid each week. The salaried workers are paid when the week%4 is 3 (i.e, on the last week of each month). The constructor for each class calls the Employee constructor, as that assigns a unique employee number to each staff member. The individual subclass constructors also save their own information, in addition to the work of the Employee constructor.

Here is the top class: Employee:

```
# Class Employee has the code common to all employees,
# There are 3 subclasses that refine Employee with more
# details.

class Employee:
    Count = 0
    def __init__(self, name, jobTitle):
        self.name = name
        self.title = jobTitle
        Employee.Count = Employee.Count + 1
        self.employeeNumber = Employee.Count

    def Print(self):
        print "Name: %s  Employee Number: %d   Title: %s" % \
                  (self.name, self.employeeNumber, self.title)

    def Pay(self, week):
        pass
```

Program 8.4.2: The top-level class.

Note the use of the **pass** command in Employees Pay( ) method. This is one way to have a function in Python that does nothing — the system ignores the **pass** statement. Functions must have something in the body, so some statement that has no effect is necessary.

Next we have the definitions of the three subclasses of Employee: HourlyWorker, SalariedWorker, and Intern. Each of these only needs to implement the portion of their functionality that is not covered in class Employee.

```
class SalariedWorker(Employee):
    # Salaried workders are paid once a month
    # (every 4th pay period). Then they are paid
    # one twelfth of their annual salary
    def __init__(self, name, jobTitle, annualSalary):
        Employee.__init__(self, name, jobTitle)
        self.salary = annualSalary

    def Print(self):
        print( "%s is a salaried worker with salary $%d" %\
                       (self.name, self.salary))

    def Pay(self, week):
        if week % 4 == 3:
            print( "Pay %s $%.2f" % \
                       (self.name, self.salary/12.0))

class HourlyWorker(Employee):
    # Hourly workers are paid every week.
    # Each period they are paid 40 times their hourly wage.
    def __init__(self, name, jobTitle, hourlyWage):
        Employee.__init__(self, name, jobTitle)
        self.wage = hourlyWage

    def Print(self):
        print( "%s is an hourly worker with wage $%d" % \
                       (self.name, self.wage))

    def Pay(self, week):
        print( "Pay %s $%.2f" % \
                       (self.name, self.wage*40))

class Intern(Employee):
    # Interns aren't paid
    def __init__(self, name):
        Employee.__init__(self, name, "Robot")

    def Print(self):
        print( "%s is an intern" % (self.name))
```

Program 8.4.2: 3 subclasses of Employee.

Finally, we have two utility functions: one for maintaining a list of all employees and one for processing this list to pay all of the employees that need paying in a given week. Note that we can put employees into the employee list in any order, and we can process this list without worrying about what class each element of the list comes from. All of the objects have a Pay( ) method (even the unpaid interns; they inherit the default Pay( ) method from class Employee). This ability to manage a variety of classes as though there are all the same class is one of the great strengths of subclasses.

```
def Hire(list, employee):
    list.append(employee)

def PayWorkers(list, numWeeks):
    for week in range(0, numWeeks):
        print( "Pay period %d" % week)
        for employee in list:
            employee.Pay(week)

def main():
    Staff = []

    Hire(Staff,SalariedWorker("Suzie","Engineer",50000))
    Hire(Staff,SalariedWorker("Bob","Boss",120000))
    Hire(Staff,HourlyWorker("Fred","Construction Worker",10))
    Hire(Staff,Intern("Mary"))
    Hire(Staff,Intern("Herman"))

    for x in Staff:
        x.Print()

    PayWorkers( Staff, 12)

main()
```

Program 8.4.2: Finishing the program.

8.5 Object-Oriented Design

When you are faced with an open-ended problem of designing a program, object-oriented techniques can help. In many situations data elements can be organized into classes, which are usually easy to implement. If you can express the central algorithms of the problem in terms of these classes you probably have a good design for the program. For example, a program that records students grades (a virtual Registar program) might have a Person class, with subclasses for Students and Faculty, and a Course class, where a course would have an instructor and a list of Students. Each Student would have a current Schedule, with a list of classes, and a Transcript, which would include a list of class names and grades. A Student taking a Course would be represented by adding the student to the Course object's list of students,then at the end of the term getting a grade from the instructor and the course name and grade being added to the Student's transcript. By the time you implemented all of these classes the program would be mostly written.

For another example, suppose you are creating a predator-prey simulation. You would need a Predator class that describes the predator's ability to move around and locate the prey, and the predator's ability to kill nearby prey. You would have a corresponding Prey classes that describes the prey's ability to avoid being eaten. You might have an Environment class that both the Predator and Prey classes interact with. And you might have a World class that consists of a large grid, every cell of which has an Environment object and possibly one or more Predator and Prey objects.

Once you have identified possible classes for your program, try to write a brief description of each class. The closer the classes are to physical objects, or to objects that could be physical, the easier they will be to code. Try to write down the data elements of each class, and what kinds of Python representation each will have. Then write a description of each of the major methods of each class. You should do all of this before you start coding If you can describe the functionality of your program in terms of the class methods and you can see how to implement each of the methods, you have a winning program design that should be straightforward to implement. If you can't do these things there is no point in starting the coding. Try looking for a solution in terms of different class breakdowns.

As a demonstration of this we will develop a card game. This game will be played with a standard deck of 52 playing cards, with 13 cards in each of the suits Hearts, Spades, Diamonds and Clubs. To avoid implementing lengthy rules (which is better, a Full House or a Flush??) the game we will implement is the children's game War – on each round all of the players turn over their next card; the highest card wins. If there is a tie the players go again. The player who wins the most cards wins the game. This is not a very interesting game to play and it is even more uninteresting too watch, but the techniques and much of the code we will develop could be used to implement more interesting card games, from Go Fish to Texas Hold 'Em.

Our game will need 3 classes:

- class Card represents a single playing card. The data Card needs is the suit and name of the card. The names will be strings: "2", "3", "4", ..., up to "10", "Jack", "Queen", "King" and "Ace". For suits we could easily use the strings "Hearts", "Spades" and so forth, but for fun we will take advantage of Python's use of Unicode and use graphical representations of the suits. The Heart symbol is **chr**(9825), Spades is char(9824), Clubs is char(9827), and Diamonds is **chr**(9826). Cards don't do anything, so our only methods will be a constructor, a __str__ () method for printing, and a few comparison methods.

- class Deck will represent a deck of 52 cards. The only data is a list of the cards currently in the deck. Besides a constructor we need a method for shuffling the deck and another for dealing cards to players.

- class Player will represent one player of the game. This is a very simple class that keeps track of the player's name, current hand, and points. The only method this needs other than a constructor is a method that reveals the next card in the player's hand.

We will begin with the Card class. We'll define constants at the top that give the Unicode characters for our suits; this is the only place in the entire program we'll need to refer to these characters. Elsewhere we can refer to these suits as Card.HEARTS, Card.SPADES, and so forth. The constructor just assigns values; the real work has to be done by whoever calls the constructor.

Here is the complete code for the Card class:

```
class Card:
    HEARTS = chr(9825)
    SPADES = chr(9824)
    CLUBS = chr(9827)
    DIAMONDS = chr(9826)

    def __init__(self, name, suit, value):
        self.name = name
        self.suit = suit
        self.value = value

    def __str__(self):
        return "%s %s" %(self.name, self.suit)

    def __lt__(self, x):
        if self.value < x.value:
            return True
        else:
            return False

    def __gt__(self, x):
        return self.value > x.value

    def __eq__(self, x):
        return self.value == x.value
```

The Card class

Note the __lt__() and __gt__() methods. These give two different ways to write similar comparisons. In __lt__() we do the comparison with <, look at the result, and decide what to return. In __gt__() we realize that what we need to return is the result of the > comparison; we could just as easily have written __lt__() as

```
def __lt__(self, x):
    return self.value < x.value
```

The Deck class only has three methods, but each is more substantial than anything in the Card class. The constructor isn't difficult, but it requires some organization. We start by initializing the instance variable that will hold the list of cards in the deck to the empty list: self.cards=[]. Then, for each suit we add each of the number cards. Note that if x is a number betweem 2 and 10, str(x) is the string version of the number: "2" for 2, "10" for 10. After the number cards we add the Jack, Queen, King and Ace. The number on the number cards is the card's numeric value for comparison purposes; the value of a Jack is 11, that for a Queen is 12, for a King is 13 and for an Ace is 14. Note that in some card games an Ace is low, and in some it is either high or low; in ours it is always high. It would be possible to design a different class that had options for how the Ace is treated.

