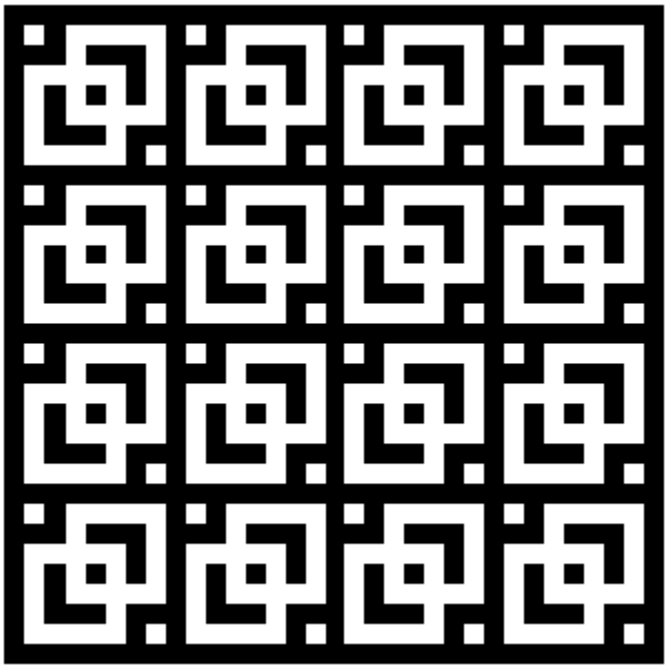
A Practical Introduction to Python Programming



Brian Heinold

Department of Mathematics and Computer Science
Mount St. Mary's University

Contents

Preface

My goal here is for something that is partly a tutorial and partly a reference book. I like how tutorials get you up and running quickly, but they can often be a little wordy and disorganized. Reference books contain a lot of good information, but they are often too terse, and they don't often give you a sense of what is important. My aim here is for something in the spirit of a tutorial but still useful as a reference. I summarize information in tables and give a lot of short example programs. I also like to jump right into things and fill in background information as I go, rather than covering the background material first.

This book started out as about 30 pages of notes for students in my introductory programming class at Mount St. Mary's University. Most of these students have no prior programming experience, and that has affected my approach. I leave out a lot of technical details and sometimes I oversimplify things. Some of these details are filled in later in the book, though other details are never filled in. But this book is not designed to cover everything, and I recommend reading other books and the Python documentation to fill in the gaps.

The style of programming in this book is geared towards the kinds of programming things I like to do—short programs, often of a mathematical nature, small utilities to make my life easier, and small computer games. In fact, the things I cover in the book are the things that I have found most useful or interesting in my programming experience, and this book serves partly to document those things for myself. This book is not designed as a thorough preparation for a career in software engineering. Interested readers should progress from this book to a book that has more on computer science and the design and organization of large programs.

In terms of structuring a course around this book or learning on your own, the basis is most of Part I. The first four chapters are critically important. Chapter 5 is useful, but not all of it is critical. Chapter 6 (strings) should be done before Chapter 7 (lists). Chapter 8 contains some more advanced list topics. Much of this can be skipped, though it is all interesting and useful. In particular, that chapter covers list comprehensions, which I use extensively later in the book. While you can get away without using list comprehensions, they provide an elegant and efficient way of doing things. Chapter 9 (while loops) is important. Chapter 10 contains a bunch of miscellaneous topics, all of which are useful, but many can be skipped if need be. The final four chapters of Part I are about dictionaries, text files, functions, and object-oriented programming.

Part II is about graphics, mostly GUI programming with Tkinter. You can very quickly write some nice programs using Tkinter. For instance, Section 15.7 presents a 20-line working (though not

perfect) tic-tac-toe game. The final chapter of Part II covers a bit about the Python Imaging Library.

Part III contains a lot of the fun and interesting things you can do with Python. If you are structuring a one-semester course around this book, you might want to pick a few topics in Part III to go over. This part of the book could also serve as a reference or as a place for interested and motivated students to learn more. All of the topics in this part of the book are things that I have found useful at one point or another.

Though this book was designed to be used in an introductory programming course, it is also useful for those with prior programming experience looking to learn Python. If you are one of those people, you should be able to breeze through the first several chapters. You should find Part II to be a concise, but not superficial, treatment on GUI programming. Part III contains information on the features of Python that allow you to accomplish big things with surprisingly little code.

In preparing this book the Python documentation at www.python.org was indispensable. This book was composed entirely in LaTeX. There are a number of LaTeXpackages, particularly `listings` and `hyperref`, that were particulary helpful. LaTeXcode from http://blog.miliauskas.lt/ helped me get the `listings` package to nicely highlight the Python code.

Listings for the longer programs are available at https://www.brianheinold.net/python/. Text files used in the text and exercises are available at https://www.brianheinold.net/python/textfiles.html I don't have solutions available to the exercises here, but there is a separate set of a few hundred exercises and solutions at https://www.brianheinold.net/python/worked_exercises.html.

Please send comments, corrections, and suggestions to `heinold@msmary.edu`.

Last updated March 18, 2022.

Part I

Basics

Chapter 1

Getting Started

This chapter will get you up and running with Python, from downloading it to writing simple programs.

1.1 Installing Python

Go to www.python.org and download the latest version of Python (version 3.5 as of this writing). It should be painless to install. If you have a Mac or Linux, you may already have Python on your computer, though it may be an older version. If it is version 2.7 or earlier, then you should install the latest version, as many of the programs in this book will not work correctly on older versions.

1.2 IDLE

IDLE is a simple integrated development environment (IDE) that comes with Python. It's a program that allows you to type in your programs and run them. There are other IDEs for Python, but for now I would suggest sticking with IDLE as it is simple to use. You can find IDLE in the Python 3.4 folder on your computer.

When you first start IDLE, it starts up in the shell, which is an interactive window where you can type in Python code and see the output in the same window. I often use the shell in place of my calculator or to try out small pieces of code. But most of the time you will want to open up a new window and type the program in there.

Note At least on Windows, if you click on a Python file on your desktop, your system will run the program, but not show the code, which is probably not what you want. Instead, if you right-click on the file, there should be an option called `Edit with Idle`. To edit an existing Python file,

either do that or start up IDLE and open the file through the `File` menu.

Keyboard shortcuts The following keystrokes work in IDLE and can really speed up your work.

