1.

Computer Systems and Software

Programmable software is what makes a computer a powerful tool. Each different program essentially “rewires” the computer to allow it to perform a different task. In this course, you will learn basic principles of writing software in the Python programming language.

Python is a popular scripting language available as a free download from `http://www.python.org/`. Follow the instructions given there to install the latest production version of Python 3 on your system. The examples in this text were written with Python 3.2.

The CPU and RAM

In order to write software, it will be helpful to be able to imagine what happens inside a computer when your program runs. We begin with a rough picture and gradually fill in details along the way.

When a program is ready to run, it is loaded into RAM, usually from long-term storage such as a hard drive. **RAM** is an acronym for random access memory, which is the working **memory** of a computer. RAM is **volatile**, meaning that it requires electricity to keep its contents.

Once a program is loaded into RAM, the **CPU**, or central processing unit, executes the instructions of the program, one at a time. Each type of CPU has its own **instruction set**, and you might be surprised at how limited these instruction sets are. Most instructions boil down to a few simple types: load data, perform arithmetic, and store data. What is amazing is that these small steps can be combined in different ways to build programs that are enormously complex, such as games, spreadsheets, and physics simulations.

Computer Languages

CPU instruction sets are also known as **machine languages**. The key point to remember about machine languages is that in order to be run by a CPU, a program *must* be written in the machine language of that CPU. Unfortunately, machine languages are not meant to be read or written by humans. They are really just specific sequences of bits in memory. (We will explain bits later if you do not know what they are.)

Because of this, people usually write software in a **higher-level language**:

Level	**Language**	**Purposes**
Higher	Python	Scripts
	Java	Applications
	C, C++	Applications, Systems
	Assembly Languages	Specialized Tasks
Lower	Machine Languages	

This table is not meant to be precise, but, for example, most programmers would agree that C and C++ are closer to the machine than Python.

Compilation and Interpretation

Now if CPUs can only run programs written in their own machine language, how do we run programs written in Python, Java, or C++? The answer is that the program must be translated into machine language first.

There are two main types of translation: compilation and interpretation. When a program is **compiled**, it is completely translated into machine language, producing an executable file. C and C++ programs are usually compiled, and when they are compiled in Microsoft Windows®, for example, the executable files generally end in `.exe`.

On the other hand, when a program is **interpreted**, it is translated "on-the-fly." No separate executable file is created. Instead, the translator program (the **interpreter**) is running and it translates your program so that the CPU can execute it. Python programs are usually interpreted.

The Python Interpreter

When you start Python, you are in immediate contact with a Python interpreter. If you provide it with legitimate Python, the interpreter will translate your code so that it can be executed by the CPU. The interpreter displays the version of Python that it will interpret and then shows that it is ready for your input with this prompt:

```
>>>
```

The interpreter will translate and execute any legal Python code that is typed at the prompt. For example, if we enter the following statement:

```
>>> print("Hello!")
```

the interpreter will respond accordingly. Try it and see.

Remember that as you learn new Python constructs, you can always try them out in the interpreter without having to write a complete program. Experiment—the interpreter will not mind.

A Python Program

Still, our focus will be on writing complete Python programs. Here is a short example:

Listing 1.1: Area of a Circle

```python
from math import pi
r = 12
area = pi * r ** 2
print("The area of a circle with radius", r, "is", area)
```

Even if you have never seen Python before, you can probably figure out what this program does.

Start the Python **IDLE** application and choose "New Window" from the File menu. Type Listing 1.1 into the new window that appears. Save it as `circle.py`, and then either choose "Run Module" from the Run menu or press F5 to run the program. This program illustrates many important Python concepts, including variables, numeric expressions, assignment, output, and using the standard library. We will examine these components in the next chapter.

Why Computer Science?

Here are a few things to consider as we begin:

1. Software is everywhere. If you are skeptical, search for "weather forecast toaster."

2. Similarly, computation is having a significant impact on the way other disciplines do their work. And for almost every field X, there is a new interdisciplinary field "Computational X."

3. Programming develops your ability to solve problems. Because machine languages are so simplistic, you have to tell the computer *everything* it needs to do in order to solve a problem. Furthermore, running a program provides concrete feedback on whether or not your solution is correct.

4. Computer science develops your ability to understand systems. Software systems are among the most complicated artifacts ever created by humans, and learning to manage complexity in a program will help you learn to manage it in other areas.

5. Programming languages are tools for creation: they let you build cool things. There is nothing quite like getting an idea for a program and seeing it come to life. And then showing it to all your friends.

Exercises

1.1 Experiment with Listing 1.1 by making changes to various parts of the code. Be sure to try some things that break the program (cause errors), just to see how the interpreter reacts.

1.2 Modify Listing 1.1 to also compute and display the circumference of the circle.

1.3 The December 1978 issue of the IEEE Computer Society journal *Computer* contained the following description of a new computer that fit "on the top of any business desk":

> The PCC 2000 consists of a 3MHz 8085A microprocessor, two 32K memory boards, two FD514 double-density, 8.5-inch floppy disk drives, a 12-inch upper/lower case video display...

(a) Compare the PCC 2000's CPU speed and amount of RAM to current desktops.

(b) Does the PCC 2000 appear to have a hard disk? If not, what does it use for long-term storage?

(c) Research the capacity of one of these floppy disks.

(d) List possible reasons that the PCC 2000 has two floppy disk drives instead of one.

2.

Python Program Components

Let's look at another Python program that creates short nonsense sentences:

Listing 2.1: Mad Lib

```
# madlib.py
# Your Name

adjective = input("Enter an adjective: ")
noun = input("Enter a noun: ")
verb = input("Enter a verb: ")
adverb = input("Enter an adverb: ")
print("A", adjective, noun, "should never", verb, adverb)
```

Type this program into your Python environment, save it as `madlib.py`, and run it a few times to play with it. Enter any responses you like when the interpreter asks for them. Then consider the following aspects of both this program and Listing 1.1.

Variables

Every program accomplishes its work by manipulating data. **Variables** are names that refer to data in memory. Variable names, also known as **identifiers** in Python, may consist of upper and lower alphabetic characters, the underscore (_), and, except for the first character, the digits 0–9. There is a small set of reserved Python **keywords** that may not be used as variable names; otherwise, you are free to choose any names you wish. You should already be able to see how meaningful identifiers can make a program easier to follow. The variables in Listing 2.1 are `adjective`, `noun`, `verb`, and `adverb`.

Program Statements and Syntax

A program is a sequence of **statements**, which are individual commands executed one after another by the Python interpreter. Every statement must have the correct syntax; the **syntax** of any language is the precise form that it is written in. Thus, if a statement does not have the correct form, the Python interpreter will respond with a **syntax error**. Three types of statements are used in Listings 1.1 and 2.1:

Assignment statements are used to give variables a value. The syntax of an assignment statement is:

```
<variable> = <expression>
```

Read assignment statements from right to left:

1. Evaluate the expression on the right.
2. Assign the variable on the left to refer to that value.

For example, the assignment

```
x = 10 - 17 + 5
```

computes the right side to be -2 and then assigns `x` to refer to -2.

$\Longrightarrow$ Caution: An assignment statement “=” is not like a mathematical equals sign. Technically, it is a **delimiter** because it helps the interpreter delimit between the variable on the left and the expression on the right.

Print statements are used to produce program output. The syntax of a `print()` statement is:

```
print(<expression1>, <expression2>, ...)
```

Expressions inside quotation marks (either single or double) are printed literally, whereas expressions outside quotation marks have their value printed. Expressions are separated by single spaces in the output, and each print statement produces output on a separate line.

Technically, `print()` is a **built-in function** that we call in order to print, but it is generally called as its own separate statement.

Import statements are used to access library functions or data, as in the first line of Listing 1.1. The **Python Standard Library** consists of many **modules**, each of which adds a specific set of additional functionality. The statement:

```
from <module> import <name1>, <name2>, ...
```

allows the listed names to be used in your program.

The `math` module includes the constant `pi` and functions such as `sin`, `cos`, `log` and `exp`. Both **`import`** and **`from`** are reserved Python keywords and so may not be used as the names of variables. Keywords appear in bold in program listings.

Data Types

Listings 1.1 and 2.1 use three different types of data:

Strings are sequences of characters inside single or double quotation marks.