Here is the constructor code:

```
def __init__(self):
    self.cards = []
    for suit in [Card.HEARTS, Card.SPADES,
                 Card.DIAMONDS, Card.CLUBS]:
        for number in range(2, 11):
            self.cards.append(Card(str(number),
                                   suit, number))
        self.cards.append(Card("Jack", suit, 11))
        self.cards.append(Card("Queen", suit, 12))
        self.cards.append(Card("King", suit, 13))
        self.cards.append(Card("Ace", suit, 14))
```

In most card games the dealer cycles around all of the players giving each one card until everyone has the right number of cards. This guards against poorly shuffled decks and also provides some protection against a malicious dealer stacking the deck. We are not concerned about either of those problems since we will control how the cards are dealt, so it seems easier to have the Deal() method take the appropriate number of cards off the top of the deck and return them as a list of cards. We must remember to remove them from the deck so they can't be given to any other player. The only issue is what to do if the deck runs out of cards. Our solution is to give as many cards as the deck has, though you could make an argument for throwing an exception or taking other drastic action. Here is the resulting Deal method.

```
def __init__(self):
def Deal(self, n):
    # returns a list of n cards
    L = []
    for i in range( 0, n):
        if len(self.cards) > 0:
            card = self.cards[0]
            del self.cards[0]
            L.append(card)
    return L
```

The remaining method for the Deck class is Shuffle(). There are simple ways to randomize the elements of a list, but we are going to use a formal shuffling algorithm that has been proven to be fair: if the random number generator we use is good, our shuffle algorithm produces every possible ordering of the list and all of these orderings are equally likely. This is known as the *Fisher-Yates* *shuffle* since it was first defined by statisticians Ronald Fisher and Frank Yates in 1938. It is also called the *Knuth shuffle* since it was popularized by Donald Knuth in *The Art of Computer Programming.* The idea behind this algorithm is simple. We start at the end of the list and work our way towards the front. At each step we swap the current element with a random element closer to the front. When this is finished, the list is sorted:

```
def Shuffle(self):
    n = len(self.cards)-1
    while n > 0:
        k = randint(0, n)
        # swap the kth and nth cards
        A = self.cards[k]
        B = self.cards[n]
        self.cards[k] = B
        self.cards[n] = A
        n = n - 1
```

Altogether, here is the complete Deck class:

```
from Card  import *
from random import *

class Deck:
    def __init__(self):
        self.cards = []
        for suit in [Card.HEARTS, Card.SPADES,
                     Card.DIAMONDS, Card.CLUBS]:
            for number in range(2, 11):
                self.cards.append(Card(str(number),
                    suit, number))
            self.cards.append(Card("Jack", suit, 11))
            self.cards.append(Card("Queen", suit, 12))
            self.cards.append(Card("King", suit, 13))
            self.cards.append(Card("Ace", suit, 14))

    def Deal(self, n):
        # returns a list of n cards
        L = []
        for i in range( 0, n):
            if len(self.cards) > 0:
                card = self.cards[0]
                del self.cards[0]
                L.append(card)
        return L

    def Shuffle(self):
       n = len(self.cards)-1
       while n > 0:
           k = randint(0, n)
           # swap the kth and nth cards
           A = self.cards[k]
           B = self.cards[n]
           self.cards[k] = B
           self.cards[n] = A
           n = n - 1
```

The Deck class

With just this much we can implement a simple game I call *Cut the Deck*. We make a deck and two players each "cut the deck" by choosing an index between 0 and 51. The one who gets the higher card wins. If the cards have the same value they go again.

```
# This is a simple game simulating two people
# cutting a deck of cards.
# The game continues until one gets a higher
# card than the other.
from Deck import *

def cut(deck, player):
    index = eval( input("Where does %s cut the deck? "
                              % player))
    if index >= 52:
        index = 52
    elif index < 0:
        index = 0
    return deck.cards[index]

def main():
    D = Deck()
    player1 = input( "Who is the first player? " )
    player2 = input( "Who is the second player? " )
    done = False
    while not done:
        D.Shuffle()
        card1 = cut(D, player1)
        print( "%s draws %s" %(player1, card1) )
        card2 = cut(D, player2)
        print( "%s draws %s" %(player2, card2))

        if card1 > card2:
            print( "%s wins!" %player1 )
            done = True
        elif card2 > card1:
            print( "%s wins!" %player2 )
            done = True
        else:
            print( "Tie; play again." )

main()
```

Program 8.5.1: Game CutTheDeck

For our *War* game we need a player class. This will keep track of the player's name and hand of cards. In one round of the game each player plays one card;

since this involves multiple players it isn't a method of the Player class. Instead we give the class a method for playing one card, and put off the rest for our final implementation of the game. Here is the player class:

```
from Deck import *

class Player:
    def __init__(self, name, deck, numCards):
        self.name = name
        self.hand = deck.Deal(numCards)
        self.points = 0

    def nextCard(self):
        card = self.hand[0]
        del self.hand[0]
        print("%s plays %s" %(self.name, card))
        return card
```

The Player class

Finally, we need to implement the game itself. This has a play() function that has each player play one card. The cards are compared. If there is a winner the appropriate number of points (the number of cards won) are added to that player's total score; if there is no winner the play continues. Naturally, a loop describes this; it continues until one of the players wins or the players run out of cards.

The rest of the game is described in the main() function. This gets the players' names, constructs the players, and calls the play() method in a loop until the players run out of cards. At the end it reports the winner: the player with the highest score. Here is the full game:

```
from Player import *

def play(player1, player2):
    points = 0
    done = False
    while not done:
        c1 = player1.nextCard()
        c2 = player2.nextCard()
        points = points+2
        if c1 > c2:
            player1.points = player1.points + points
            print( "%s has %d points\n" %
                    (player1.name, player1.points))
            done = True
        elif c2 > c1 :
            player2.points = player2.points + points
            print( "%s has %d points\n" %
                    (player2.name, player2.points))
            done = True
        elif len(player1.hand) == 0:
            done = True

def main():
    deck = Deck()
    deck.Shuffle()
    player1 = Player( "bob", deck, 26)
    player2 = Player("marvin", deck, 26)

    while len( player1.hand) > 0:
        play( player1, player2)

    if player1.points == player2.points:
        print("Tie game")
    else:
        if player1.points > player2.points:
            winner = player1
        else:
            winner = player2
        print( "%s wins with %d points" %
                (winner.name, winner.points))

main()
```

Program 8.5.2: Game CutTheDeck

Notice how easy the game was to implement after all of our classes were defined. This is a sign of a successful object-oriented design. If you can't use the classes to solve the overarching problem you are working on, you probably haven't yet found a good class system for your problem.

8.6 Exercises

Write a complete Python program to solve each of the following problems.

8.1. Make a program containing class Student. This class should have two instance variables: name and age. The constructor takes a name argument and uses it to initialize the name instance variable; it also initializes the age instance variable to 18. The class has a SetAge() method that can set the age to any other value. Finally, the class has a Print() method. Here is a main() method for your program:

```
def main():
    x = Student( "bob" )
    x.SetAge(23)
    x.Print()
```

8.2. Give class Student in problem 8.1 an instance variable Gender, which can be "M" or "F". Modfiy the constructor for Student to take a gender argument as well as a name argument to give a value to this variable.