Keystroke	Result
`CTRL+C`	Copy selected text
`CTRL+X`	Cut selected text
`CTRL+V`	Paste
`CTRL+Z`	Undo the last keystroke or group of keystrokes
`CTRL+SHIFT+Z`	Redo the last keystroke or group of keystrokes
`F5`	Run module

1.3 A first program

Start IDLE and open up a new window (choose `New Window` under the `File` Menu). Type in the following program.

```
temp = eval(input('Enter a temperature in Celsius: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

Then, under the `Run` menu, choose `Run Module` (or press F5). IDLE will ask you to save the file, and you should do so. Be sure to append `.py` to the filename as IDLE will not automatically append it. This will tell IDLE to use colors to make your program easier to read.

Once you've saved the program, it will run in the shell window. The program will ask you for a temperature. Type in 20 and press enter. The program's output looks something like this:

```
Enter a temperature in Celsius: 20
In Fahrenheit, that is 68.0
```

Let's examine how the program does what it does. The first line asks the user to enter a temperature. The **`input`** function's job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a *string* and it will appear to the program's user exactly as it appears in the code itself. The **`eval`** function is something we use here, but it won't be clear exactly why until later. So for now, just remember that we use it when we're getting numerical input.

We need to give a name to the value that the user enters so that the program can remember it and use it in the second line. The name we use is `temp` and we use the equals sign to assign the user's value to `temp`.

The second line uses the **`print`** function to print out the conversion. The part in quotes is another string and will appear to your program's user exactly as it appears in quotes here. The second

argument to the **print** function is the calculation. Python will do the calculation and print out the numerical result.

This program may seem too short and simple to be of much use, but there are many websites that have little utilities that do similar conversions, and their code is not much more complicated than the code here.

A second program Here is a program that computes the average of two numbers that the user enters:

```
num1 = eval(input('Enter the first number: '))
num2 = eval(input('Enter the second number: '))
print('The average of the numbers you entered is', (num1+num2)/2)
```

For this program we need to get two numbers from the user. There are ways to do that in one line, but for now we'll keep things simple. We get the numbers one at a time and give each number its own name. The only other thing to note is the parentheses in the average calculation. This is because of the order of operations. All multiplications and divisions are performed before any additions and subtractions, so we have to use parentheses to get Python to do the addition first.

1.4 Typing things in

Case Case matters. To Python, `print`, `Print`, and `PRINT` are all different things. For now, stick with lowercase as most Python statements are in lowercase.

Spaces Spaces matter at the beginning of lines, but not elsewhere. For example, the code below will not work.

```
temp = eval(input('Enter a temperature in Celsius: '))
  print('In Fahrenheit, that is', 9/5*temp+32)
```

Python uses indentation of lines for things we'll learn about soon. On the other hand, spaces in most other places don't matter. For instance, the following lines have the same effect:

```
print('Hello world!')
print ('Hello world!')
print( 'Hello world!' )
```

Basically, computers will only do what you tell them, and they often take things very literally. Python itself totally relies on things like the placement of commas and parentheses so it knows what's what. It is not very good at figuring out what you mean, so you have to be precise. It will be very frustrating at first, trying to get all of the parentheses and commas in the right places, but after a while it will become more natural. Still, even after you've programmed for a long time, you will still miss something. Fortunately, the Python interpreter is pretty good about helping you find your mistakes.

1.5 Getting input

The **input** function is a simple way for your program to get information from people using your program. Here is an example:

```
name = input('Enter your name: ')
print('Hello, ', name)
```

The basic structure is

variable name = **input**(*message to user*)

The above works for getting text from the user. To get numbers from the user to use in calculations, we need to do something extra. Here is an example:

```
num = eval(input('Enter a number: '))
print('Your number squared:', num*num)
```

The **eval** function converts the text entered by the user into a number. One nice feature of this is you can enter expressions, like `3*12+5`, and **eval** will compute them for you.

Note If you run your program and nothing seems to be happening, try pressing enter. There is a bit of a glitch in IDLE that occasionally happens with **input** statements.

1.6 Printing

Here is a simple example:

```
print('Hi there')
```

The **print** function requires parenthesis around its arguments. In the program above, its only argument is the string `'Hi there'`. Anything inside quotes will (with a few exceptions) be printed exactly as it appears. In the following, the first statement will output `3+4`, while the second will output `7`.

```
print('3+4')
print(3+4)
```

To print several things at once, separate them by commas. Python will automatically insert spaces between them. Below is an example and the output it produces.

```
print('The value of 3+4 is', 3+4)
print('A', 1, 'XYZ', 2)
```

```
The value of 3+4 is 7
A 1 XYZ 2
```

Optional arguments

There are two optional arguments to the **print** function. They are not overly important at this stage of the game, so you can safely skip over this section, but they are useful for making your output look nice.

sep Python will insert a space between each of the arguments of the print function. There is an optional argument called `sep`, short for separator, that you can use to change that space to something else. For example, using `sep=':'` would separate the arguments by a colon and `sep='##'` would separate the arguments by two pound signs.

One particularly useful possibility is to have nothing inside the quotes, as in `sep=''`. This says to put no separation between the arguments. Here is an example where `sep` is useful for getting the output to look nice:

```
print ('The value of 3+4 is', 3+4, '.')
print ('The value of 3+4 is ', 3+4, '.', sep='')
```

```
The value of 3+4 is 7 .
The value of 3+4 is 7.
```

end The print function will automatically advance to the next line. For instance, the following will print on two lines:

```
print('On the first line')
print('On the second line')
```

```
On the first line
On the second line
```

There is an optional argument called `end` that you can use to keep the print function from advancing to the next line. Here is an example:

```
print('On the first line', end='')
print('On the second line')
```

```
On the first lineOn the second line
```

Of course, this could be accomplished better with a single print, but we will see later that there are interesting uses for the `end` argument.

1.7 Variables

Looking back at our first program, we see the use of a variable called `temp`:

```
temp = eval(input('Enter a temperature in Celsius: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

One of the major purposes of a variable is to remember a value from one part of a program so that it can be used in another part of the program. In the case above, the variable `temp` stores the value that the user enters so that we can do a calculation with it in the next line.

In the example below, we perform a calculation and need to use the result of the calculation in several places in the program. If we save the result of the calculation in a variable, then we only need to do the calculation once. This also helps to make the program more readable.

```
temp = eval(input('Enter a temperature in Celsius: '))
f_temp = 9/5*temp+32
print('In Fahrenheit, that is', f_temp)
if f_temp > 212:
    print('That temperature is above the boiling point.')
if f_temp < 32:
    print('That temperature is below the freezing point.')
```

We haven't discussed if statements yet, but they do exactly what you think they do.

A second example Here is another example with variables. Before reading on, try to figure out what the values of `x` and `y` will be after the code is executed.

```
x=3
y=4
z=x+y
z=z+1
x=y
y=5
```

After these four lines of code are executed, `x` is 4, `y` is 5 and `z` is 8. One way to understand something like this is to take it one line at a time. This is an especially useful technique for trying to understand more complicated chunks of code. Here is a description of what happens in the code above:

1. `x` starts with the value 3 and `y` starts with the value 4.
2. In line 3, a variable `z` is created to equal `x+y`, which is 7.
3. Then the value of `z` is changed to equal one more than it currently equals, changing it from 7 to 8.
4. Next, `x` is changed to the current value of `y`, which is 4.
5. Finally, `y` is changed to 5. Note that this does not affect `x`.
6. So at the end, `x` is 4, `y` is 5, and `z` is 8.

Variable names

There are just a couple of rules to follow when naming your variables.

- Variable names can contain letters, numbers, and the underscore.
- Variable names *cannot* contain spaces.
- Variable names *cannot* start with a number.
- Case matters—for instance, `temp` and `Temp` are different.

It helps make your program more understandable if you choose names that are descriptive, but not so long that they clutter up your program.