Integers are whole number values such as 847, −19, or 7.

Floats (short for "floating point") are values that use a decimal point and therefore may have a fractional part. For example, 3.14159, −23.8, and even 7.0 (because it has a decimal point) are all considered floats.

A variable in Python may refer to any type of data.

Expressions

Recall that the right-hand side of every assignment statement must be an **expression**, which in Python is just something that can be evaluated to give a value.

Input expressions are used to ask the user of a program for information. They almost always appear on the right side of an assignment statement; using an `input` expression allows the variable on the left side to refer to different values each time the program is run.

`input(prompt)`	Display `prompt` and return user input as a string.

The **prompt** is printed to alert the user that the program is waiting for input.

Numeric expressions use the **arithmetic operations** `+`, `-`, `*`, `/`, `//`, and `**` for addition, subtraction, multiplication, division, integer division, and raising to a power. These follow the normal rules of arithmetic and **operator precedence**: exponentiation is done first, then multiplication and division (left to right), and finally addition and subtraction (also left to right). Thus, `1 + 2 * 3` evaluates to 7 because multiplication is done before addition. Parentheses may be used to change the order of operations.

Integer division rounds down to the nearest integer, so `7//2` equals 3 and `-1//3` equals −1. Integer division applied to floats rounds down to the nearest integer, but the result is still of type float.

Finally, variables must be assigned a value before being used in an expression.

Comments

Comments begin with a pound sign # and signify that whatever follows is to be completely ignored by the interpreter.

```
# Text that helps explain your program to others
```

Comments may appear anywhere in a Python program and are meant for human readers rather than the interpreter, in order to explain some aspect of the code.

Recap

There was a lot of terminology here, so let's summarize what is most important:

1. Variables refer to data...
2. ...via assignment statements. Remember to read them from right to left.
3. Data can be of type string, integer, or float (so far). The data type of a variable determines what you can do with it.

Exercises

2.1 Give three names that are not legal Python identifiers.

2.2 Use Listing 1.1 to:

   (a) List each variable and give the type of data stored in it. Hint: `pi` is a variable.
   (b) Identify the assignment statements, and explain the effect of each.

2.3 Determine the output of Listing 1.1 if the `print()` statement is changed to:

```
print("The area of a circle with radius r is area")
```

   Explain the result.

2.4 Use Listing 2.1 to:

   (a) List each assignment statement and identify the variable whose content changes in each.

(b) Describe what happens if the final space in any of the `input` statements is deleted.

2.5 Determine the value of each of these Python expressions:

(a) `1 + 2 * 3 - 4 * 5 + 6`

(b) `1 + 2 ** 3 * 4 - 5`

(c) `1 / 2 - 3 / 4`

(d) `1 // 2 - 3 // 4 + 5 // 6`

(e) `1 + 4 - 2 / 2`

(f) `(1 + 4 - 2) / 2`

2.6 Determine the output of each of these code fragments:

(a)
```
x = 10
y = 15
print(x, y)
y = x
print(x, y)
x = 5
print(x, y)
```

(b)
```
x = 10
y = 15
print(x, y)
y = x
x = y
print(x, y)
x = 5
print(x, y)
```

(c)
```
x = 10
y = 15
z = 20
x = z
z = y
y = x
print(x, y, z)
```

2.7 Modify Listing 2.1 to create your own Mad Lib program. You may want to look ahead to Chapter 8 to use the `sep=` or `end=` options of the `print()` statement to better control your output.

2.8 Explain the difficulty with using `input()` to get numeric values at this stage. (We will address it soon.)

2.9 Modify Listing 1.1 to write a program `average.py` that calculates and prints the average of two numbers.

2.10 Modify Listing 1.1 to write a program `rect.py` that calculates and prints the area and perimeter of a rectangle given its length and width. Run your program with several different values of the length and width to test it.

2.11 Modify Listing 1.1 to write a program `cube.py` that calculates and prints the volume and surface area of a cube given its width. Run your program with different values of the width to find the point at which volume equals surface area.

2.12 Modify Listing 1.1 to write a program `sphere.py` that calculates and prints the volume and surface area of a sphere given its radius. Look up the formulae if you do not remember them. Run your program with different values of the radius to find the point at which volume equals surface area.

3.

Functions

Functions in Python allow you to break a task down into appropriate subtasks. They are one of the powerful tools of abstraction that higher-level programming languages provide. Ideally, every function has a single, well-defined purpose.

Consider this example, which uses the `sqrt()` function from the `math` module to implement `hypot()`, also in the `math` module.

Listing 3.1: Hypotenuse

```
1  # hypot.py
2
3  from math import sqrt
4
5  def myhypot(x, y):
6      return sqrt(x ** 2 + y ** 2)
7
8  def main():
9      a = float(input("a: "))
10     b = float(input("b: "))
11     print("Hypotenuse:", myhypot(a, b))
12
13 main()
```

Each chapter, as we begin with a new example, type the program in to your programming environment (such as IDLE), save it with the indicated name, and run it a few times before you continue. Try to determine how the program works, even though it may use new features that you have not seen before.

For our purposes, a Python **program** can be thought of as a collection of function definitions with one main function call. Listing 3.1 defines and calls two functions: `myhypot()` and `main()`. A function cannot be *called* unless it has been *defined*. The function `myhypot()` is defined in lines 5–6 and called in line 11. The function `main()` is defined in lines 8–11 and called in line 13. The order of function definitions is not important: as long as `myhypot()` is defined somewhere in this file, the definition of `main()` could have come first.

Code that is outside all function definitions (like all of Listings 1.1 and 2.1) is directly executed when the program is run. In Listing 3.1, only the function call in line 13 is outside all function definitions, and so it will be executed when the program is run. In a sense, it "drives" execution of the whole program. Because of this, the `main()` function is known as the **driver**. Usually, the driver definition and call are put at the end of a Python program.

Defining a Function

A function **definition** specifies the code that will be executed when the function is called. The syntax of a function definition looks like this:

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

The **def** line must end with a **colon**. The colon signals the beginning of an indented **block** of code, in this case called the **body** of the function. The body is the code that will be executed when the function is called and may consist of any number of lines. In Listing 3.1, line 6 is the entire body of the `myhypot()` function.

Parameters are used to send additional information to a function so that it can do its job. The `<parameters>` in a function definition are optional. In Listing 3.1, the `myhypot()` function has two parameters named `x` and `y`.

Usually, function definitions appear near the top of Python programs, after any **import** statements, but before any directly executable code.

Calling a Function

Once a function has been defined, it may be **called**, meaning that it will be executed with particular **arguments** passed as the values for its parameters. The syntax for a function call is:

```
<function>(<arguments>)
```

When this expression appears in a program statement that is being executed, the function body executes, using the argument values as the values of the parameters.

In Listing 3.1, the function `myhypot()` is called in line 11. The arguments being **passed** or sent to `myhypot()` are the values of `a` and `b` at the time the program runs, after the `input()` expressions. The value of `a` will be used as the value of `x` in `myhypot()`, and the value of `b` will be used as `y`. For example, if you enter 4 and 7 in response to the `input()` expressions, then `x` will get the value 4 (the value of `a`) and `y` will get 7 (the value of `b`) when `myhypot()` is called on line 11.

Note: the name of an argument does *not* have to be the same as its corresponding parameter.

Function calls may be **nested** as in lines 9 and 10 of Listing 3.1. The inner function (in these cases, `input()`) runs first, and then the output of that function is immediately sent as the argument to the outer function (`float()`).

Return Statements

A `return` statement may optionally appear anywhere in the body of a function, and looks like this:

```
return <expression>
```

When this statement is executed inside of a function, it immediately terminates the function and returns the value of its `<expression>` as the value of the function call. In fact, the `<expression>` is also optional: if no expression is given, the value returned is the special value `None`.

In Listing 3.1, the `return` statement in line 6 computes and returns the length of the hypotenuse. The two steps of computing and returning could be separated:

```
def myhypot(x, y):
    hyp = sqrt(x ** 2 + y ** 2)
    return hyp
```

This version will work exactly the same as the original.

If a function does not contain a `return` statement, the function returns the value `None` when the last executable statement of its body finishes.