8.3. Give the Student class in problem 8.2another instance variable: transcript . This is a list of pairs, where the first element of the pair is the name of a course, and the second element is a numeric grade between 0 and 4.There are two additional methods:

AddCourse(self) which asks the user for a course name and grade, and appends this pair onto the transcript, and

GPA(self) which averages all of the grades in the transcript and returns the average.

The main program

```
def main():
    x = Student( "bob", "M")
    x.AddCourse( )
    x.AddCourse()
    print X.GPA()
```

will cause the system to twice ask for course name and grade. If you supply the following information:

Course name? `Basketweaving`
Grade for Basketweaving? `4`
Course name? `Time Wasting`
Grade for Time Wasting? `3`

then the print statement **print** X.GPA at the end of main() will print 3.5

8.4. Give class Student in problem 8.1 another instance variable Roommate. This can either have value None, or an object of class Student. Give the class a method AssignRoommate() to give a value to this variable.

8.5. Make another class Faculty in problem **??**, which just has one instance variable, varname. The only methods Faculty needs are a constructor and a Print() method. Add to class Student an instance variable advisor. This should be initialized to None in the Student constructor. Give Student a method AssignAdvisor() that assigns an advisor (a faculty member) to the student.

8.6. The relationship between Faculty and Student in problem 8.5 is too one-sided. Give to class Faculty an instance variable Advisees, which is a list of all of the advisees the faculty member has. Add to Faculty a method AssignAdvisee() that will append a student to the Advisees list. If x is a Student and y is a Faculty, then x.AssignAdvisor(y) and y.AssignAdvisee(x) should do the same thing.

Chapter 9

Graphics and User Interfaces

In the early days of personal computers Macintosh, Windows, and Unix systems had completely different sets of tools for creating graphical user interfaces. Porting a program from one system to another was very difficult because there was no simple way to translate the interface design from one platform to another. This was a difficult situation for both programmers and users. In 1991 John Ousterhout, who was a professor at UC Berkeley, announced the tk toolkit for building user interfaces. This was a system that could be implemented on all of the major platforms and programs would run the same way on any of them. tk is now built into the operating systems of almost all computers. Python provides a module called tkInter to interface with the tk library. In this chapter we will see ways to make use of this module.

This chapter makes lots of use of classes and subclasses, so if you are not on top of that material you might want to review Chapter /refChapter8 before proceeding. In this chapter you will see some of the great advantages of programming with classes: we can use the class structure to hide most of the ugly implementation details and only keep in front of us the portions of the code that change for our particular program. This makes user-interface programming, which used to be the realm of advanced professionals, accessible to all programmers.

9.1 Tk Concepts and Terminology

To get started wth tk we need a little terminology. A *widget* is any kind of graphical element: a window, a button, a checkbox, or anything else that holds information that the user can interact with. A *frame* is any kind of window that can hold other widgets. A *canvas* is a specific type of window that allows us to draw into it. *Buttons* are widgets you click on, *text boxes* are widgets you can write it, *scales* are widgets that allow you to select one value from a range of values, and *labels* are widgets that hold a string they are like text boxes only not interactive. tk treats a menu as a button when you click on it the menu opens up and displays a list of items to choose from; you select an option by clicking on one of these items. There are a few more widgets in tk, but we will see those later. Each of the widgets is represented in tk by a class; we make a widget by constructing an object of the class.

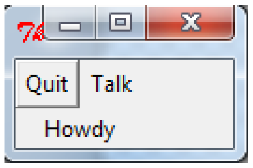
A typical interface for a program will have a main window that holds the entire interface. At the top of this window is a frame that I call a "menu bar"; this holds the menu items, buttons, scales and so forth that control the program. Below the menu bar is a canvas on which we can draw. The program that creates this sets up the widgets and then calls the mainloop() method of the Frame class. This method waits for user interactions and responds to them. Its actual code is more complex, but pseudo-code for the mainloop( ) method is

```
def mainloop(self):
    while True:
        if there-is-user-interactio():
            respond-to-user-interaction()
```

You will notice that there is no way out of this loop. The Frame class has a method quit( ) that exits you from the loop and from the program. If you want to have an animation running on the canvas, you can call an Animate( ) function for the program. This is similar to mainloop() only it does one step of the animation each time around the loop:

```
def Animate(self):
    while True:
        if there-is-user-interactio():
            respond-to-user-interaction()
        <do one step of the animation>
```

Program 9.1.1, which follows, shows a complete graphical program. This is a relatively simple program, with one button and one menu. The menu has options that allow for different strings to appear in a label widget below the control items. Here is a picture of the window this program creates:



The program is simple, but it has all of the elements of more complex GUIs. You can use it as a template for future programs. We will discuss the program in general terms here, then give details for each of the widgets it contains.

Program 9.1.1 creates a class GUI that is a subclass of tkInters standard Frame class. In the main() function it constructs and instance of this class and calls the mainloop() method of the Frame **class**. That is all it does; the rest is handled by the Python system.

Since class GUI is a subclass of Frame, the first thing the constructor for class GUI does is to call the Frame constructor: Frame. __init__ ( ). In addition to self, this has one required argument: the parent object of the Frame being constructed. All widgets need this argument for their constructors. In this case the Frame being constructed is a top-level window; it has no parent, so we pass the "unconstructed object" None. The call to the grid( ) method of the Frame class makes this window visible. All widgets have a grid( ) method. Until you call it the objects will exist but not be visible. There are several kinds of arguments you can give this method. If you call the method with no arguments it makes the widget visible and places it wherever the Python system thinks appropriate. If you call it with row and column arguments, as we do with some of the other widgets, the widget will be made visible and placed at in this location within its parents window.

Next, the constructor calls a function to create a menu system, then a function to create a quit-button; both of these are placed in row 0 of the windows grid. Finally, it creates a Label widget and places it in row 1 of the grid. A string variable called outputString is created to supply the text of this widget. We change labels by modifying this string variable.

Here is the full text of the program. We will follow this with more details about the various widgets created in the program.

```
from tkinter import *

class GUI(Frame):
   def __init__(self):
      Frame.__init__(self, None)
      self.grid()

      MenuBar = Frame(self)
      MenuBar.grid()

      self.makeTalkMenu(MenuBar)

      QuitButton=Button(MenuBar,text="Quit",command=self.quit)
      QuitButton.grid(row = 0, column = 0)

      self.outputString = StringVar(value = "")
      PrintBox = Label(MenuBar, textvar=self.outputString)
      PrintBox.grid(row = 1, column = 0, columnspan=2);

   def makeTalkMenu(self, MB):
      Talk_button = Menubutton(MB, text='Talk')
      Talk_button.menu = Menu(Talk_button)
      Talk_button['menu'] = Talk_button.menu
      Talk_button.grid(row = 0, column = 1)
      Talk_button.menu.add_command(label='Say Hi',\
                      command=self.hi)
      Talk_button.menu.add_command(label='Say Bye',\
                      command=self.bye)

   def hi(self):
      self.outputString.set("Howdy")

   def bye(self):
       self.outputString.set("Bye bye")

def main():
   window = GUI()
   window.mainloop()

main()
```

Program 9.1.1: A first GUI program

Menus

In TkInter, a "menu" consists of

9.1. A menu button, which is the top-level string the user clicks on to display the menu options.

9.2. A menu of items, one of which will be selected when the user clicks on it.

We create this in three steps:

a. Calling the Menubutton( ) constructor to make the menu button. This needs to arguments: the parent window where this button lives, and a text variable that holds the string that represents the menu.

b. Setting up the menu that holds the items. This takes two statements: one call to the Menu( ) constructor,which needs only the name of the menu button, and one assignment that links the Menu( ) constructor to the menu button

c. A series of statements that create the individual menu items. Each of these needs a string for its label,and a command function that will be called when this item is selected. This command function should take no arguments.