1.8 Exercises

1. Print a box like the one below.

```
********************
********************
********************
********************
```

2. Print a box like the one below.

```
********************
*                  *
*                  *
********************
```

3. Print a triangle like the one below.

```
*
**
***
****
```

4. Write a program that computes and prints the result of $\frac{512-282}{47 \cdot 48 + 5}$. It is roughly .1017.

5. Ask the user to enter a number. Print out the square of the number, but use the `sep` optional argument to print it out in a full sentence that ends in a period. Sample output is shown below.

```
Enter a number: 5
The square of 5 is 25.
```

6. Ask the user to enter a number x. Use the `sep` optional argument to print out x, $2x$, $3x$, $4x$, and $5x$, each separated by three dashes, like below.

```
Enter a number: 7
7---14---21---28---35
```

7. Write a program that asks the user for a weight in kilograms and converts it to pounds. There are 2.2 pounds in a kilogram.

8. Write a program that asks the user to enter three numbers (use three separate input statements). Create variables called `total` and `average` that hold the sum and average of the three numbers and print out the values of `total` and `average`.

9. A lot of cell phones have tip calculators. Write one. Ask the user for the price of the meal and the percent tip they want to leave. Then print both the tip amount and the total bill with the tip included.

Chapter 2

For loops

Probably the most powerful thing about computers is that they can repeat things over and over very quickly. There are several ways to repeat things in Python, the most common of which is the for loop.

2.1 Examples

Example 1 The following program will print `Hello` ten times:

```
for i in range(10):
    print('Hello')
```

The structure of a for loop is as follows:

for *variable name* **in range**(*number of times to repeat*):
statements to be repeated

The syntax is important here. The word **for** must be in lowercase, the first line must end with a colon, and the statements to be repeated *must* be indented. Indentation is used to tell Python which statements will be repeated.

Example 2 The program below asks the user for a number and prints its square, then asks for another number and prints its square, etc. It does this three times and then prints that the loop is done.

```
for i in range(3):
    num = eval(input('Enter a number: '))
    print ('The square of your number is', num*num)
print('The loop is now done.')
```

```
Enter a number: 3
The square of your number is 9
Enter a number: 5
The square of your number is 25
Enter a number: 23
The square of your number is 529
The loop is now done.
```

Since the second and third lines are indented, Python knows that these are the statements to be repeated. The fourth line is not indented, so it is not part of the loop and only gets executed once, after the loop has completed.

Looking at the above example, we see where the term *for loop* comes from: we can picture the execution of the code as starting at the `for` statement, proceeding to the second and third lines, then looping back up to the `for` statement.

Example 3 The program below will print A, then B, then it will alternate C's and D's five times and then finish with the letter E once.

```
print('A')
print('B')
for i in range(5):
    print('C')
    print('D')
print('E')
```

The first two print statements get executed once, printing an A followed by a B. Next, the C's and D's alternate five times. Note that we don't get five C's followed by five D's. The way the loop works is we print a C, then a D, then loop back to the start of the loop and print a C and another D, etc. Once the program is done looping with the C's and D's, it prints one E.

Example 4 If we wanted the above program to print five C's followed by five D's, instead of alternating C's and D's, we could do the following:

```
print('A')
print('B')
for i in range(5):
    print('C')
for i in range(5):
    print('D')
print('E')
```

2.2 The loop variable

There is one part of a for loop that is a little tricky, and that is the loop variable. In the example below, the loop variable is the variable `i`. The output of this program will be the numbers 0, 1, . . . , 99, each printed on its own line.

```
for i in range(100):
    print(i)
```

When the loop first starts, Python sets the variable `i` to 0. Each time we loop back up, Python increases the value of `i` by 1. The program loops 100 times, each time increasing the value of `i` by 1, until we have looped 100 times. At this point the value of `i` is 99.

You may be wondering why `i` starts with 0 instead of 1. Well, there doesn't seem to be any really good reason why other than that starting at 0 was useful in the early days of computing and it has stuck with us. In fact most things in computer programming start at 0 instead of 1. This does take some getting used to.

Since the loop variable, `i`, gets increased by 1 each time through the loop, it can be used to keep track of where we are in the looping process. Consider the example below:

```
for i in range(3):
    print(i+1, '-- Hello')
```

```
1 -- Hello
2 -- Hello
3 -- Hello
```

Names There's nothing too special about the name `i` for our variable. The programs below will have the exact same result.

```
for i in range(100):
    print(i)
```

```
for wacky_name in range(100):
    print(wacky_name)
```

It's a convention in programming to use the letters i, j, and k for loop variables, unless there's a good reason to give the variable a more descriptive name.

2.3 The `range` function

The value we put in the **`range`** function determines how many times we will loop. The way **`range`** works is it produces a list of numbers from zero to the value minus one. For instance, **`range`**`(5)` produces five values: 0, 1, 2, 3, and 4.

If we want the list of values to start at a value other than 0, we can do that by specifying the starting value. The statement `range(1,5)` will produce the list 1, 2, 3, 4. This brings up one quirk of the `range` function—it stops one short of where we think it should. If we wanted the list to contain the numbers 1 through 5 (including 5), then we would have to do `range(1,6)`.

Another thing we can do is to get the list of values to go up by more than one at a time. To do this, we can specify an optional step as the third argument. The statement `range(1,10,2)` will step through the list by twos, producing 1, 3, 5, 7, 9.

To get the list of values to go backwards, we can use a step of -1. For instance, `range(5,1,-1)` will produce the values 5, 4, 3, 2, in that order. (Note that the `range` function stops one short of the ending value 1). Here are a few more examples:

Statement	Values generated
`range(10)`	`0,1,2,3,4,5,6,7,8,9`
`range(1,10)`	`1,2,3,4,5,6,7,8,9`
`range(3,7)`	`3,4,5,6`
`range(2,15,3)`	`2,5,8,11,14`
`range(9,2,-1)`	`9,8,7,6,5,4,3`

Here is an example program that counts down from 5 and then prints a message.

```
for i in range(5,0,-1):
    print(i, end=' ')
print('Blast off!!!')
```

```
5 4 3 2 1 Blast off!!!
```

The `end=' '` just keeps everything on the same line.

2.4 A Trickier Example

Let's look at a problem where we will make use of the loop variable. The program below prints a rectangle of stars that is 4 rows tall and 6 rows wide.

```
for i in range(4):
    print('*'*6)
```

The rectangle produced by this code is shown below on the left. The code `'*'*6` is something we'll cover in Section 6.2; it just repeats the asterisk character six times.

```
******
******
******
******
```

```
*
**
***
****
```

Suppose we want to make a triangle instead. We can accomplish this with a very small change to the rectangle program. Looking at the program, we can see that the for loop will repeat the **print** statement four times, making the shape four rows tall. It's the `6` that will need to change.