Local Variables

Variables used for the first time inside of a function definition are called **local** because their use is limited to within the definition of that function. In other words, attempts to access that variable from outside the function definition will fail. The **scope** of a variable is the part of the program in which the variable may be accessed, so the scope of a local variable in Python is the function definition in which it is used. Variables defined outside of all function definitions are called **global**; we will rarely use global variables in this text.

Parameters in a function definition have the same scope as local variables: they may not be accessed outside of the function definition.

In Listing 3.1, `a` and `b` are local variables in the `main()` function, whereas `myhypot()` does not have any local variables.

Type Conversions

The `input()` function always returns user input as a string. In order to process numeric input, we need to **convert** the string to either an integer or float. Python has the following built-in functions to convert the types we have seen so far:

`int(x)`	Convert `x` to an integer, truncating floats towards 0.
`int(x, b)`	Convert `x` given in base `b` to an integer.
`float(x)`	Convert `x` to a floating-point value.
`str(x)`	Convert `x` to a string.

For the input of numeric data, choose between `int()` and `float()` depending on the context. **Truncation** means that any non-integer fraction is dropped, so, for example, `int(3.975)` returns 3, and `int(-3.975)` is -3.

Testing

An important part of writing functions (and programs in general) is **testing** them to make sure that they compute exactly what you intend. Some basic principles of testing are:

Test all important families of inputs. For example, include both positive and negative input values if they are appropriate. Test with both integers and floating point values.

Test known cases. Use input values that give outputs you can predict.

Test boundaries. For example, be sure your function behaves correctly at minimum and maximum input values (when appropriate).

Testing is probably the most important tool we have to improve software quality.

Exercises

3.1 Use Listing 3.1 to answer these questions:

(a) Describe in your own words the subtask that the `myhypot()` function is designed to accomplish.

(b) Which line or lines of code define the body of the `main()` function?

(c) Is there a **`return`** statement inside `main()`? If not, when does `main()` return, and what value does it return?

(d) Which line of code contains a call to the `main()` function? What happens if you try to run the program with that line of code deleted? Explain the result.

3.2 Describe the result if line 6 of Listing 3.1 is replaced with the single line:

```
hyp = sqrt(x ** 2 + y ** 2)
```

Explain the behavior you observe.

3.3 Does the alternate version of the `myhypot()` function on page 15 contain any local variables? If so, identify them; if not, explain why not.

3.4 Use Listing 3.1 to answer these questions:

(a) List important families of inputs to test the `myhypot()` function. Test the program on values from each family and report the results.

(b) List two sets of test values for the `myhypot()` function that fall in the category of known cases, where you know ahead of time what the results should be. Test the program on those values and report the results.

(c) Does `myhypot()` have boundary cases to test? If it does, give them; if not, explain why not.

3.5 Modify Listing 3.1 to call `myhypot(a, b)` before the `print()` statement, storing the result in a new local variable, and then use the local variable in the `print()`. Discuss the tradeoffs.

3.6 Consider Listing 1.1 and its extension to include circumference in Exercise 1.2. Rewrite that version of `circle.py` to ask the user for the radius and use three functions for the area, circumference, and main program.

3.7 Rewrite Exercise 2.9 to ask the user for input and use two functions: one for computing the average and one for the main program.

3.8 Rewrite Exercise 2.10 to ask the user for input and use three functions for the area, perimeter, and main program. How many parameters will the `area()` and `perimeter()` functions need?

3.9 Rewrite Exercise 2.11 to ask the user for input and use three functions for the volume, surface area, and main program.

3.10 Rewrite Exercise 2.12 to ask the user for input and use three functions for the volume, surface area, and main program.

3.11 Write a program `annulus.py` that asks the user for an inner and outer radius, and then calculates and prints the area of an annulus with those dimensions. (Look up annulus if you do not know what one is.) Use a function to calculate the area of an annulus that itself calls a function that computes the area of a circle.

3.12 Write a program `shell.py` that calculates and prints the volume of a spherical shell, given its inner and outer radii. Use a function to calculate the voume of a shell that itself calls a function that computes the volume of a sphere.

3.13 Write a program `temp.py` that asks the user for a temperature in degrees Fahrenheit, and then calculates and prints the corresponding temperature in degrees Celsius. Use a function to perform the conversion.

3.14 Modify the previous exercise to also compute and display the equivalent temperature in degrees Kelvin. Use a separate conversion function (or two).

3.15 Write a program `heart.py` that asks for a person's age in years y and then estimates his or her maximum heart rate in beats per minute using the formula $208 - 0.7y$. Use appropriate functions.

3.16 Write a program `heron.py` that asks the user for the lengths of the sides of a triangle (a, b, and c) and then computes the area of the triangle using Heron's formula. (Look up Heron's formula if you do not remember it.) Use appropriate functions.

3.17 Write a program `interest.py` to calculate the new balance of a savings account if interest is compounded yearly. Ask the user for the principal, interest rate, and number of years, and then display the new balance. Use the formula $P(1+r)^t$ and appropriate functions.

Enter interest rates as decimals: for example, a rate of 4% should be entered as 0.04. Use your program to determine the number of years it takes a principal to double with interest rate 1.5%.

3.18 Write a program `compound.py` to calculate the new balance of a savings account if interest is compounded n times per year. Ask the user for the principal, interest rate, number of years, and number of compounding periods per year, and display the new balance. Use the formula $P(1+\frac{r}{n})^{nt}$ and appropriate functions.

Use your program to determine the difference over 20 years on a $1000 balance between compounding yearly and compounding monthly at 8% interest.

3.19 Write a program `vp.py` that asks the user for the temperature t in degrees Celsius and then displays the estimated vapor pressure of water vapor in millibars at that temperature using the approximation

$$6.112e^{\frac{17.67t}{t+243.5}}$$

Use appropriate functions. Use your program to estimate the temperature at which the vapor pressure is approximately 10 mb.

4.

Repetition: For Loops

To this point, our Python programs have been limited to tasks that could be accomplished by a calculator. **Repetition**, the ability to repeat any section of a program, adds a surprising amount of new power.

Listing 4.1: Harmonic Sum

```python
# harmonic.py

def harmonic(n):
    # Compute the sum of 1/k for k=1 to n.
    total = 0
    for k in range(1, n + 1):
        total += 1 / k
    return total

def main():
    n = int(input('Enter a positive integer: '))
    print("The sum of 1/k for k = 1 to", n, "is", harmonic(n))

main()
```

This program computes the **harmonic sum**

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Type it in and run it a few times to get a feeling for what it does and how it works. Try to follow the steps that are executed. This program contains several new features, including the **`for`** loop, the `range()` function, and the use of `+=`. We will examine each of these components in turn.

For Loops

A **`for`** loop is one of the two main tools in Python that allow programs to repeat. (The other is **`while`**, which you will see in Chapter 8.) They are most useful when you know ahead of time how many times the loop needs to run.

The syntax of a **for** loop looks like this:

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

The **for loop** works in this way: for each item in the `<sequence>`, the `<variable>` is assigned to have the value of that item and then the loop `<body>` is executed. Thus, the loop will be executed once for each item in the sequence.

A **sequence** in Python is an ordered set of elements. The simplest type of sequence is called a **list**, in which the elements are listed inside square brackets. We will study lists in depth beginning in Chapter 14.

For example, this loop:

```
for i in [8, 3, 0]:
  print(i, i**2)
```

will produce this output:

```
8 64
3 9
0 0
```

The loop body in this case executes three times, once each with `i` having the value 8, 3, and then 0.

Range

The built-in `range()` function is often used inside Python **for** loops instead of writing out long lists. Technically, `range()` returns an **iterable**, which provides the precise thing a **for** loop expects in the place we have referred to as a sequence.

The `range()` function may be called in three different ways. In all three versions, the parameters must be integers.

`range(stop)`	Begin at 0. Take steps of size 1. End just before `stop`.
`range(start, stop)`	Begin at `start`. Take steps of size 1. End just before `stop`.
`range(start, stop, step)`	Begin at `start`. Take steps of size `step`. End just before `stop`.

$\Longrightarrow$ Caution: These are not intuitive at first. Ranges start at 0 unless you specify otherwise, and they end *before* the stopping point you provide.