For our program these statements are

```
Talk_button=Menubutton(MB, text='Talk')
Talk_button.menu=Menu(Talk_button)
Talk_button['menu']=Talk_button.menu
Talk_button.grid(row = 0,column = 1)
Talk_button.menu.add_command(label='Say Hi',command=self.hi)
Talk_button.menu.add_command(label='Say Bye',command=self.bye)
```

We could have as many `Talk_button.menu.add_command( )` lines as we wish; each adds another item to the menu.

Buttons

These are created with one call to the `Button( )` constructor. This needs 3 arguments:

a. The parent widget that gives the window where the button lives

b. The text to be printed on the button

c. The "command" or *callback* function to be called when the user clicks on the button. Again, this should be a function with no arguments.

In our program we create a quit-button with

```
QuitButton = Button(MenuBar, text="Quit", command = self.quit)
```

The callback function we use, self.quit( ), is a standard method of the Frame class. We create the button in our GUI subclass of Frame, so this method is inherited by the subclass.

Labels

These are generally static holders of strings To make a simple label we use the Label( ) constructor which needs 2 arguments:

a. The parent widget that gives the window where the label lives.

b. A text string to be printed for the label.

We could make such a label with a statement like

```
Label(parent, text = "This is a string")
```

In our case we want the text of the label to be modifiable, so we go one step further. Instead of a static string as the text of the label we give a control variable that can be dynamically modified by our program. There is a separate class for each kind of control variable: IntVar, StringVar, and DoubleVar (a floating point control variable). Each has a value instance variable to hold the value of the control variable, and each class has **set**() and get( ) methods to manipulate this value. Accordingly, we create this control variable with

```
<name> = StringVar(value = "string")
```

and we assign it to the label as its textvar field.

The only other items our program needs are the callback functions to be called when the user selects the menu items. These are registered with the program when the menu items are created. Here is a typical one; it changes the label's textvar value to the string "Howdy".

```
def hi(self):
    self.outputString.set("Howdy")
```

We make this a method of the GUI class, though it could just as easily be a stand-alone function. As a method it needs self as a formal parameter; as a stand-alone function it would have no parameters.

9.2 Widgets with Values

A variety of tk widgets allow the user to enter values that are accessible to the program. In this section we focus on two of these: scales, which have sliders that allow the user to choose a value out of a range of values, and entry boxes, that allow the use to type a value. These two classes of widgets will handle most of the data entry situations you are likely to encounter.

Both of these widgets make data available to the program via *control variables*. We used a control variable in section 8.1 when we wanted to make the text of a label change according to input from our program. The control variable mechanism provides a uniform way the system can pass data between your program and its user interface. There are three classes of control variables:

- IntVar( ), which holds an integer value
- DoubleVar( ), which holds a floating point value (*double* is an old C term for a certain kind of float point value)
- StringVar( ), which holds a string.

The datum held by these objects is stored in a variable called value. There are two methods for objects of each class: get( ) returns the value stored in the object, while **set**(v) stores v as the value of the object.

For example, we we want to make an integer control variable and give it the value 8, we would use the code

```
numVertices = IntVar( )
numVertices.set(8)
```

If at some future point we wanted to retrieve the value stored in numVertices, we would say

```
n = numVertices.get( )
```

The Scale class is used to make widgets that allow the user to select one value out of a range. The constructor for this class has many defaulted parameters; here is a typical call, with the only parameters we usually need to give values to

```
Scale( parent, \
            from_ =<low end of range>,\
            to =<high end of range>, \
            orient =<HORIZONTAL or VERTICAL>,  \
            variable =<control variable> )
```

As with all of the widgets, the first argument is the parent window in which the widget lives. We save the value returned by this constructor, since we need it for the grid( ) method; the widget is not visible until we place it in its grid location.

For example,

```
numVertices = IntVar()
vScale = Scale(MenuBar,from_=1,to=100, \
      orient=HORIZONTAL, variable= numVertices)
vScale.grid(row=0, column=2)
```

You may attach a callback function to a scale, but if you do it will be continuously called as the user drags the scales slider. This may be what you want, but in many situations you want to wait until the user has finished with the scale before taking action. If this is your choice, use the code above for the Scale widget and provide a button next to the scale for the user to click after the scale is adjusted. The buttons callback can refer to the scales control variable to see the value the scale is set to.

If you do wish to use a callback function, you should assign it to the command parameter in the Scale constructor. For example,

```
numVertices = IntVar()
vScale = Scale(MenuBar, from_=1, to=100, \
      orient=HORIZONTAL, variable= numVertices, \
      command=Draw)
```

The callback function for a scale must take one argument, which will be the scales current value. For technical reasons this argument is given a string value; if you want it to be an integer you must convert it.

For example, the following might be the start of the code for the Draw function in the Scale constructor above:

```
def Draw(string_n):
        n = str(string_n)
        < code to draw a polygon with n vertices>
```

Again, a callback function is called continuously as the user drags the Scale widget. If the widget is currently set to 6 and the user drags it to 10, this function will be called for values 7, 8, 9, and 10. In a larger range the function will be called continuously, but the system might not be fast enough to keep up with the users dragging. You are guaranteed, however, that it will be called for the value the dragging stops at.

An Entry widget serves a similar purpose, but for the entry of strings. This gives the user a text box into which to type. One difference between the Entry widget and other widgets is that nothing signals when the user is finished typing. We will later see a way to use the <Return>-key as a signal, but for now we will just use a button. The user can enter a string into the Entry box, and click the button as a signal that the entry is ready.

Because there is no callback function, the code to create an Entry box is particularly simple:

```
Entry( parent, variable =<control variable> )
```

As with all of the widgets, the first argument to this constructor is the window in which it will be placed. The only other argument is the control variable that will hold the text the user enters into the box. This must be of type StringVar.

Here is code to create an entry box and a button, followed by code for the buttons callback:

```
global numberOfVertices
numberOfVertices = StringVar()
t = Entry(MenuBar, textvar = numberOfVertices)
t.grid(row=0, column=1)

DrawButton =  Button(MenuBar, text = "Draw", \
     command = self.Draw)
DrawButton.grid(row=0, column = 2 )

def Draw(self)
     n = int(numberOfVertices.get())
     <code to draw a polygon with n vertices>
```

The following program illustrates these ideas. This is a complete program for drawing a number of circles given by the user. The user has two choices for input: either a Scale widget or an Entry box. Each is accompanied by a button whose callback function handles the actual drawing. Dont worry about the code for drawing circles; we will discuss that in section 8.3.

```
from tkinter import *
from random import *

class GUI(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.grid()

        MenuBar = Frame(self)
        MenuBar.grid(row = 0, column = 0, sticky=W)

        QuitButton=Button(MenuBar,text="Quit",command=self.quit)
        QuitButton.grid(row = 0, column = 0)

        self.numCircles = IntVar()
        s = Scale(MenuBar, from_ = 1, to=50, \
          orient=HORIZONTAL, variable = self.numCircles)
        s.grid(row=0, column = 1)
        lab = Label(MenuBar, text="Number of Circles" )
        lab.grid(row=1, column=1)

        sButton = Button(MenuBar, text="Draw",  \
          command = self.DrawCircles)
        sButton.grid(row = 0, column = 2)

        self.numCircles2 = StringVar()
        e = Entry(MenuBar, textvar = self.numCircles2)
        e.grid(row = 0, column = 3)
        lab2 = Label(MenuBar, text = "Number of Circles" )
        lab2.grid(row = 1, column = 3)

        sButton2 = Button(MenuBar, text = "Draw", \
          command = self.DrawCircles2)
        sButton2.grid(row = 0, column = 4)

        global canvas
        canvas = Canvas(self, width=500, height=500, \
          background="white")
        canvas.grid(row=1, column=0)
```