The key is to change the `6` to `i+1`. Each time through the loop the program will now print `i+1` stars instead of 6 stars. The loop counter variable `i` runs through the values 0, 1, 2, and 3. Using it allows us to vary the number of stars. Here is triangle program:

```
for i in range(4):
    print('*'*(i+1))
```

2.5 Exercises

1. Write a program that prints your name 100 times.
2. Write a program to fill the screen horizontally and vertically with your name. [Hint: add the option `end=''` into the **print** function to fill the screen horizontally.]
3. Write a program that outputs 100 lines, numbered 1 to 100, each with your name on it. The output should look like the output below.

```
1 Your name
2 Your name
3 Your name
4 Your name
...
100 Your name
```

4. Write a program that prints out a list of the integers from 1 to 20 and their squares. The output should look like this:

```
1 --- 1
2 --- 4
3 --- 9
...
20 --- 400
```

5. Write a program that uses a for loop to print the numbers 8, 11, 14, 17, 20, . . . , 83, 86, 89.
6. Write a program that uses a for loop to print the numbers 100, 98, 96, . . . , 4, 2.
7. Write a program that uses exactly four for loops to print the sequence of letters below.

```
AAAAAAAAAABBBBBBBCDCDCDCDEFFFFFFG
```

8. Write a program that asks the user for their name and how many times to print it. The program should print out the user's name the specified number of times.

9. The Fibonacci numbers are the sequence below, where the first two numbers are 1, and each number thereafter is the sum of the two preceding numbers. Write a program that asks the user how many Fibonacci numbers to print and then prints that many.

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

10. Use a for loop to print a box like the one below. Allow the user to specify how wide and how high the box should be. [Hint: `print('*'*10)` prints ten asterisks.]

```
**********
**********
**********
**********
```

11. Use a for loop to print a box like the one below. Allow the user to specify how wide and how high the box should be.

```
**********
*        *
*        *
**********
```

12. Use a for loop to print a triangle like the one below. Allow the user to specify how high the triangle should be.

```
*
**
***
****
```

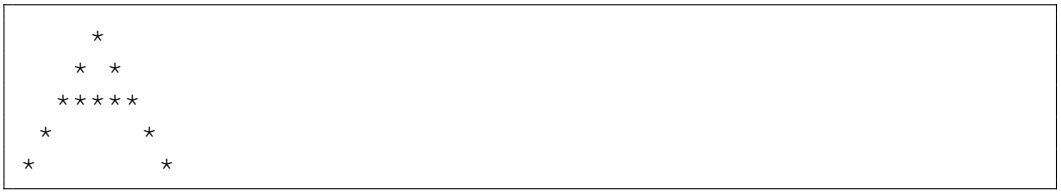
13. Use a for loop to print an upside down triangle like the one below. Allow the user to specify how high the triangle should be.

```
****
***
**
*
```

14. Use for loops to print a diamond like the one below. Allow the user to specify how high the diamond should be.

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

15. Write a program that prints a giant letter A like the one below. Allow the user to specify how large the letter should be.



Chapter 3

Numbers

This chapter focuses on numbers and simple mathematics in Python.

3.1 Integers and Decimal Numbers

Because of the way computer chips are designed, integers and decimal numbers are represented differently on computers. Decimal numbers are represented by what are called floating point numbers. The important thing to remember about them is you typically only get about 15 or so digits of precision. It would be nice if there were no limit to the precision, but calculations run a lot more quickly if you cut off the numbers at some point.

On the other hand, integers in Python have no restrictions. They can be arbitrarily large.

For decimal numbers, the last digit is sometimes slightly off due to the fact that computers work in binary (base 2) whereas our human number system is base 10. As an example, mathematically, we know that the decimal expansion of 7/3 is $2.333\cdots$, with the threes repeating forever. But when we type `7/3` into the Python shell, we get `2.3333333333333335`. This is called *roundoff error*. For most practical purposes this is not too big of a deal, but it actually can cause problems for some mathematical and scientific calculations. If you really need more precision, there are ways. See Section 22.5.

3.2 Math Operators

Here is a list of the common operators in Python:

Operator	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation
`//`	integer division
`%`	modulo (remainder)

Exponentiation Python uses `**` for exponentiation. The caret, `^`, is used for something else.

Integer division The integer division operator, `//`, requires some explanation. Basically, for positive numbers it behaves like ordinary division except that it throws away the decimal part of the result. For instance, while `8/5` is `1.6`, we have `8//5` equal to `1`. We will see uses for this operator later. Note that in many other programming languages and in older versions of Python, the usual division operator `/` actually does integer division on integers.

Modulo The modulo operator, `%`, returns the remainder from a division. For instance, the result of `18%7` is 4 because 4 is the remainder when 18 is divided by 7. This operation is surprisingly useful. For instance, a number is divisible by n precisely when it leaves a remainder of 0 when divided by `n`. Thus to check if a number, `n`, is even, see if `n%2` is equal to 0. To check if `n` is divisible by 3, see if `n%3` is 0.

One use of this is if you want to schedule something in a loop to happen only every other time through the loop, you could check to see if the loop variable modulo 2 is equal to 0, and if it is, then do that something.

The modulo operator shows up surprisingly often in formulas. If you need to "wrap around" and come back to the start, the modulo is useful. For example, think of a clock. If you go six hours past 8 o'clock, the result is 2 o'clock. Mathematically, this can be accomplished by doing a modulo by 12. That is, `(8+6)%12` is equal to 2.

As another example, take a game with players 1 through 5. Say you have a variable `player` that keeps track of the current player. After player 5 goes, it's player 1's turn again. The modulo operator can be used to take care of this:

```
player = player%5+1
```

When `player` is 5, `player%5` will be 0 and expression will set `player` to 1.

3.3 Order of operations

Exponentiation gets done first, followed by multiplication and division (including `//` and `%`), and addition and subtraction come last. The classic math class mnemonic, PEMDAS (Please Excuse My Dear Aunt Sally), might be helpful.

This comes into play in calculating an average. Say you have three variables `x`, `y`, and `z`, and you want to calculate the average of their values. To expression `x+y+z/3` would not work. Because division comes before addition, you would actually be calculating $x + y + \frac{z}{3}$ instead of $\frac{x+y+z}{3}$. This is easily fixed by using parentheses: `(x+y+z)/3`.

In general, if you're not sure about something, adding parentheses might help and usually doesn't do any harm.

3.4 Random numbers

To make an interesting computer game, it's good to introduce some randomness into it. Python comes with a module, called `random`, that allows us to use random numbers in our programs.

Before we get to random numbers, we should first explain what a *module* is. The core part of the Python language consists of things like for loops, if statements, math operators, and some functions, like **`print`** and **`input`**. Everything else is contained in modules, and if we want to use something from a module we have to first *import* it—that is, tell Python that we want to use it.

At this point, there is only one function, called `randint`, that we will need from the `random` module. To load this function, we use the following statement:

```
from random import randint
```

Using `randint` is simple: `randint(a,b)` will return a random integer between `a` and `b` including both `a` and `b`. (Note that `randint` includes the right endpoint `b` unlike the **`range`** function). Here is a short example:

```
from random import randint
x = randint(1,10)
print('A random number between 1 and 10: ', x)
```

```
A random number between 1 and 10: 7
```

The random number will be different every time we run the program.