To see the elements that a `range()` will iterate over, you can use the `list()` type converter:

`list(x)`	Convert `x` to a list.

For example,

```
list(range(4)) = [0, 1, 2, 3]
list(range(2, 4)) = [2, 3]
list(range(2, 10, 3)) = [2, 5, 8]
list(range(5, 2, -1)) = [5, 4, 3]
```

Notice that with a step size of -1, the sequence still stops *one before* the `stop` we provided.

As you learn more Python, you will find that the `range()` function is written in this way for good reasons.

Assignment Shorthands

We still need to explain the `+=`, although you may have figured it out by now. It is simply a shorthand for a common type of assignment statement:

`x += y`	Equivalent to `x = x + y`.
`x -= y`	Equivalent to `x = x - y`.
`x *= y`	Equivalent to `x = x * y`.
`x /= y`	Equivalent to `x = x / y`.

Accumulation Loops

Listing 4.1 contains an example of an **accumulation loop**, where one of the variables, known as the **accumulator**, gradually accumulates some quantity as the loop runs. In this case, the accumulator is the variable `total`. It begins with the value 0 in line 5, and then each time the loop runs, the code in line 7 adds a bit more to the total.

All accumulation loops look something like this:

```
<accumulator> = <starting value>
loop:
    <accumulator> += <amount to add>
```

After the loop finishes, `<accumulator>` contains the accumulated value. Accumulation loops may also accumulate via multiplication rather than addition. For example, compare Exercise 4.16 with Exercise 4.17.

$\Longrightarrow$ Caution: Do not use `sum` as an accumulator variable name, because it is a built-in Python function.

Runaway Loops

Now that we have programs that repeat, we may write a program that runs much too long. If you need to stop a program, use CTRL-C or look for an option to restart your interpreter or shell.

Exercises

4.1 Use Listing 4.1 to answer these questions:

(a) Explain the apparent purpose of comments like the one in line 4.

(b) Explain the use of `n + 1` instead of `n` in line 6.

(c) Give two test values of `n` for which you can easily predict the correct output. Give the outputs for those cases.

(d) Conjecture what happens to the harmonic sum for very large `n`.

4.2 List all local variables used in Listing 4.1 and describe the scope of each.

4.3 Determine the elements that will be iterated over for each of these `range` expressions:

(a) `range(10)`

(b) `range(5, 10)`

(c) `range(10, 5)`

(d) `range(3, 10, 2)`

(e) `range(10, 0, -1)`

(f) `range(10, 0, -2)`

(g) `range(0, 10, -1)`

(h) `range(0, 1, 0.1)`

4.4 Determine a `range` expression to iterate over each of these sequences:

(a) `0, 1, 2, 3`

(b) `3, 2, 1, 0`

(c) `1, 3, 5, 7, 9, 11`

(d) `10, 27, 44, 61, ..., 197`

(e) `1000, 900, 800, ..., 100`

(f) `2, 4, 6, 8, ..., 200`

4.5 Write a program `quote.py` that prints a short quote of your choice exactly one thousand times. You do not need to use any functions other than `main()`.

4.6 Write a program `powers.py` that prints a table of values of n and 2^n for $n = 1, 2, \ldots, 10$. Do these values look familiar? You do not need to use any functions other than `main()`.

4.7 Write a program `table.py` that prints a table of values of n, $\log n$, $n \log n$, n^2, and 2^n for $n = 10, 20, \ldots, 200$. The `log` function is in the `math` module. You do not need to use any functions other than `main()`. Does the function $n \log n$ grow more like n or more like n^2? How would you describe the growth of 2^n? What about $\log n$?

4.8 Write a program `circletable.py` that prints a table of the areas of circles of radius $r = 1, 2, \ldots, 10$. Use an area function.

4.9 Write a program `temptable.py` that prints a table of Fahrenheit to Celsius conversions for temperatures between $-30°F$ and $100°F$ at 10 degree intervals. Use a conversion function.

4.10 Write a program `disttable.py` that prints a table of mile to kilometer conversions for distances between 100 and 1500 miles at 100 mile intervals. Write a function to do the conversion. One mile is approximately 1.609 km.

4.11 Modify Exercise 3.15 to write a program `hearttable.py` that prints the age and maximum heart rate for ages between 20 and 60 at 2 year intervals.

4.12 Modify Exercise 3.17 to write a program `interesttable.py` that prints the balance in an account earning simple interest at the end of each year for a given number of years. Ask the user for the starting principal, interest rate, and number of years.

4.13 Rewrite the function from Exercise 3.17 that computes interest compounded yearly to use an accumulator rather than the direct formula.

4.14 Modify Exercise 3.19 to write a program `vptable.py` that prints the estimated vapor pressures for temperatures between -20 and 50 degrees Celsius at 5 degree intervals.

4.15 Write a program `basel.py` that modifies Listing 4.1 to print a table of values of $\displaystyle\sum_{k=1}^{n} \frac{1}{k^2}$ for $n = 10, 100, 1000, \ldots, 10^7$. Use your program to conjecture what happens to this sum as n becomes large. Hints: Use a `range()` to create the exponents, and in addition to the sum, also print the square root of six times the sum.

4.16 Define a function `triangular(n)` that returns the sum $1+2+3+\cdots+n$. Then write a program `triangular.py` that prints a table of values of `triangular(n)` for $n = 1, 2, 3, \ldots, 20$.

4.17 Write a function `myfactorial(n)` that returns the product $1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$. Use an accumulator; do not use the `factorial()` function from the `math` module. Then write a program `factorial.py` that prints a table of values of `myfactorial(n)` for $n = 1, 2, 3, \ldots, 20$. Finally, modify your program to compare the growth of factorials with powers (like n^2) and exponentials (like 2^n). Discuss your results.

4.18 Write a function `pyramid(n)` that returns $1^2 + 2^2 + 3^2 + \cdots + n^2$. Then write a program `pyramid.py` that prints a table of values of `pyramid(n)` for $n = 1, 2, \ldots, 20$. Finally, modify your program to estimate the growth rate of the `pyramid()` function. Does it appear similar to a power function or an exponential? Discuss your results.

5.

Computer Memory: Integers

It is time to go back to the machine level and look a little more deeply at memory. At a high level, computer memory may be categorized according to concepts such as access speed and size. Definite patterns emerge:

Type	**Access Speed**	**Proximity to CPU**	**Size**	**Volatile**
Register	Fastest	Inside	10's	Y
Caches	Very fast	Adjacent	100's to MB	Y
RAM	Fast	Near	GB	Y
Hard disks	Slow	Far	TB	N

At a lower level, the question is, how is data actually stored in computer memory? In this chapter, we begin to develop an answer by looking at how integers are stored.

Bits and Bytes

Computer memory of all types may be thought of at different levels of interpretation. The bottom level is electronics and physics, which is taught in those courses. We will move up one level of interpretation and begin by thinking of memory as a sequence of electronic on/off switches. Each on/off switch is called a **bit**. A group of 8 bits is called a **byte**.

In abbreviations, a small "b" refers to bits, while capital "B" refers to bytes. So, for example, Mbps refers to megabits per second, while GB refers to gigabytes.

Binary Numbers

We then interpret bits as numbers by thinking of "off" as 0 and "on" as 1. For example,

off	on	on	off	on	off	off	off
0	1	1	0	1	0	0	0

We interpret this number as a number in base two instead of base ten. Now, the numbers that you use every day are **decimal** or **base ten**. Think about how they work:

$$\begin{array}{cccc} \boxed{2} & \boxed{1} & \boxed{7} & \boxed{4} \\ 1000\text{'s} & 100\text{'s} & 10\text{'s} & 1\text{'s} \\ 10^3 & 10^2 & 10^1 & 10^0 \end{array} = 2*1000+1*100+7*10+4*1$$

Binary numbers are **base two** and work the same way except that instead of powers of ten, they use powers of two. Note that with base ten, we use the digits 0–9 (less than ten); with base two, we only use the digits 0 and 1 (less than two). For example,

$$\begin{array}{cccc} \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} \\ 8\text{'s} & 4\text{'s} & 2\text{'s} & 1\text{'s} \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array} = 1*8+0*4+1*2+1*1=11$$

Hexadecimal Numbers

Hexadecimal numbers are **base sixteen** and work the same way with powers of sixteen and digits 0–9, A=10, B=11, C=12, D=13, E=14, and F=15. Every four bits can be thought of as a single hexadecimal "digit," since four bits can hold values between 0 and 15. Thus, the byte 01101100 can be represented as `6C` hexadecimal, since 0110 equals 6 and 1100 equals 12, which is `C`.