Program 9.2.1: Circle Drawer, first part

```
    def DrawCircles(self):
        canvas.delete("all")
        n = self.numCircles.get()
        for x in range(0, n):
            RandomCircle()

    def DrawCircles2(self):
        canvas.delete("all")
        n = int( self.numCircles2.get() )
        for x in range(0, n):
            RandomCircle()

class Circle:
    def __init__(self, x, y, radius, color):
        self.my_shape = canvas.create_oval(x-radius, y-radius,
          x+radius, y+radius, fill = color)

def RandomCircle():
    x = randint(1, 500)
    y = randint(1, 500)
    radius = randint(5, 50)
    colors = ["red", "green", "blue", "yellow", "purple"]
    color = colors[randint(0, 4)]
    Circle(x, y, radius, color)

def main():
    window = GUI()
    window.mainloop()

main()
```

Program 9.2.1: Circle Drawer, conclusion

9.3 Drawing

Tk provides a rich and easy-to-use set of primitive drawing operations. To use them we need to create a canvas widget, and then draw upon it. Since all of the drawing operations need access to the canvas widget, it is easiest to make this a global variable. The code for constructing a canvas is simple:

```
global canvas
canvas=Canvas(self ,width=500,height=500,background="white")
canvas.grid(row=1, column=0)
```

The canvas is generally a child of the main GUI window. The only objects in this window typically are the MenuBar and the canvas. I usually put the MenuBar in row 0 and the canvas in row 1, below the MenuBar.

Canvases have a natural (x, y) coordinate system. The x-coordinate measures the distance in pixels from the left edge of the canvas; the y-coordinate measures the distance of the pixel from the top of the canvas. As we move to the right the x-coordinate increases; as we move DOWN the y-coordinate increases. This surprises some people who remember y-coordinates increasing as you move up in the traditional Cartesian coordinates in analytic geometry.

Canvas objects have methods for drawing various shapes. The most common shapes are rectangles (which include squares), ovals (including circles) and polygons. For rectangles you must give first the coordinates of the upper-left corner of the rectangle, then the coordinate of the lower-right corner. The default drawing color is black, but you can change this to any other color by giving the fill parameter of the drawing command the name of the color to draw.

For example, the following function draws a rectangle with upper-left corner at (x0, y0) and lower-right corner at (x1, y1):

```
def Rectangle(x0, y0, x1, y1, color):
    canvas.create_rectangle(x0, y0, x1, y1, fill=color)
```

We might call this with:

```
Rectangle(300, 300, 400, 450,    red    )
```

There is a corresponding canvas. create_oval ( ) method that draws ellipses. This takes the same arguments as canvas. create_rectangle ( ): the coordinates of the upper-left and lower-right corners, then fill −color. Since we are used to thinking of a circle as points a fixed distance from the center of the circle, we can write a function that takes the center and radius of the circle and builds the rectangle that surrounds the circle. Here is such a function:

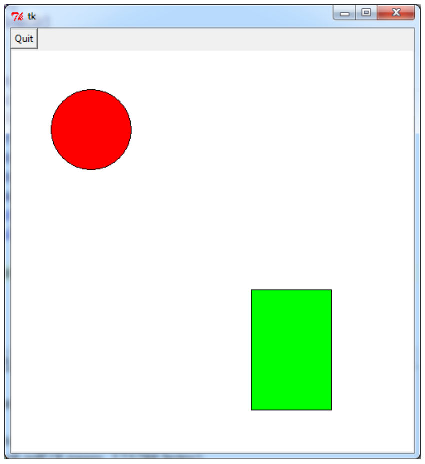
```
def Circle(x, y, radius, color):
    canvas.create_oval(x-radius, y-radius, \
                       x+radius, y+radius, fill=color)
```

Altogether, the following program will create a canvas and draw a circle and square on it:

```
from tkinter import *

class GUI(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.grid()

        MenuBar = Frame(self)
        MenuBar.grid(row = 0, column = 0, sticky=W)

        QuitButton=Button(MenuBar,text="Quit",command=self.quit)
        QuitButton.grid(row = 0, column = 0)

        global canvas
        canvas=Canvas(self, width=500, height=500,\
          background="white")
        canvas.grid(row=1, column=0)


def Circle(x, y, radius, color):
    canvas.create_oval(x-radius, y-radius, \
          x+radius, y+radius, fill=color)
    canvas.update()

def Rectangle(x0, y0, x1, y1, color):
    canvas.create_rectangle(x0, y0, x1, y1, fill=color)

def main():
    window = GUI()
    Circle(100,100, 50, "red" )
    Rectangle(300, 300, 400, 450, "green" )
    window.mainloop()

main()
```

Program 9.3.1: Simple Drawing Program

Here is a picture of the canvas this creates:



The canvas. create_rectangle ( ) and similar drawing methods return a value that can be used to manipulate the shape has been drawn. In particular, if we draw a 100x100 green rectangle with

```
r =canvas.create_rectangle(100,150,200,250,fill="green")
```

then we can move this rectangle 10 units horizontally and 30 vertically with

```
canvas.move(r, 10, 30)
```

We can change its color to red with

```
canvas.itemconfigure(r, fill="red")
```

Since this allows us to give functionality to the shapes, it makes sense to represent each category of shapes by a class. In the following code we make classes Rectangle, Square, Oval, Circle and Polygon. To avoid repeating the same code for each class, we make a top-level Shape class that contains all of the uniform code. Each of our basic shapes is a subclass of Shape, and each is constructed with only the code that differs from the uniform Shape code.

```
from tkinter import *

class GUI(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.grid()

        MenuBar = Frame(self)
        MenuBar.grid(row = 0, column = 0, sticky=W)

        QuitButton=Button(MenuBar,text="Quit",command=self.quit)
        QuitButton.grid(row = 0, column = 0)

        global canvas
        canvas = Canvas(self, width=500, height=500, \
          background="white")
        canvas.grid(row=1, column=0)

class Shape:
    def __init__(self, vertices, color):
        self.color = color
        self.vertices = vertices
        self.my_shape = None

    def Moveto(self, a, b):
        # This moves the shape to point (a,b)
        [x, y] = self.vertices[0]
        d0 = a-x
        d1 = b-y
        self.Moveby(d0, d1)

    def Moveby(self, a, b):
        # This moves the shape a units horizontally
        # and b units vertically.
        canvas.move(self.my_shape, a, b)
        canvas.update()
        for v in self.vertices:
            v[0] = v[0] + a
            v[1] = v[1] + b
```

Program 9.3.2: Shape Classes

```
def ChangeColor(self, color):
        # This changes the shape's color.
        # Possible colors include "white", "black",
        # "red", "green", "blue", "yellow", "cyan",
        # "magenta", "light green", etc.
        canvas.itemconfigure(self.my_shape, fill = color )
        canvas.update()
        self.color = color

    def delete(self):
        canvas.delete(self.my_shape)

class Rectangle(Shape):
    def __init__(self, x, y, x1, y1, color):
        # This creates a square with corners (x, y) and (x1, y1)
        Shape.__init__(self, [[x, y], [x1, y1]], color)
        self.my_shape = canvas.create_rectangle(x, y, x1, y1, \
          fill = color)

    def pos(self):
        return (self.vertices[0][0], self.vertices[0][1])

class Oval(Shape):
    def __init__(self, x, y, hrad, vrad, color):
        # This creates an oval centered at (x, y)
        # with horizontal radius hrad and
        # vertical radius vrad
        Shape.__init__(self, [ [x-hrad, y-vrad], \
          [x+hrad, y+vrad] ], color)
        v0 = self.vertices[0]
        v1 = self.vertices[1]
        self.my_shape = canvas.create_oval(v0[0], v0[1], \
          v1[0], v1[1], fill = color)

    def pos(self):
        v0 = self.vertices[0]
        v1 = self.vertices[1]
        return ( (v0[0]+v1[0])/2, (v0[1]+v1[1])/2)

class Square(Rectangle):
    def __init__(self, x, y, side, color):
        # This creates a square with upper left corner at (x, y)
        # and length side
        Rectangle.__init__(self, x, y, x+side, y+side, color)
```