3.5 Math functions

The `math` module Python has a module called `math` that contains familiar math functions, including `sin`, `cos`, `tan`, `exp`, `log`, `log10`, `factorial`, `sqrt`, `floor`, and `ceil`. There are also the inverse trig functions, hyperbolic functions, and the constants `pi` and `e`. Here is a short example:

```
from math import sin, pi
print('Pi is roughly', pi)
print('sin(0) =', sin(0))
```

```
Pi is roughly 3.14159265359
sin(0) = 0.0
```

Built-in math functions There are two built in math functions, **abs** (absolute value) and **round** that are available without importing the `math` module. Here are some examples:

```
print(abs(-4.3))
print(round(3.336, 2))
print(round(345.2, -1))
```

```
4.3
3.34
350.0
```

The **round** function takes two arguments: the first is the number to be rounded and the second is the number of decimal places to round to. The second argument can be negative.

3.6 Getting help from Python

There is documentation built into Python. To get help on the `math` module, for example, go to the Python shell and type the following two lines:

```
>>> import math
>>> dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

This gives a list of all the functions and variables in the `math` module. You can ignore all of the ones that start with underscores. To get help on a specific function, say the `floor` function, you can type **help**`(math.floor)`. Typing **help**`(math)` will give you help for everything in the `math` module.

3.7 Using the Shell as a Calculator

The Python shell can be used as a very handy and powerful calculator. Here is an example session:

```
>>> 23**2
529
>>> s = 0
>>> for n in range(1,10001):
        s = s + 1/n**2

>>> s
1.6448340718480652
>>> from math import *
>>> factorial(10)
3628800
```

The second example here sums the numbers $1 + 1/4 + 1/9 + \cdots + 1/10000^2$. The result is stored in the variable `s`. To inspect the value of that variable, just type its name and press enter. Inspecting variables is useful for debugging your programs. If a program is not working properly, you can type your variable names into the shell after the program has finished to see what their values are.

The statement `from math import *` imports every function from the `math` module, which can make the shell a lot like a scientific calculator.

Note Under the Shell menu, select `Restart shell` if you want to clear the values of all the variables.

3.8 Exercises

1. Write a program that generates and prints 50 random integers, each between 3 and 6.
2. Write a program that generates a random number, x, between 1 and 50, a random number y between 2 and 5, and computes x^y.
3. Write a program that generates a random number between 1 and 10 and prints your name that many times.
4. Write a program that generates a random decimal number between 1 and 10 with two decimal places of accuracy. Examples are 1.23, 3.45, 9.80, and 5.00.
5. Write a program that generates 50 random numbers such that the first number is between 1 and 2, the second is between 1 and 3, the third is between 1 and 4, ..., and the last is between 1 and 51.
6. Write a program that asks the user to enter two numbers, x and y, and computes $\frac{|x-y|}{x+y}$.
7. Write a program that asks the user to enter an angle between $-180°$ and $180°$. Using an expression with the modulo operator, convert the angle to its equivalent between $0°$ and $360°$.

8. Write a program that asks the user for a number of seconds and prints out how many minutes and seconds that is. For instance, 200 seconds is 3 minutes and 20 seconds. [Hint: Use the `//` operator to get minutes and the `%` operator to get seconds.]

9. Write a program that asks the user for an hour between 1 and 12 and for how many hours in the future they want to go. Print out what the hour will be that many hours into the future. An example is shown below.

```
Enter hour: 8
How many hours ahead? 5
New hour: 1 o'clock
```

10. (a) One way to find out the last digit of a number is to mod the number by 10. Write a program that asks the user to enter a power. Then find the last digit of 2 raised to that power.

    (b) One way to find out the last two digits of a number is to mod the number by 100. Write a program that asks the user to enter a power. Then find the last two digits of 2 raised to that power.

    (c) Write a program that asks the user to enter a power and how many digits they want. Find the last that many digits of 2 raised to the power the user entered.

11. Write a program that asks the user to enter a weight in kilograms. The program should convert it to pounds, printing the answer rounded to the nearest tenth of a pound.

12. Write a program that asks the user for a number and prints out the factorial of that number.

13. Write a program that asks the user for a number and then prints out the sine, cosine, and tangent of that number.

14. Write a program that asks the user to enter an angle in degrees and prints out the sine of that angle.

15. Write a program that prints out the sine and cosine of the angles ranging from 0 to 345° in 15° increments. Each result should be rounded to 4 decimal places. Sample output is shown below:

```
0 --- 0.0 1.0
15 --- 0.2588 0.9659
30 --- 0.5 0.866
...
345 --- -0.2588 0.9659
```

16. Below is described how to find the date of Easter in any year. Despite its intimidating appearance, this is not a hard problem. Note that $\lfloor x \rfloor$ is the *floor* function, which for positive numbers just drops the decimal part of the number. For instance $\lfloor 3.14 \rfloor = 3$. The floor function is part of the `math` module.

    C = century (1900's $\rightarrow C = 19$)

Y = year (all four digits)

$m = (15 + C - \lfloor \frac{C}{4} \rfloor - \lfloor \frac{8C+13}{25} \rfloor) \bmod 30$

$n = (4 + C - \lfloor \frac{C}{4} \rfloor) \bmod 7$

$a = Y \bmod 4$

$b = Y \bmod 7$

$c = Y \bmod 19$

$d = (19c + m) \bmod 30$

$e = (2a + 4b + 6d + n) \bmod 7$

Easter is either March $(22 + d + e)$ or April $(d + e - 9)$. There is an exception if $d = 29$ and $e = 6$. In this case, Easter falls one week earlier on April 19. There is another exception if $d = 28$, $e = 6$, and $m = 2, 5, 10, 13, 16, 21, 24$, or 39. In this case, Easter falls one week earlier on April 18. Write a program that asks the user to enter a year and prints out the date of Easter in that year. (See Tattersall, *Elementary Number Theory in Nine Chapters*, 2nd ed., page 167)

17. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Ask the user to enter a year, and, using the // operator, determine how many leap years there have been between 1600 and that year.

18. Write a program that given an amount of change less than $1.00 will print out exactly how many quarters, dimes, nickels, and pennies will be needed to efficiently make that change. [Hint: the // operator may be useful.]

19. Write a program that draws "modular rectangles" like the ones below. The user specifies the width and height of the rectangle, and the entries start at 0 and increase typewriter fashion from left to right and top to bottom, but are all done mod 10. Below are examples of a 3×5 rectangle and a 4×8.

```
0 1 2 3 4
5 6 7 8 9
0 1 2 3 4

0 1 2 3 4 5 6 7
8 9 0 1 2 3 4 5
6 7 8 9 0 1 2 3
4 5 6 7 8 9 0 1
```

Chapter 4

If statements

Quite often in programs we only want to do something provided something else is true. Python's `if` statement is what we need.

4.1 A Simple Example

Let's try a guess-a-number program. The computer picks a random number, the player tries to guess, and the program tells them if they are correct. To see if the player's guess is correct, we need something new, called an *if statement*.