Python has built-in conversion functions that you may find useful:

`bin(n)`	Binary value of integer `n` (as a string).
`hex(n)`	Hex value of integer `n` (as a string).

Binary strings begin with `"0b"` in Python, while hex strings begin with `"0x"`.

Storing Integers

When a computer is described as "32-bit" or "64-bit," that tells you the basic memory size used by its CPU. Generally, this also gives the size of memory that is used to store an integer. **Unsigned integers** are always greater than or equal to zero and are stored in binary, as described above, using the number of bits given by the architecture. For example, on a 32-bit machine, the unsigned integer 2012 is stored as `0x000007dc`.

Signed integers, on the other hand, may be either positive or negative and so require a more complicated representation. Most computer systems use **two's complement**, which is taught in computer architecture courses.

Memory Sizes

Memory sizes are generally given in terms of bytes, except that so many bytes are involved that usually a prefix is used to indicate the scale at which we are working. Common prefixes are borrowed from the metric system, such as kilo-, mega-, giga-, and tera-. Unfortunately, these prefixes do not quite mean what they do in the metric system, where they are based on powers of 10:

Prefix	**Value**
kilo-	$10^3 = 1000 = 1$ thousand
mega-	$10^6 = 1{,}000{,}000 = 1$ million
giga-	$10^9 = 1{,}000{,}000{,}000 = 1$ billion
tera-	$10^{12} = 1{,}000{,}000{,}000{,}000 = 1$ trillion

Because computer memory is based on bits, powers of 2 are often used for these sizes instead:

Prefix	**Value**
kilo-	$2^{10} = 1024$
mega-	$2^{20} = 1{,}048{,}576$
giga-	$2^{30} = 1{,}073{,}741{,}824$
tera-	$2^{40} = 1{,}099{,}511{,}627{,}776$

These are close to their metric equivalents but are not exactly the same. This is one reason computer memory is sold in quantities that sometimes look strange.

Notice that $2^{10} = 1024 \approx 1000$. This is a useful fact to help you remember the size of powers of 2.

Exercises

5.1 Using either your own computer or one in a lab, determine the number of registers in its CPU, as well as the size of its caches, RAM, and hard drive (or other long-term storage).

5.2 Give the largest binary value that can be stored in one unsigned byte, along with its decimal and hexadecimal equivalents.

5.3 Give the largest binary value that can be stored in two unsigned bytes, along with its decimal and hexadecimal equivalents.

5.4 Determine the largest unsigned integer that can be stored in a 32-bit machine.

5.5 Determine the largest unsigned integer that can be stored in a 64-bit machine.

5.6 Give the values of these Python expressions:

(a) `hex(25)`
(b) `bin(35)`
(c) `int("0x1C", 16)`
(d) `int("0b10101", 2)`
(e) `int(bin(1000), 2)`
(f) `int(hex(1000), 16)`

5.7 Show how each of these is stored as an unsigned integer in a byte. Write both binary and hexadecimal forms.

(a) 87
(b) 195
(c) 18
(d) 119
(e) 93
(f) 234

5.8 Convert these unsigned binary integers to decimal and hexadecimal.

(a) `0b10010011`
(b) `0b00101101`
(c) `0b01001011`
(d) `0b10011110`
(e) `0b01011100`
(f) `0b11000001`

5.9 Convert these unsigned hexadecimal integers to binary and decimal.

(a) `0x7D`
(b) `0xA1`
(c) `0x59`
(d) `0xBC`
(e) `0x96`
(f) `0x04`

5.10 Write a short program `binhex.py` that prints a table of binary and hexadecimal values for the (decimal) integers 1 through 100. You do not need any functions other than `main()`.

6.

Selection: If Statements

At both high levels and at the machine level, programs execute statements one after the other. **Selection** statements allow a program to execute different code depending on what happens as the program runs. This flexibility is another key to the power of computation.

Listing 6.1: Centipede

```
# centipede.py

from turtle import *

def centipede(length, step, life):
    penup()
    theta = 0
    dtheta = 1
    for i in range(life):
        forward(step)
        left(theta)
        theta += dtheta
        stamp()
        if i > length:
            clearstamps(1)
        if theta > 10 or theta < -10:
            dtheta = -dtheta
        if ycor() > 350:
            left(30)

def main():
    setworldcoordinates(-400, -400, 400, 400)
    centipede(14, 10, 200)
    exitonclick()

main()
```

This program may be harder to read at first than previous examples. It uses what are called “turtle graphics,” explained later in this chapter. Run it, and then you will begin to see what the code is doing. Use “Restart Shell” from the Shell menu in IDLE if your window ever becomes unresponsive.

The new features in this program include the **if** statement, boolean expressions, and the turtle graphics library. In order to describe **if** statements, we need to begin with boolean expressions.

Boolean Expressions

Python has an additional data type known as **boolean**. Variables that hold this type have only two possible values: `True` or `False`. Generally, you will use boolean values in expressions rather than variables. Thus, a **boolean expression** is an expression that evaluates to either `True` or `False`. Python has the following **comparison operations** that return boolean values:

`x == y`	Equal.
`x != y`	Not equal.
`x < y`	Less than.
`x > y`	Greater than.
`x <= y`	Less than or equal.
`x >= y`	Greater than or equal.

⟹ Caution: To test for equality in Python, you must use `==` rather than `=`.

⟹ Caution: Avoid using `==` or `!=` to compare floats; instead, try to use an inequality if you can. The reason is that floating-point calculations are not always exact. See Chapter 9 for an explanation.

Boolean expressions may be combined with these **boolean operations**:

`P` **and** `Q`	`True` if both `P` and `Q` are `True`; otherwise, `False`.
`P` **or** `Q`	`True` if either `P` or `Q` (or both) are `True`; otherwise, `False`.
not `P`	`True` if `P` is `False`; otherwise, `False`.

The **and** and **or** operations **short-circuit** their evaluation, meaning, for example, that when evaluating `P and Q`, if `P` is `False`, then there is no need to evaluate `Q` because the result must be `False`.

If Statements

Almost all programming languages (including machine languages) have some form of **if** statement. In Python, the statement has essentially three forms. The simplest is:

```
if <boolean>:
    <body>
```

In this form, the `<boolean>` expression is evaluated, and if it is `True`, then `<body>` is executed. If the expression is `False`, then `<body>` is not executed.

An **if** statement may contain an optional **else** clause, which contains alternative code to run when the boolean expression is `False`:

```
if <boolean>:
    <body1>
else:
    <body2>
```

In this case, `<body1>` is executed if the boolean expression is `True`; otherwise, `<body2>` is executed.

Finally, a sequence of tests may be checked by using the **elif** option:

```
if <boolean1>:
    <body1>
elif <boolean2>:
    <body2>
elif <boolean3>:
    <body3>
...
else:
    <bodyN>
```

Here, if `<boolean1>` is `True`, then `<body1>` is executed; otherwise, `<boolean2>` is evaluated, and if it is `True`, `<body2>` is executed; and so on. Later tests are checked only if all preceding tests are `False`.

Python Turtle Module

Turtle graphics is a type of computer graphics that draws relative to the position of a "turtle" on the screen. The turtle holds a pen, and if the pen is down when the turtle moves, then a line will be drawn. The turtle may also move with the pen up or initiate other types of drawing such as "stamping" its own shape or drawing dots.

Listing 6.1 uses the following functions from the Python `turtle` module:

`penup()`	Stop drawing turtle path.
`forward(x)`	Move forward `x`.
`left(theta)`	Turn left angle `theta` (default is degrees).
`stamp()`	Draw turtle shape at current location.
`clearstamps(n)`	Delete first `n` stamps (if `n` is positive).
`ycor()`	y coordinate of current location.
`setworldcoordinates(llx, lly, urx, ury)`	Set lower-left and upper-right coordinates.
`exitonclick()`	Close turtle window when clicked.