Program 9.3.2: Shape Classes, continued

```
class Circle(Oval):
    # This creates a circle centered at (x, y)
    # with the given radius
    def __init__(self, x, y, radius, color):
        Oval.__init__(self, x, y, radius, radius, color)

class Polygon(Shape):
# To create a polygon make a list of its vertices
# (each vertext itself is a list [x, y];
# you need a list of these)
# and call canvas.create_polygon().
# This returns a reference to the polygon
# that you will need if you ever want to
# change or delete it.
    def __init__(self,vertices, color):
        Shape.__init__(self, vertices, color)
        self.my_shape=canvas.create_polygon(vertices,fill=color)
        canvas.update()
        self.pos = (vertices[0][0], vertices[0][1])

def main():
    window = GUI()
    Circle(250,250, 100, "red" )
    window.mainloop()

main()
```

Program 9.3.2: Shape Classes, continued

9.4 Animation

It is tempting to try to animate objects by giving each object a loop that describes its motion, as in

```
def randomCircle(self):
        c = Circle(250, 50, 30, "red")
        while True:
            c.Moveby(0, -0.1)
            (x, y)= c.pos()
            if y < 0:
                break
```

This suffers from two problems. First, it isnt interruptable; if we try to exit from the program, perhaps via a Quit button, the program will wait until this animation is finished before exiting. Even worse, if we have several objects with code like this only one will move at a time, continuing its motion until it is finished before the next object starts to move.

To make our animations more effective, we need three simple components:

a. We will give every object a NextStep( ) method that performs one small step of the objects entire motion.

b. We will store all of the moving objects in an AnimationList. One step of the global animation will consist of running through the AnimationList and telling each object in it to take its NextStep( ).

c. Instead of looping through this process, which would not be interruptable, we will make use of a special method of the Canvas class:

   ```
   canvas.after(<time to pause>, <function>)
   ```

   This method pauses for a small amount of time (measured in milliseconds), then calls the given function.

This does everything we want. If our individual NextStep() methods arent too complex and if we dont have too many objects in the animation list, this can make the motions look continuous and smooth. By adjusting the first argument of the canvas.after( ) call we can fine-tune the speed of the animation. All of our widgets that need user control will still be functional; the system will respond to them after each pass through the AnimationList.

For a first example, we have a program with a drawing menu with two options: *Square* and *Circle*. When the user selects one of these options a random shape of that type is drawn and added to the AnimationList. The NextStep function is particularly simple: it moves the shape downward a small amount. When the object gets to the bottom of the canvas it is deleted. Here is the NextStep( ) method for each object:

```
def NextStep(self):
        self.Moveby(0, 2)
```

and here is the Animate( ) function:

```
def Animate():
    for object in AnimationList:
        object.NextStep()
        (x, y) = object.pos()
        if y > 500:
            i = AnimationList.index(object)
            del AnimationList[i]
            object.delete()
    canvas.after( 50, Animate )
```

What follows is the complete program for this, omitting the definition of the drawing classes that we saw in the last section:

```
from tkinter import *
from time import *
from random import *

class GUI(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.grid()

        MenuBar = Frame(self)
        MenuBar.grid(row = 0, column = 0, sticky=W)

        self.MakeDrawingMenu(MenuBar)

        QuitButton=Button(MenuBar,text="Quit",command=self.quit)
        QuitButton.grid(row = 0, column = 0)

        global canvas
        canvas=Canvas(self, width=500, height=500, \
          background="white")
        canvas.grid(row=1, column=0)

        global CircleList
        CircleList = []
```

Program 9.4.1: Squares and Circles

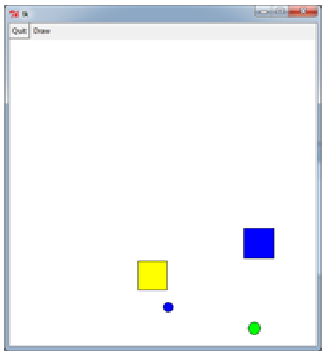
```
def MakeDrawingMenu(self, MB):
    Draw_button = Menubutton(MB, text='Draw')
    Draw_button.menu = Menu(Draw_button)
    Draw_button['menu'] = Draw_button.menu
    Draw_button.grid(row = 0, column = 1)
    Draw_button.menu.add_command(label='Square', \
      command=self.randomSquare)
    Draw_button.menu.add_command(label='Circle', \
      command=self.randomCircle)

def randomSquare(self):
    colors=["red","green","blue","yellow","orange","brown"]
    x = randint(1, 500)
    y = randint(1, 500)
    size = randint(5, 50)
    color = colors[ randint(0, len(colors)-1) ]
    s = Square( x, y, size, color)
    AnimationList.append(s)

def randomCircle(self):
    colors=["red","green","blue","yellow","orange","brown"]
    x = randint(1, 500)
    y = randint(1, 500)
    size = randint(3, 25)
    color = colors[ randint(0, len(colors)-1) ]
    c = Circle( x, y, size, color)
    AnimationList.append(c)
```

Program 9.4.1: Squares and Circles, continued

```
def Idle():
    for object in AnimationList:
        object.NextStep()
        (x, y) = object.pos()
        if y > 500:
            i = AnimationList.index(object)
            del AnimationList[i]
            object.delete()
    canvas.after( 50, Idle )

def main():
    global AnimationList
    AnimationList = []
    window = GUI()
    Idle()
    window.mainloop()

main()
```

Program 9.4.1: Squares and Circles, concluded

Here is a picture of this programs window as it runs:



The next example has particles bouncing off walls indicated by the edges of the canvas. This time we make a class `Particle` to represent the moving bodies. `Particle` is just a subclass of `Circle`, with a few added properties. To make the particle move we maintain a vector <`x_dir`, `y_dir`> that holds a change amount for the x- and y-directions. At each step of the animation we add `x_dir` to the particles x-coordinate and `y_dir` to its y-coordinate. This moves the particle in a straight, not necessarily horizontal or vertical, line. To bounce off the vertical walls, on which the x-coordinate is a constant, we maintain the same value of `y_dir` and change `x_dir` to `x_dir` . Similarly, to bounce off the horizontal walls, where the y-coordinate is constant, we change `y_dir` to `y_dir` . Here is the resulting `Particle` class:

```
class Particle(Circle):
    def __init__(self):
        x = randint(0, 500)
        y = randint(0, 500)
        self.radius = 10
        colors = ["red","green","blue","yellow",\
          "purple","orange","magenta"]
        color = colors[ randint(0, len(colors)-1) ]
        Circle.__init__(self, x, y, self.radius, color)
        self.x_dir = randint(-3, 3)
        self.y_dir = randint(-3, 3)

    def NextStep(self):
        self.Moveby(self.x_dir, self.y_dir)
        (x, y) = self.pos()
        if x < 5 or x > 495:
            self.x_dir = -self.x_dir
        if y < 5 or y > 495:
            self.y_dir = -self.y_dir
```