```
from random import randint

num = randint(1,10)
guess = eval(input('Enter your guess: '))
if guess==num:
    print('You got it!')
```

The syntax of the if statement is a lot like the `for` statement in that there is a colon at the end of the if condition and the following line or lines are indented. The lines that are indented will be executed only if the condition is true. Once the indentation is done with, the if block is concluded.

The guess-a-number game works, but it is pretty simple. If the player guesses wrong, nothing happens. We can add to the if statement as follows:

```
if guess==num:
    print('You got it!')
else:
    print('Sorry.  The number is ', num)
```

We have added an `else` statement, which is like an "otherwise."

4.2 Conditional operators

The comparison operators are ==, >, <, >=, <=, and !=. That last one is for *not equals*. Here are a few examples:

Expression	Description
`if x>3:`	if x is greater than 3
`if x>=3:`	if x is greater than or equal to 3
`if x==3:`	if x is 3
`if x!=3:`	if x is not 3

There are three additional operators used to construct more complicated conditions: `and`, `or`, and `not`. Here are some examples:

```
if grade>=80 and grade<90:
    print('Your grade is a B.')

if score>1000 or time>20:
    print('Game over.')

if not (score>1000 or time>20):
    print('Game continues.')
```

Order of operations In terms of order of operations, `and` is done before `or`, so if you have a complicated condition that contains both, you may need parentheses around the `or` condition. Think of `and` as being like multiplication and `or` as being like addition. Here is an example:

```
if (score<1000 or time>20) and turns_remaining==0:
    print('Game over.')
```

4.3 Common Mistakes

Mistake 1 The operator for equality consists of two equals signs. It is a really common error to forget one of the equals signs.

Incorrect	Correct
`if x=1:`	`if x==1:`

Mistake 2 A common mistake is to use `and` where `or` is needed or vice-versa. Consider the following if statements:

```
if x>1 and x<100:
if x>1 or x<100:
```

The first statement is the correct one. If x is any value between 1 and 100, then the statement will be true. The idea is that x has to be *both* greater than 1 *and* less than 100. On the other hand, the second statement is not what we want because for it to be true, *either* x has to be greater than 1 *or* x has to be less than 100. But every number satisfies this. The lesson here is if your program is not working correctly, check your **and**'s and **or**'s.

Mistake 3 Another very common mistake is to write something like below:

```
if grade>=80 and <90:
```

This will lead to a syntax error. We have to be explicit. The correct statement is

```
if grade>=80 and grade<90:
```

On the other hand, there is a nice shortcut that does work in Python (though not in many other programming languages):

```
if 80<=grade<90:
```

4.4 elif

A simple use of an if statement is to assign letter grades. Suppose that scores 90 and above are A's, scores in the 80s are B's, 70s are C's, 60s are D's, and anything below 60 is an F. Here is one way to do this:

```
grade = eval(input('Enter your score: '))

if grade>=90:
    print('A')
if grade>=80 and grade<90:
    print('B')
if grade>=70 and grade<80:
    print('C')
if grade>=60 and grade<70:
    print('D')
if grade<60:
    print('F')
```

The code above is pretty straightforward and it works. However, a more elegant way to do it is shown below.

```
grade = eval(input('Enter your score: '))

if grade>=90:
    print('A')
elif grade>=80:
    print('B')
elif grade>=70:
    print('C')
```

```
elif grade>=60:
    print('D')
else:
    print('F')
```

With the separate if statements, each condition is checked regardless of whether it really needs to be. That is, if the score is a 95, the first program will print an A but then continue on and check to see if the score is a B, C, etc., which is a bit of a waste. Using `elif`, as soon as we find where the score matches, we stop checking conditions and skip all the way to the end of the whole block of statements. An added benefit of this is that the conditions we use in the `elif` statements are simpler than in their `if` counterparts. For instance, when using `elif`, the second part of the second if statement condition, `grade<90`, becomes unnecessary because the corresponding `elif` does not have to worry about a score of 90 or above, as such a score would have already been caught by the first if statement.

You can get along just fine without `elif`, but it can often make your code simpler.

4.5 Exercises

1. Write a program that asks the user to enter a length in centimeters. If the user enters a negative length, the program should tell the user that the entry is invalid. Otherwise, the program should convert the length to inches and print out the result. There are 2.54 centimeters in an inch.

2. Ask the user for a temperature. Then ask them what units, Celsius or Fahrenheit, the temperature is in. Your program should convert the temperature to the other unit. The conversions are $F = \frac{9}{5}C + 32$ and $C = \frac{5}{9}(F - 32)$.

3. Ask the user to enter a temperature in Celsius. The program should print a message based on the temperature:

   - If the temperature is less than -273.15, print that the temperature is invalid because it is below absolute zero.
   - If it is exactly -273.15, print that the temperature is absolute 0.
   - If the temperature is between -273.15 and 0, print that the temperature is below freezing.
   - If it is 0, print that the temperature is at the freezing point.
   - If it is between 0 and 100, print that the temperature is in the normal range.
   - If it is 100, print that the temperature is at the boiling point.
   - If it is above 100, print that the temperature is above the boiling point.

4. Write a program that asks the user how many credits they have taken. If they have taken 23 or less, print that the student is a freshman. If they have taken between 24 and 53, print that they are a sophomore. The range for juniors is 54 to 83, and for seniors it is 84 and over.

5. Generate a random number between 1 and 10. Ask the user to guess the number and print a message based on whether they get it right or not.

6. A store charges $12 per item if you buy less than 10 items. If you buy between 10 and 99 items, the cost is $10 per item. If you buy 100 or more items, the cost is $7 per item. Write a program that asks the user how many items they are buying and prints the total cost.

7. Write a program that asks the user for two numbers and prints `Close` if the numbers are within .001 of each other and `Not close` otherwise.

8. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Write a program that asks the user for a year and prints out whether it is a leap year or not.

9. Write a program that asks the user to enter a number and prints out all the divisors of that number. [Hint: the `%` operator is used to tell if a number is divisible by something. See Section 3.2.]

10. Write a multiplication game program for kids. The program should give the player ten randomly generated multiplication questions to do. After each, the program should tell them whether they got it right or wrong and what the correct answer is.

```
Question 1: 3 x 4 = 12
Right!
Question 2: 8 x 6 = 44
Wrong.  The answer is 48.
...
...
Question 10: 7 x 7 = 49
Right.
```

11. Write a program that asks the user for an hour between 1 and 12, asks them to enter `am` or `pm`, and asks them how many hours into the future they want to go. Print out what the hour will be that many hours into the future, printing `am` or `pm` as appropriate. An example is shown below.

```
Enter hour: 8
am (1) or pm (2)? 1
How many hours ahead? 5
New hour: 1 pm
```

12. A jar of Halloween candy contains an unknown amount of candy and if you can guess exactly how much candy is in the bowl, then you win all the candy. You ask the person in charge the following: If the candy is divided evenly among 5 people, how many pieces would be left over? The answer is 2 pieces. You then ask about dividing the candy evenly among 6 people, and the amount left over is 3 pieces. Finally, you ask about dividing the candy evenly among 7 people, and the amount left over is 2 pieces. By looking at the bowl, you can tell that there are less than 200 pieces. Write a program to determine how many pieces are in the bowl.

13. Write a program that lets the user play Rock-Paper-Scissors against the computer. There should be five rounds, and after those five rounds, your program should print out who won and lost or that there is a tie.

Chapter 5

Miscellaneous Topics I

This chapter consists of a several common techniques and some other useful information.

5.1 Counting

Very often we want our programs to count how many times something happens. For instance, a video game may need to keep track of how many turns a player has used, or a math program may want to count how many numbers have a special property. The key to counting is to use a variable to keep the count.

Example 1 This program gets 10 numbers from the user and counts how many of those numbers are greater than 10.