Using the Python Documentation

You can imagine that the `turtle` module must provide many other functions in addition to those listed above. You may also have questions about exactly how those functions work. The **Python documentation** is online, extensive, and provides information like this and much more.

From the "Documentation" link at `http://www.python.org`, two links will be particularly useful: the Tutorial and the Library Reference.

Tutorial provides informal descriptions of how most things work in Python. Use it when you start to learn a new topic.

Library Reference is a good place to look up specific reference information. For example, at the time of this writing, the complete list of functions in the `turtle` module is in Section 23.1 of the Library Reference for Python 3.2.

Be sure to use the documentation set that matches your version of Python.

Import Star

Python provides a star form of the **`import`** statement that is helpful when you need to use many names from the same module:

```
from turtle import *
```

This imports all names from the module `turtle`. Use the star sparingly; otherwise, you may find that a module has imported names that you didn't anticipate.

Multiple Return Values

Occasionally, it is useful to have a function return more than one value:

```
return <expression1>, <expression2>, ...
```

The mechanism used to accomplish this will be described later in Chapters 16 and 19.

Exercises

6.1 Explain the difference between = and == in Python.

6.2 Evaluate these boolean expressions:

(a) `1.4 ** 2 > 2`

(b) `6 // 7 == 1 // 7`

(c) `10 >= 11 or 11 < 12`

(d) `3 > 1 and 4.25 > 9 / 2`

6.3 Write the mathematical condition "x is in the interval $(2, 3]$" as a Python boolean expression.

6.4 Describe the circumstances when the **`or`** operation can short-circuit, and briefly explain why that behavior is correct.

6.5 List all local variables used in Listing 6.1 and describe the scope of each.

6.6 Determine the range of x and y values for the turtle screen in Listing 6.1. Use `print(xcor(), ycor())` to verify your answer.

6.7 Determine the initial location and orientation of the turtle in a turtle graphics program.

6.8 Describe the effect of changing each of these quantities in Listing 6.1:

(a) `length`

(b) `step`

(c) `life`

(d) `dtheta`

(e) The 350 in line 18

(f) The 10 and −10 in line 16

6.9 Is it possible to turn right using the `turtle left()` function? Explain why or why not.

6.10 Use Listing 6.1 to answer these questions:

(a) Describe the effect of removing the **`if`** statement (and its body) at line 14. Explain the result.

Hint: an easy way to do this is to put comment symbols at the beginning of those two lines, thereby changing the code into comments. This is called **commenting out** a section of code.

(b) Describe the effect of removing the **`if`** statement (and its body) at line 16. Explain the result.

(c) Describe the effect of removing the **`if`** statement (and its body) at line 18. Explain the result.

6.11 Modify Listing 6.1 to keep the pen down rather than up. Describe the effect.

6.12 Modify line 16 of Listing 6.1 to use the built-in absolute value function `abs()` instead of an **`or`** expression.

6.13 Modify Listing 6.1 to prevent it from escaping across all four sides.

6.14 Modify Listing 6.1 to produce an interesting different behavior of your choice.

6.15 Write an interesting turtle graphics program of your choice.

6.16 Write a function `grade(score)` that returns the corresponding letter grade for a given numerical score. Use 90 or above for an A, 80 for a B, etc. Write a `main()` that tests your function.

6.17 Write a function `mymax2(x, y)` that returns the larger of `x` and `y`. Do not use the built-in Python function `max()`. Write a `main()` that tests your function.

6.18 Write a function `mymax3(x, y, z)` that returns the largest of `x`, `y`, and `z`. Do not use the built-in Python function `max()`. Write a `main()` that tests your function.

6.19 Write a function `median3(x, y, z)` that returns the middle value among `x`, `y`, and `z`. (If two of the values happen to be the same, that value is the median.) Do not use any built-in Python sorting functions. Write a `main()` that tests your function.

6.20 Write a function `sort3(x, y, z)` that returns the three values `x`, `y`, and `z` in **sorted order** `(a, b, c)`, where $a \leq b \leq c$. Do not use any built-in Python sorting functions, but you may put **`if`** statements inside of other **`if`** statements. Write a `main()` that thoroughly tests your function.

6.21 Rewrite the function `median3(x, y, z)` of Exercise 6.19 using the function `sort3()` from Exercise 6.20. Use multiple assignment (see page 95) to store the return values of `sort3()`.

6.22 Write a function `myabs(x)` that returns $|\,x\,|$, the absolute value of `x`, which is given by:

$$|\,x\,| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

Do not use the built-in Python function `abs(x)`. Write a `main()` that tests your function.

6.23 Rewrite the `myfactorial(n)` function from Exercise 4.17 to use this recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

You may assume `n` is not negative. Write a `main()` that tests your function.

6.24 Write a function `solvequadratic(a, b, c)` that returns the solution(s) of the quadratic equation $ax^2 + bx + c = 0$. Use the discriminant $d = b^2 - 4ac$ to determine whether there are 0, 1, or 2 real roots. Python has a built-in `complex()` function if you want to return complex roots.

6.25 Write a function `zone(age, rate)` that returns a description of a person's training zone based on his or her age and training heart rate, `rate`. The zone is determined by comparing `rate` with the person's maximum heart rate `m`:

	Training Zone
$\texttt{rate} \geq .90\,\texttt{m}$	interval training
$.70\,\texttt{m} \leq \texttt{rate} < .90\,\texttt{m}$	threshold training
$.50\,\texttt{m} \leq \texttt{rate} < .70\,\texttt{m}$	aerobic training
$\texttt{rate} < .50\,\texttt{m}$	couch potato

Use the function from Exercise 3.15 to determine `m`. Write a `main()` to ask the user for input and display the result.

7.

Algorithm Design and Debugging

Developing Algorithms

As the problems we tackle become more complex, it becomes harder to quickly write out programs to solve them, and we need to devote correspondingly more time to planning and designing solutions as opposed to writing code. At this stage, the difficulty is generally in designing an algorithm capable of solving the problem.

An **algorithm** is just a specific sequence of steps that will solve some problem. Prior to writing code, programmers often use **pseudocode**, which is a hybrid of English (or other language) and programming code. Thinking in pseudocode allows you to focus on how to solve the problem rather than language details or syntax. However, to be useful, the pseudocode must be specific enough that it can be translated into a working program.

Recipes are good everyday examples of algorithms. They can be written at different levels of detail, depending on the expertise or experience of the cook. What can be difficult about programming is that the computer almost always requires a *more precise* level of detail than we are used to providing.

The following guidelines may be helpful as you begin to design algorithms:

Know what the language can do. To some extent, you have to limit your thinking to what a program can do; otherwise, you may write pseudocode that cannot be translated into a program.

Recognize patterns. For example, accumulation loops occur in many different contexts (as you will see). Patterns like it expand the vocabulary you can think in.

Think top-down or bottom-up. Top-down design starts with large process steps and gradually breaks each one down until there is enough detail to implement it. Bottom-up works in reverse, beginning with relatively detailed tasks, and then putting those together until there is a complete solution. Combining top-down with bottom-up thinking can be quite powerful.

Comparing Algorithms

Once we have an algorithm, how do we know if it is a good one? The main criterion, whether we are beginners or professionals, is the same:

Does it work?

This simple question hides enormous complexity, but keep it firmly in mind.

Beyond the question of producing correct results, algorithms and programs are also judged on:

Efficiency. Does the program run in a reasonable amount of time? Could it be faster?

Space. Does the program use a reasonable amount of memory, or is it unnecessarily taxing system resources?

Elegance. Does the program represent an elegant solution to the problem, or does it feel like a maze with no exit?

Courses in data structures and algorithm analysis explore these questions in more depth.

Debugging

As programs become more complicated, it also becomes more difficult to analyze their behavior, particularly when problems arise. **Debugging** is the art of finding and removing errors or **bugs** in programs.

Arts often sound like mysteries, especially when you are a beginner in the field, but there is also a science to debugging. The science involves becoming *systematic* rather than random when trying to understand program behavior.

$\Longrightarrow$ Caution: Random changes to programs produce random results.

The key to systematic debugging is to take advantage of the fact that programming is one of the few disciplines that offers quick and accurate feedback to your problem-solving efforts. Imagine a math problem that told you whether you had solved it or not—essentially, that is what a computer does every time you run a program.