Here is the complete program.

```
from tkinter import *
from random import *

class GUI(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.grid()

        MenuBar = Frame(self)
        MenuBar.grid(row = 0, column = 0, sticky=W)

        QuitButton = Button(MenuBar, text="Quit", \
          command = self.quit)
        QuitButton.grid(row = 0, column = 0)

        MoreButton = Button(MenuBar, text="More", \
          command = self.MoreParticles)
        MoreButton.grid(row = 0, column = 1)

        PauseButton = Button(MenuBar, text="Pause", \
          command = Pause)
        PauseButton.grid(row = 0, column = 2)

        global canvas
        canvas = Canvas(self, width=500, height=500, \
          background="white")
        canvas.grid(row=1, column=0)

    def MoreParticles(self):
        p = Particle()
        AnimationList.append(p)
```

Program 9.4.2: Bouncing Off the Walls

```
class Shape:
    def __init__(self, vertices, color):
        self.color = color
        self.vertices = vertices
        self.my_shape = None

    def Moveto(self, a, b):
        # This moves the shape to point (a,b)
        (x, y) = self.pos()
        d0 = a-x
        d1 = b-y
        self.Moveby(d0, d1)

    def Moveby(self, a, b):
        # This moves the shape a units horizontally
        # and b units vertically
        canvas.move(self.my_shape, a, b)
        canvas.update()
        for v in self.vertices:
            v[0] = v[0] + a
            v[1] = v[1] + b

    def ChangeColor(self, color):
        # This changes the shape's color.  Possible
        # colors include "white", "black", "red", etc.
        canvas.itemconfigure(self.my_shape, fill = color )
        canvas.update()
        self.color = color

    def delete(self):
        canvas.delete(self.my_shape)

    def pos(self):
        return (self.vertices[0][0], self.vertices[0][1])

class Oval(Shape):
    def __init__(self, x, y, hrad, vrad, color):
        # This creates an oval centered at (x, y) with \
        # horizontal radius hrad and vertical radius vrad
        Shape.__init__(self,[[x-hrad,y-vrad],[x+hrad,y+vrad]],\
          color)
        v0 = self.vertices[0]
        v1 = self.vertices[1]
        self.my_shape = canvas.create_oval(v0[0], v0[1], \
          v1[0], v1[1], fill = color)

    def pos(self):
        v0 = self.vertices[0]
        v1 = self.vertices[1]
        return ( (v0[0]+v1[0])/2, (v0[1]+v1[1])/2)
```

Program 9.4.2: Bouncing Off the Walls, continued

```
class Circle(Oval):
    # Creates a circle centered at (x, y) with the given radius
    def __init__(self, x, y, radius, color):
        Oval.__init__(self, x, y, radius, radius, color)

class Particle(Circle):
    def __init__(self):
        x = randint(0, 500)
        y = randint(0, 500)
        self.radius = 10
        colors = [ "red", "green", "blue", "yellow", \
          "purple", "orange", "magenta"]
        color = colors[ randint(0, len(colors)-1) ]
        Circle.__init__(self, x, y, self.radius, color)
        self.x_dir = randint(-3, 3)+ 0.5
        self.y_dir = randint(-3, 3)+ 0.5

    def NextStep(self):
        self.Moveby(self.x_dir, self.y_dir)
        (x, y) = self.pos()
        if x < 5 or x > 495:
            self.x_dir = -self.x_dir
        if y < 5 or y > 495:
            self.y_dir = -self.y_dir

def Animate():
    if pause:
        return
    for x in AnimationList:
        x.NextStep()
    canvas.after( 1, Animate )

def Pause():
    global pause
    if pause:
        pause = False
        Animate()
    else:
        pause = True

def main():
    global AnimationList
    AnimationList = []
    global pause
    pause = False

    window = GUI()
    Animate()
    window.mainloop()

main()
```

Program 9.4.2: Bouncing Off the Walls, concluded

9.5 Events

An *event* is any user interaction with a program. This might involve pressing a key, clicking the mouse button, and moving or dragging the mouse. Every time the user initiates any of these actions the system generates an event signal. Some widgets are designed to react to these signals. A button, for example, waits for a mouse button to be clicked on it; when this happens the buttons callback function is called. Any widget can be programmed to listen for specific events and then call designated functions when these events occur. In this section we will see how this *event programming* works.

The first step in getting a widget to respond to event signals is to tell the widget to listen for signals. If w is a widget, then

```
w.focus_set( )
```

tells the widget to pay attention to signals.

The next step is to register a function as the callback for a specific event. Again, if w is the widget this is done with

```
Widget.bind(w, <event descriptor>, <function>)
```

Here Widget is the superclass for all widget classes. The <event descriptor> is a string, such as "Button−1", which refers to the left mouse button (the only mouse button on a Mac). Here is a table of event descriptors:

Descriptor	**Event**
"$<KeyPress-a>$"	The users presses the "a" key
"$<KeyPress-A>$"	The user presses the "A" key
	The other letters and digits are similar.
"$<KeyPress-space>$"	The user presses the space bar
"$<KeyPress-Return>$"	The user presses the Return (or Enter) key
"$<KeyPress-Up>$"	The user presses the up-arrow
"$<KeyPress-Down>$"	The user presses the down-arrow
"$<KeyPress-Left>$"	The user presses the left-arrow
"$<KeyPress-Right>$"	The user presses the right-arrow
"$<Any-KeyPress>$"	The user presses any key.
"$<Button-1>$"	The user clicks the left mouse button (or only mouse button)
"$<Button-2>$"	The user clicks the middle mouse button
"$<Button-3>$"	The user clicks the right mouse button
"$<B1-Motion>$"	This represents a dragging event with the left mouse button

The callback function for any event binding should take one argument, which is an object representing the event. We usually call this argument "event". The specifics of the object will vary according to the type of event that has occurred. With mouse events event.x and event.y are the coordinates of the mouse at the time the event occurred. With keyboard events event.keysym is a string representing the key that was pressed. For letter keys this is the letter as a string: "A", or "a", "space", "Return" and so forth.

Here, for example, is the code needed to empower the "q" key to exit the program:

```
class GUI(Frame):
        ..
        global canvas
        canvas = Canvas(self, width=500, height=500, background="white
        canvas.grid(row=1, column=0)
        canvas.focus_set()
        Widget.bind(canvas, "<KeyPress-q>", self.quitter)

    def quitter(self, event):
        self.quit()
```

This binds the "<KeyPress−q>" description to a function that calls the quit() method of the Frame class. Note that the callback for the binding must take an event argument,so we cant directly connect to the self.quit() method.

If you want to interact with objects that have been drawn on the canvas, it is important to be able to determine which objects are under the cursor when the mouse button is clicked. While it would be possible to program this yourself, the tk system provides some hooks that make this easy. When shapes are created it is possible to give them "tags", which are strings. The canvas.getttags(**object**) method returns a list of all of the tags that have been given to a specific object. The canvas.find_withtag(tag) method returns a list of all of the objects that have the given tag. As the mouse moves around the canvas, the system gives a tag "current" to the topmost (most recently drawn) object that the mouse is currently over. So we can find the object the mouse is currently over by finding the shape in the first entry of canvas.find_withtag("current")