```
count = 0
for i in range(10):
    num = eval(input('Enter a number: '))
    if num>10:
        count=count+1
print('There are', count, 'numbers greater than 10.')
```

Think of the `count` variable as if we are keeping a tally on a piece of paper. Every time we get a number larger than 10, we add 1 to our tally. In the program, this is accomplished by the line `count=count+1`. The first line of the program, `count=0`, is important. Without it, the Python interpreter would get to the `count=count+1` line and spit out an error saying something about not knowing what `count` is. This is because the first time the program gets to this line, it tries to do what it says: take the old value of `count`, add 1 to it, and store the result in `count`. But the first time the program gets there, there is no old value of `count` to use, so the Python interpreter doesn't know what to do. To avoid the error, we need to define `count`, and that is what the first

line does. We set it to 0 to indicate that at the start of the program no numbers greater than 10 have been found.

Counting is an extremely common thing. The two things involved are:

1. `count=0` — Start the count at 0.
2. `count=count+1` — Increase the count by 1.

Example 2 This modification of the previous example counts how many of the numbers the user enters are greater than 10 and also how many are equal to 0. To count two things we use two count variables.

```
count1 = 0
count2 = 0
for i in range(10):
    num = eval(input('Enter a number: '))
    if num>10:
        count1=count1+1
    if num==0:
        count2=count2+1
print('There are', count1, 'numbers greater than 10.')
print('There are', count2, 'zeroes.')
```

Example 3 Next we have a slightly trickier example. This program counts how many of the squares from 1^2 to 100^2 end in a 4.

```
count = 0
for i in range(1,101):
    if (i**2)%10==4:
        count = count + 1
print(count)
```

A few notes here: First, because of the aforementioned quirk of the `range` function, we need to use `range(1,101)` to loop through the numbers 1 through 100. The looping variable `i` takes on those values, so the squares from 1^2 to 100^2 are represented by `i**2`. Next, to check if a number ends in 4, a nice mathematical trick is to check if it leaves a remainder of 4 when divided by 10. The modulo operator, `%`, is used to get the remainder.

5.2 Summing

Closely related to counting is summing, where we want to add up a bunch of numbers.

Example 1 This program will add up the numbers from 1 to 100. The way this works is that each time we encounter a new number, we add it to our running total, `s`.

```
s = 0
for i in range(1,101):
    s = s + i
print('The sum is', s)
```

Example 2 This program that will ask the user for 10 numbers and then computes their average.

```
s = 0
for i in range(10):
    num = eval(input('Enter a number: '))
    s = s + num
print('The average is', s/10)
```

Example 3 A common use for summing is keeping score in a game. Near the beginning of the game we would set the score variable equal to 0. Then when we want to add to the score we would do something like below:

```
score = score + 10
```

5.3 Swapping

Quite often we will want to swap the values of two variables, `x` and `y`. It would be tempting to try the following:

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

But this will not work. Suppose `x` is 3 and `y` is 5. The first line will set `x` to 5, which is good, but then the second line will set `y` to 5 also because `x` is now 5. The trick is to use a third variable to save the value of `x`:

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

In many programming languages, this is the usual way to swap variables. Python, however, provides a nice shortcut:

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

We will learn later exactly why this works. For now, feel free to use whichever method you prefer. The latter method, however, has the advantage of being shorter and easier to understand.

5.4 Flag variables

A flag variable can be used to let one part of your program know when something happens in another part of the program. Here is an example that determines if a number is prime.

```
num = eval(input('Enter number: '))

flag = 0
for i in range(2,num):
    if num%i==0:
        flag = 1

if flag==1:
    print('Not prime')
else:
    print('Prime')
```

Recall that a number is prime if it has no divisors other than 1 and itself. The way the program above works is `flag` starts off at 0. We then loop from 2 to `num-1`. If one of those values turns out to be a divisor, then `flag` gets set to 1. Once the loop is finished, we check to see if the flag got set or not. If it did, we know there was a divisor, and `num` isn't prime. Otherwise, the number must be prime.

5.5 Maxes and mins

A common programming task is to find the largest or smallest value in a series of values. Here is an example where we ask the user to enter ten positive numbers and then we print the largest one.

```
largest = eval(input('Enter a positive number: '))
for i in range(9):
    num = eval(input('Enter a positive number: '))
    if num>largest:
        largest=num
print('Largest number:', largest)
```

The key here is the variable `largest` that keeps track of the largest number found so far. We start by setting it equal to the user's first number. Then, every time we get a new number from the user, we check to see if the user's number is larger than the current largest value (which is stored in `largest`). If it is, then we set `largest` equal to the user's number.

If, instead, we want the smallest value, the only change necessary is that `>` becomes `<`, though it would also be good to rename the variable `largest` to `smallest`.

Later on, when we get to lists, we will see a shorter way to find the largest and smallest values, but the technique above is useful to know since you may occasionally run into situations where the list way won't do everything you need it to do.

5.6 Comments

A comment is a message to someone reading your program. Comments are often used to describe what a section of code does or how it works, especially with tricky sections of code. Comments have no effect on your program.

Single-line comments For a single-line comment, use the # character.

```
# a slightly sneaky way to get two values at once
num1, num2 = eval(input('Enter two numbers separated by commas: '))
```

You can also put comments at the end of a line:

```
count = count + 2  # each divisor contributes two the count
```

Multi-line comments For comments that span several lines, you can use triple quotes.

```
""" Program name: Hello world
    Author: Brian Heinold
    Date: 1/9/11 """

print('Hello world')
```

One nice use for the triple quotes is to comment out parts of your code. Often you will want to modify your program but don't want to delete your old code in case your changes don't work. You could comment out the old code so that it is still there if you need it, and it will be ignored when your new program is run. Here is a simple example:

```
"""
print('This line and the next are inside a comment.')
print('These lines will not get executed.')
"""
print('This line is not in a comment and it will be executed.')
```

5.7 Simple debugging

Here are two simple techniques for figuring out why a program is not working:

1. Use the Python shell. After your program has run, you can type in the names of your program's variables to inspect their values and see which ones have the values you expect them to have and which don't. You can also use the Shell to type in small sections of your program and see if they are working.

2. Add print statements to your program. You can add these at any point in your program to see what the values of your variables are. You can also add a print statement to see if a point in your code is even being reached. For instance, if you think you might have an error in

a condition of an if statement, you can put a print statement into the if block to see if the condition is being triggered.

Here is an example from the part of the primes program from earlier in this chapter. We put a print statement into the for loop to see exactly when the flag variable is being set:

```
flag = 0
num = eval(input('Enter number: '))
for i in range(2,num):
    if num%i==0:
        flag = 1
    print(i, flag)
```

3. An empty input statement, like below, can be used to pause your program at a specific point:

```
input()
```

5.8 Example programs

It is a valuable skill is to be able to read code. In this section we will look in depth at some simple programs and try to understand how they work.

Example 1 The following program prints `Hello` a random number of times between 5 and 25.