Here is one way[1] to systematically debug a program:

Experience what the program does when it runs. In other words, *pay attention* to the program's behavior. This step sounds obvious, but it is easy to overlook and is a necessary precondition to making intentional progress. If your program does not run, the Python interpreter should help you find syntax errors. Fix those first.

[1]This outline is based on James Zull's *The Art of Changing the Brain* [6], which gives these steps as an outline for learning. The power of this particular model is that the outline is based on brain structure and how the brain seems to process information.

Reflect on the behavior you observe. Take a minute to make sure you understand what you have seen. At the end of this step, you should be able to give a precise description of the (incorrect) behavior of your program in your own words.

Hypothesize what might be causing the program's current behavior. This is the key step. You need to discover *why* the program is doing what it is doing. That will often tell you how to fix it.

Test your hypothesis. Make an intentional change (not a random one), and see if the program responds in the way you expect. One of the easiest ways to test an idea is to insert extra `print()` statements that let you see the values of important variables.

Repeat until the program works.

Practice these steps regularly, and you will develop effective debugging habits.

Exercises

7.1 Write an algorithm to make a favorite food for:

(a) A younger sibling who knows the food but is new to cooking

(b) A college student who does not know the food but cooks regularly

7.2 Write an algorithm to get to some location in your home town for:

(a) A local friend

(b) A friend from out of town

7.3 You have a list of numbers which provide access to one element at a time. Write an algorithm in pseudocode to:

(a) Find the smallest number in the list.

(b) Find the largest number in the list.

(c) Sort the list in increasing order.

7.4 Research the bug associated with Grace Hopper. Summarize and discuss what you find.

8. Repetition: While Loops

While loops allow programs to repeat without necessarily knowing ahead of time how many times the loop will run. Like selection, this allows programs to adapt and run more or less code, depending on the circumstances.

Consider the task of printing a table of prime numbers. An integer n is **prime** if it is greater than 1 and its only positive divisors are 1 and itself. This definition leads to the following program:

Listing 8.1: Prime Numbers

```
# primes.py

def isprime(n):
    # Return True if n is prime.
    return n > 1 and smallestdivisor(n) == n

def smallestdivisor(n):
    # Find smallest divisor (other than 1) of integer n > 1.
    k = 2
    while k < n and not divides(k, n):
        k += 1
    return k

def divides(k, n):
    # Return True if k divides n.
    return n % k == 0

def main():
    for n in range(2, 100):
        if isprime(n):
            print(n, end=" ")
    print()

main()
```

The definition of the `isprime()` function parallels the definition of prime number in a nice way and reads like pseudocode. In addition to the **`while`** loop, this program uses the modulus operator `%` and the `end=` option of `print()` for the first time.

While Loops

The syntax of a **`while`** loop is almost identical to that of an **`if`** statement.

```
while <boolean>:
    <body>
```

The `<boolean>` expression is evaluated, and if it is `True`, then `<body>` is executed. (So far, this is identical to an **`if`** statement.) After the body executes, `<boolean>` is evaluated again, and if it is still `True`, then the body executes again. This is repeated until the boolean expression is `False`.

Unlike a **`for`** loop, it is distinctly possible for a **`while`** loop to be **infinite**. Use CTRL-C or look for an option to restart your interpreter or shell if you execute an infinite loop.

Mod Operation

The **modulo** or **mod** operation finds the remainder when one integer is divided by another.

`n % m`	Remainder when `n` is divided by `m`.

So, for example, `k` divides `n` evenly if the remainder is 0; i.e., if `n % k == 0`.

Print Options

The Python `print()` function has two options that give you somewhat more control over output:

`print(..., end=s)`	Print with string `s` at end instead of newline.
`print(..., sep=s)`	Print with string `s` between expressions.

Both options may be set at the same time.

Tracing Code by Hand

You may have found it hard to follow exactly what is happening in Listing 8.1. One reason might be its use of several functions, another could be the **`while`** loop in `smallestdivisor()`. Tracing code by hand can help overcome both of these difficulties. When we **trace** code by hand, we try to simulate on paper what the interpreter is asking the CPU to do.

For example, we might trace the evaluation of `isprime(9)` like this:

$$\begin{array}{l}
\texttt{isprime(9)} \\
\qquad \to \texttt{smallestdivisor(9)} \\
\qquad\qquad \to \texttt{divides(2, 9)} \quad \text{returns } \boxed{\text{F}} \\
\qquad\qquad \to \texttt{divides(3, 9)} \quad \text{returns } \boxed{\text{T}} \\
\qquad \text{returns } \boxed{3} \\
\text{returns } \boxed{\text{F}}
\end{array}$$

To follow the operation of `smallestdivisor()`, it may also be helpful to track the values of its variables, like this for the call `smallestdivisor(25)`:

n	k	
25	2	
	3	
	4	
	5	Returns $\boxed{5}$

There is nothing special about these ways of writing the traces; the point is just to find a helpful way of simulating the program's execution.

Nested Loops

Although it is not immediately obvious, Listing 8.1 runs its **`while`** loop *inside* of another loop—the **`for`** loop in `main()`. That means the entire **`while`** loop from `smallestdivisor()` runs once every time the body of the **`for`** loop runs.

Any time one loop is run inside of another loop, the loops are called **nested**. As you may imagine, nested loops can have a significant impact on program performance.

Exercises

8.1 Calculate these values by hand:

(a) `23 % 4`
(b) `15 % 7`
(c) `52 % 19`
(d) `219 % 105`
(e) `1738 % 3`
(f) `418 % 2`
(g) `5033 % 2`
(h) `128 % 10`
(i) `741 % 5`

8.2 Trace the execution of each of these function calls:

(a) `isprime(7)`
(b) `isprime(27)`
(c) `isprime(1024)`
(d) `smallestdivisor(100)`
(e) `smallestdivisor(81)`
(f) `smallestdivisor(49)`

8.3 Write a function `is_even(x)` that returns the boolean value `True` if `x` is even; otherwise, it returns `False`.

8.4 Write a function `is_odd(x)` that returns the boolean value `True` if `x` is odd; otherwise, it returns `False`.

8.5 Determine the output of these two fragments of Python code. Which of the two has nested loops? How many `print()` statements are executed in each case?

(a)
```
for i in range(4):
    print(i)
for j in range(3):
    print(j)
```

(b)
```
for i in range(4):
    print(i)
    for j in range(3):
        print(j)
```

8.6 Write an infinite loop.

8.7 Use Listing 8.1 to answer these questions:

(a) Explain the purpose and effect of the `end=" "` in line 21. What happens if it is omitted?
(b) Explain the purpose and effect of the `print()` statement in line 22. What happens if it is omitted?
(c) Determine the value of the function call `smallestdivisor(1)`.
(d) Is there any way for the loop in `smallestdivisor()` to be infinite? Explain your answer.
(e) Discuss the tradeoffs in having `divides()` as a separate function.

8.8 Consider this variation of the `divides()` function from Listing 8.1:

```
def divides(k, n):
    if n % k == 0:
        return True
    else:
        return False
```

Discuss the tradeoffs between writing the function in this way compared with the original version.

8.9 Modify the `smallestdivisor()` function in Listing 8.1 to use a **for** loop instead of a **while** loop. Discuss the tradeoffs. Hint: a function may **return** at any time.

8.10 If an integer n is not prime, then at least one of its divisors must be less than or equal to $\sqrt{n}$. (Think about why.) Use this fact to improve the performance of the `smallestdivisor()` function in Listing 8.1.

8.11 Write a program `change.py` that asks the user for a number of cents and computes the number of dollars, quarters, dimes, nickels, and pennies needed to make that amount, using the largest denomination whenever possible. For example, 247 cents is 2 dollars, 1 quarter, 2 dimes, and 2 pennies. Use `//` when you intend to use integer division. You do not need to write any functions other than `main()`.

8.12 Write a program `guesser.py` that guesses an integer chosen by the user between 1 and 100. After each guess, the user indicates if the guess was too high, too low, or correct. Your program should use as few guesses as possible. You do not need to write any functions other than `main()`.

8.13 Modify `guesser.py` from Exercise 8.12 to add these features:

(a) Allow the user to play more than once.

(b) Report the total number of guesses taken after each round.