Here is how we might use all of this to create a simple circle-drawing program. We set up the interface to have a Draw button for circles and a set of radio buttons to select the current drawing color. To allow for dragging circles after they have been drawn we bind the left mouse button to a function that selects the circle into which we have clicked, and we bind the dragging operation with the left mouse button to a function that moves the current circle. We also bind the "d" key on the keyboard to a function that deletes the circle the mouse is currently over. This much of our interface definition is as follows:

```
class GUI(Frame):
        .......
        global ColorChoice
        ColorChoice = StringVar()
        ColorChoice.set("red")
        r1=Radiobutton(MenuBar,value = "red",text="red",\
           variable=ColorChoice)
        r1.select()
        r1.grid(row=0, column = 1)
```

```
            r2=Radiobutton(MenuBar,value="green",text="green",\
               variable = ColorChoice)
            r2.grid(row=0, column = 2)
            r3=Radiobutton(MenuBar,value="blue",text = "blue",\
               variable = ColorChoice)
            r3.grid(row=0, column = 3)
            r4=Radiobutton(MenuBar,value="cyan",text="cyan",\
               variable = ColorChoice)
            r4.grid(row=1, column = 1)
            r5=Radiobutton(MenuBar,value="magenta",text="magenta",\
               variable = ColorChoice)
            r5.grid(row=1, column = 2)
            r6=Radiobutton(MenuBar,value="yellow",text="yellow",\
               variable = ColorChoice)
            r6.grid(row=1, column = 3)

            CircleButton = Button(MenuBar, text="Circle", \
              command = self.DrawCircle)
            CircleButton.grid(row = 0, column = 4)

            global canvas
            canvas = Canvas(self, width=500, height=500, \
              background="white")
            canvas.grid(row=1, column=0)
            canvas.focus_set()
            Widget.bind(canvas, "<Button-1>", self.SelectCircle)
            Widget.bind(canvas, "<B1-Motion>", self.Drag)
            Widget.bind(canvas, "<KeyPress-d>", self.DeleteCircle)
```

We draw circles (after the Draw button is clicked) at a random position using the ColorChoice selected with the radio buttons:

```
    def DrawCircle(self):
            x = randint(0, 500)
            y = randint(0, 500)
            color = ColorChoice.get()
            c = Circle(x, y, 20, color)
            CircleList.append(c)
```

We move a circle by clicking inside it, then dragging the mouse with the button depressed. The initial click calls the function

```
    def SelectCircle(self, event):
        global current
        L = canvas.find_withtag("current")
        for c in CircleList:
            if c.my_shape in L:
                current = c
                break
        global last_pos
        last_pos = (event.x, event.y)
```

This sets a global variable current, which is the object in our Circle class that contains the "current" tag. It also sets the initial value of a global variable last_pos, that we use in the dragging operation.

The mouse-drag callback looks at the current mouse position and the current circle (set in the SelectCircle ( ) function) by the difference between the current position and the last position. It would be slightly simpler to move the current circle to the current mouse position, but that has the unfortunate effect of making the circle jump so that its center is at the mouse position when you start dragging. This unexplained jump is awkward for users, so instead we keep track of the mouses relative motion and use that as the basis for the circle movement. Of course, each time the Drag( ) function is called we need to reset the value of our global variable last_pos.

```
    def Drag(self, event):
        global last_pos
        current.Moveby(event.x-last_pos[0], \
          event.y-last_pos[1])
        last_pos = (event.x, event.y)
```

Finally, we delete a circle in two steps. We need to both erase it from the screen and remove it from our CircleList. Erasing it means finding its shape and calling the canvas.delete( ) method on this shape. To remove it from the CircleList, we must first find the Circle object that contains this shape, then the index of this object in our CircleList, and finally delete this element from the list. Altogether we get the following code:

```
    def DeleteCircle(self, event):
        print "deleting"
        shape = canvas.find_withtag("current")
        if shape == ():
            return
        canvas.delete(shape[0])
        for c in CircleList:
            if c.my_shape == shape[0]:
                i = CircleList.index(c)
                del CircleList[i]
```

Putting all of these together, here is the full program:

```
from tkinter import *
from random import *

class GUI(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.grid()

        MenuBar = Frame(self)
        MenuBar.grid(row = 0, column = 0, sticky=W)

        QuitButton=Button(MenuBar, text="Quit", \
          command = self.quit)
        QuitButton.grid(row = 0, column = 0)

        global ColorChoice
        ColorChoice = StringVar()
        ColorChoice.set("red")
        r1=Radiobutton(MenuBar,value = "red",text="red",\
           variable=ColorChoice)
        r1.select()
        r1.grid(row=0, column = 1)

        r2=Radiobutton(MenuBar,value="green",text="green",\
           variable = ColorChoice)
        r2.grid(row=0, column = 2)
        r3=Radiobutton(MenuBar,value="blue",text = "blue",\
           variable = ColorChoice)
        r3.grid(row=0, column = 3)
        r4=Radiobutton(MenuBar,value="cyan",text="cyan",\
           variable = ColorChoice)
        r4.grid(row=1, column = 1)
        r5=Radiobutton(MenuBar,value="magenta",text="magenta",\
           variable = ColorChoice)
        r5.grid(row=1, column = 2)
        r6=Radiobutton(MenuBar,value="yellow",text="yellow",\
           variable = ColorChoice)
        r6.grid(row=1, column = 3)
```

Program 9.5.1: Drawing With Circles

```
        CircleButton = Button(MenuBar, text="Circle", \
          command = self.DrawCircle)
        CircleButton.grid(row = 0, column = 4)

        global canvas
        canvas = Canvas(self, width=500, height=500, \
         background="white")
        canvas.grid(row=1, column=0)
        canvas.focus_set()
        Widget.bind(canvas, "<Button-1>", self.SelectCircle)
        Widget.bind(canvas, "<B1-Motion>", self.Drag)
        Widget.bind(canvas, "<KeyPress-d>", self.DeleteCircle)

        global CircleList
        CircleList = []

    def DrawCircle(self):
        x = randint(0, 500)
        y = randint(0, 500)
        color = ColorChoice.get()
        c = Circle(x, y, 20, color)
        CircleList.append(c)

    def SelectCircle(self, event):
        global current
        L = canvas.find_withtag("current")
        for c in CircleList:
            if c.my_shape in L:
                current = c
                break
        global last_pos
        last_pos = (event.x, event.y)

    def Drag(self, event):
        global last_pos
        current.Moveby(event.x-last_pos[0], event.y-last_pos[1])
        last_pos = (event.x, event.y)
```

Program 9.5.1: Drawing With Circles, continued

```
    def DeleteCircle(self, event):
        print "deleting"
        shape = canvas.find_withtag("current")
        if shape == ():
            return
        canvas.delete(shape[0])
        for c in CircleList:
            if c.my_shape == shape[0]:
                i = CircleList.index(c)
                del CircleList[i]

class Shape:
    def __init__(self, vertices, color):
        self.color = color
        self.vertices = vertices
        self.my_shape = None

    def pos(self):
        return (self.vertices[0][0], self.vertices[0][1])

    def Moveto(self, a, b):
        # This moves the shape to point (a,b)
        (x, y) = self.pos()
        d0 = a-x
        d1 = b-y
        self.Moveby(d0, d1)

    def Moveby(self, a, b):
        # This moves the shape a units horizontally
        # and b units vertically
        canvas.move(self.my_shape, a, b)
        canvas.update()
        for v in self.vertices:
            v[0] = v[0] + a
            v[1] = v[1] + b

    def ChangeColor(self, color):
        # This changes the shape's color.
Possible colors include
        # "white", "black", "red", "green", etc.
        canvas.itemconfigure(self.my_shape, fill = color )
        canvas.update()
        self.color = color
```

Program 9.5.1: Drawing With Circles, continued

```
class Oval(Shape):
    def __init__(self, x, y, hrad, vrad, color):
        # This creates an oval centered at (x, y) with
        # horizontal radius hrad and vertical radius vrad
        Shape.__init__(self, [[x-hrad, y-vrad], \
          [x+hrad, y+vrad]], color)
        v0 = self.vertices[0]
        v1 = self.vertices[1]
        self.my_shape = canvas.create_oval(v0[0], v0[1], \
          v1[0], v1[1], fill = color)

    def pos(self):
        v0 = self.vertices[0]
        v1 = self.vertices[1]
        return ( (v0[0]+v1[0])/2, (v0[1]+v1[1])/2 )

class Circle(Oval):
    # Creates a circle centered at (x, y) with the given radius
    def __init__(self, x, y, radius, color):
        Oval.__init__(self, x, y, radius, radius, color)

def main():
    window = GUI()
    window.mainloop()

main()
```

Program 9.5.1: Drawing With Circles, conclusion