```
from random import randint

rand_num = randint(5,25)
for i in range(rand_num):
    print('Hello')
```

The first line in the program is the import statement. This just needs to appear once, usually near the beginning of your program. The next line generates a random number between 5 and 25. Then, remember that to repeat something a specified number of times, we use a for loop. To repeat something 50 times, we would use `range(50)` in our for loop. To repeat something 100 times, we would use `range(100)`. To repeat something a random number of times, we can use `range(rand_num)`, where `rand_num` is a variable holding a random number. Although if we want, we can skip the variable and put the `randint` statement directly in the `range` function, as shown below.

```
from random import randint

for i in range(randint(5,25)):
    print('Hello')
```

Example 2 Compare the following two programs.

```
from random import randint

rand_num = randint(1,5)
for i in range(6):
    print('Hello'*rand_num)
```

```
from random import randint

for i in range(6):
    rand_num = randint(1,5)
    print('Hello'*rand_num)
```

```
Hello Hello
Hello Hello
Hello Hello
Hello Hello
Hello Hello
Hello Hello
```

```
Hello Hello Hello
Hello
Hello Hello Hello Hello
Hello Hello Hello
Hello Hello
Hello
```

The only difference between the programs is in the placement of the `rand_num` statement. In the first program, it is located outside of the for loop, and this means that `rand_num` is set once at the beginning of the program and retains that same value for the life of the program. Thus every print statement will print `Hello` the same number of times. In the second program, the `rand_num` statement is within the loop. Right before each print statement, `rand_num` is assigned a new random number, and so the number of times `Hello` is printed will vary from line to line.

Example 3 Let us write a program that generates 10000 random numbers between 1 and 100 and counts how many of them are multiples of 12. Here are the things we will need:

- Because we are using random numbers, the first line of the program should import the `random` module.
- We will require a for loop to run 10000 times.
- Inside the loop, we will need to generate a random number, check to see if it is divisible by 12, and if so, add 1 to the count.
- Since we are counting, we will also need to set the count equal to 0 before we start counting.
- To check divisibility by 12, we use the modulo, `%`, operator.

When we put this all together, we get the following:

```
from random import randint

count = 0
for i in range(10000):
    num = randint(1, 100)
    if num%12==0:
        count=count+1

print('Number of multiples of 12:', count)
```

Indentation matters

A common mistake is incorrect indentation. Suppose we take the above and indent the last line. The program will still run, but it won't run as expected.

```
from random import randint

count = 0
for i in range(10000):
    num = randint(1, 100)
    if num%12==0:
        count=count+1
    print('Number of multiples of 12:', count)
```

When we run it, it outputs a whole bunch of numbers. The reason for this is that by indenting the print statement, we have made it a part of the for loop, so the print statement will be executed 10,000 times.

Suppose we indent the print statement one step further, like below.

```
from random import randint

count = 0
for i in range(10000):
    num = randint(1, 100)
    if num%12==0:
        count=count+1
        print('Number of multiples of 12:', count)
```

Now, not only is it part of the for loop, but it is also part of the if statement. What will happen is every time we find a new multiple of 12, we will print the count. Neither this, nor the previous example, is what we want. We just want to print the count once at the end of the program, so we don't want the print statement indented at all.

5.9 Exercises

1. Write a program that counts how many of the squares of the numbers from 1 to 100 end in a 1.

2. Write a program that counts how many of the squares of the numbers from 1 to 100 end in a 4 and how many end in a 9.

3. Write a program that asks the user to enter a value n, and then computes $(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n})-\ln(n)$. The ln function is `log` in the `math` module.

4. Write a program to compute the sum $1-2+3-4+\cdots+1999-2000$.

5. Write a program that asks the user to enter a number and prints the sum of the divisors of that number. The sum of the divisors of a number is an important function in number theory.

6. A number is called a *perfect number* if it is equal to the sum of all of its divisors, not including the number itself. For instance, 6 is a perfect number because the divisors of 6 are 1, 2, 3, 6 and $6 = 1 + 2 + 3$. As another example, 28 is a perfect number because its divisors are 1, 2, 4, 7, 14, 28 and $28 = 1 + 2 + 4 + 7 + 14$. However, 15 is not a perfect number because its divisors are 1, 3, 5, 15 and $15 \neq 1 + 3 + 5$. Write a program that finds all four of the perfect numbers that are less than 10000.

7. An integer is called *squarefree* if it is not divisible by any perfect squares other than 1. For instance, 42 is squarefree because its divisors are 1, 2, 3, 6, 7, 21, and 42, and none of those numbers (except 1) is a perfect square. On the other hand, 45 is not squarefree because it is divisible by 9, which is a perfect square. Write a program that asks the user for an integer and tells them if it is squarefree or not.

8. Write a program that swaps the values of three variables x, y, and z, so that x gets the value of y, y gets the value of z, and z gets the value of x.

9. Write a program to count how many integers from 1 to 1000 are not perfect squares, perfect cubes, or perfect fifth powers.

10. Ask the user to enter 10 test scores. Write a program to do the following:

    (a) Print out the highest and lowest scores.

    (b) Print out the average of the scores.

    (c) Print out the second largest score.

    (d) If any of the scores is greater than 100, then after all the scores have been entered, print a message warning the user that a value over 100 has been entered.

    (e) Drop the two lowest scores and print out the average of the rest of them.

11. Write a program that computes the factorial of a number. The factorial, $n!$, of a number n is the product of all the integers between 1 and n, including n. For instance, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. [Hint: Try using a multiplicative equivalent of the summing technique.]

12. Write a program that asks the user to guess a random number between 1 and 10. If they guess right, they get 10 points added to their score, and they lose 1 point for an incorrect guess. Give the user five numbers to guess and print their score after all the guessing is done.

13. In the last chapter there was an exercise that asked you to create a multiplication game for kids. Improve your program from that exercise to keep track of the number of right and wrong answers. At the end of the program, print a message that varies depending on how many questions the player got right.

14. This exercise is about the well-known Monty Hall problem. In the problem, you are a contestant on a game show. The host, Monty Hall, shows you three doors. Behind one of those doors is a prize, and behind the other two doors are goats. You pick a door. Monty Hall, who

knows behind which door the prize lies, then opens up one of the doors that doesn't contain the prize. There are now two doors left, and Monty gives you the opportunity to change your choice. Should you keep the same door, change doors, or does it not matter?

(a) Write a program that simulates playing this game 10000 times and calculates what percentage of the time you would win if you switch and what percentage of the time you would win by not switching.

(b) Try the above but with four doors instead of three. There is still only one prize, and Monty still opens up one door and then gives you the opportunity to switch.