(c) Allow the user to specify the upper limit, instead of always using 100.

(d) Have your program sound confident by predicting the maximum number of guesses it will take. The `log()` and `ceil()` functions from the `math` module may be helpful.

8.14 Write a program `savings.py` that calculates the number of years it will take to reach a savings goal given a starting principal and interest rate. Write appropriate functions, ask the user for input, and display the result. Hint: use an accumulator rather than a formula.

8.15 Write the function `intlog2(n)` that returns the largest integer k such that $2^k \leq n$. For example, `intlog2(20) = 4` because $2^4 = 16$ is the largest power of 2 less than or equal to 20, and `intlog2(32) = 5` because $2^5 = 32$ is the largest power of 2 less than or equal to 32. Write a `main()` to test your function. Do not use any library functions inside your `intlog2()`.

8.16 Modify Listing 4.1 to ask the user for a number `m` and then compute the smallest `n` for which the harmonic sum $\sum_{k=1}^{n} \frac{1}{k}$ is greater or equal to `m`. Be careful to test your program only with small `m`.

8.17 Write a function `gcd(m, n)` that calculates the **GCD** (greatest common divisor) of `m` and `n`, which is the largest positive `k` that divides both `m` and `n`. Use **Euclid's algorithm** to calculate the gcd. Here is one (very succinct!) way to describe it:

```
Replace m with n, and n with m % n until n is 0.
```

Once `n` becomes 0, the gcd is in `m`. Write a program `gcd.py` with a `main()` that uses nested loops to test your function thoroughly.

Project: Newton's Method

Newton's method is a powerful technique for numerically computing the zeros of differentiable functions. It is an **iterative technique**, meaning that instead of applying a formula to directly compute the answer, it repeatedly tries to compute a better approximation until the desired accuracy is reached. Iteration is just a fancy word for repetition, which we know how to do in Python using **`for`** loops and **`while`** loops.

Newton's method gives a surprisingly simple algorithm for computing $\sqrt{k}$.

Listing 8.2: Newton's Method (Algorithm)

```
# To compute sqrt(k)
Begin with an initial guess x.
Repeat until desired accuracy is reached:
    Replace x with the average of x and k/x.
```

That's all there is to it. At the end of the loop, `x` will be approximately equal to $\sqrt{k}$.

This is an algorithm in pseudocode, but it isn't yet an executable program. Your task will be to convert this algorithm into a working Python program.

Here are some new bits of Python that may be helpful:

Scientific notation Floats may be specified in Python with an "e" before the exponent. For example, `2.914e6` represents $2.914 * 10^6 = 2914000.0$.

Absolute value Python provides the built-in function:

`abs(x)`	Absolute value of `x`.

The reason absolute value is useful is that it gives an easy measure of how close together two numbers are: the distance between x and y is $|\,x - y\,|$.

One last piece of advice that addresses a question you may have already thought of. How is it possible to know how close x is to $\sqrt{k}$ without knowing the value of $\sqrt{k}$ ahead of time? In other words, how do we know when we are close enough? The answer is to compare x^2 with k instead of x to $\sqrt{k}$. This will not tell you how close x is to the actual root, but it gives you some measurement of accuracy.

Exercises

1. Write a function `mysqrt(k)` that uses Newton's method to approximate $\sqrt{k}$ using an accuracy of 10^{-10}. Do not use the library `sqrt()` function. Write a program `newton.py` that uses your function to display the square roots of 2, 3, 4, ..., 20.

2. Compare the results of your `mysqrt(k)` function with the `sqrt()` function in the `math` module.

3. Experiment with different initial guesses in your program and report the results.

4. Experiment with different values for the required accuracy and report the results.

9.

Computer Memory: Floats

Predict what this code will do when it runs:

```
x = 0
while x != 1:
    print(x)
    x += 0.1
```

Then run it. Are you surprised? Ctrl-C, in case you have forgotten.

What is going on here? Isn't 0.1 the same as 1/10?

The explanation is subtle, but it affects every floating-point computer program in the world, including some used in life-or-death situations. The basic issue is that, like integers, floating-point numbers are stored in binary rather than decimal form.

Binary Fractions

In base ten, you are familiar with the fact that 1/3 has an infinitely-repeating decimal form, 0.33333333.... If we used base ten computers, you would probably be nervous about storing the fraction 1/3 as a float, because computers are *finite* and so wouldn't be able to store all of its digits—they would be cut off at some point.

Well, computers use base two, and 1/10 has an infinitely repeating binary form:

$$\frac{1}{10} = 0.0001100110011\ldots$$

This is because fractional values are converted to binary in the same way as integers, except that they use negative powers of 2. For example,

$$0\ .\ \underset{2^{-1}}{\boxed{1}}\ \underset{2^{-2}}{\boxed{0}}\ \underset{2^{-3}}{\boxed{1}}\ \underset{2^{-4}}{\boxed{1}} = 1 * \frac{1}{2} + 0 * \frac{1}{4} + 1 * \frac{1}{8} + 1 * \frac{1}{16} = \frac{11}{16}$$

Thus, 1/10 cannot be stored exactly as a floating-point value in a binary computer; it will be cut off at some point and therefore be a little bit off.

The exact details of how floating-point values are stored in memory are beyond the scope of this course, but you will find them in courses on computer architecture and numerical analysis.

Moral: Use Inequalities with Floats

Because so many floating-point values cannot be stored exactly in memory, testing floats using either `==` or `!=` is risky. Use inequalities (<, <=, >, >=) whenever possible.

Exercises

9.1 Convert the following decimal values to binary. Indicate which can or cannot be stored precisely as floats.

(a) 0.25
(b) 0.375
(c) 5.125
(d) 10.4375
(e) 0.3
(f) 0.5

9.2 Convert the following binary values to decimal.

(a) 0.1
(b) 0.0101
(c) 0.111
(d) 10100.01
(e) 111.1011
(f) 1000.10001

9.3 Determine whether or not the fraction 1/3 can be stored exactly as a binary floating-point value.

9.4 Fix the code at the beginning of this chapter so that it terminates as apparently intended.

9.5 Research the connection between the storage of floats and the performance of Patriot missiles in the 1991 Gulf War.

9.6 Discuss the advantages and disadvantages of using floating-point variables to store monetary values.

10.

Simulation

Consider the problem of finding the area under the curve $f(x) = e^{-x^2}$ between $x = 0$ and $x = 2$:

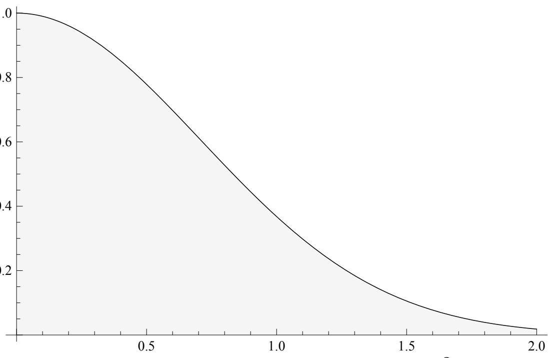

Figure 10.1: Graph of $f(x) = e^{-x^2}$.

Even if you know calculus, this is a difficult task. However, a relatively simple idea will allow us to approximate the area.

The idea begins with putting a box around the graph that contains it completely. For a function that stays positive, this means finding a maximum value m; in this case, we can use $m = 1$:

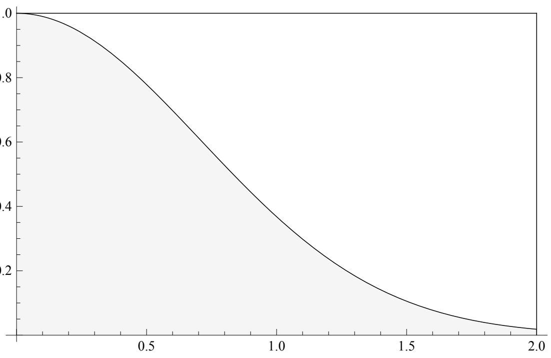

Figure 10.2: Box containing area.

Now imagine throwing random darts into the box. Some will land below the graph, in the area we want to measure, and some will miss and be above the graph. If we throw enough darts and measure the fraction of those that hit the area we want, then we can approximate the area under the curve:

Hits	88
Misses	112

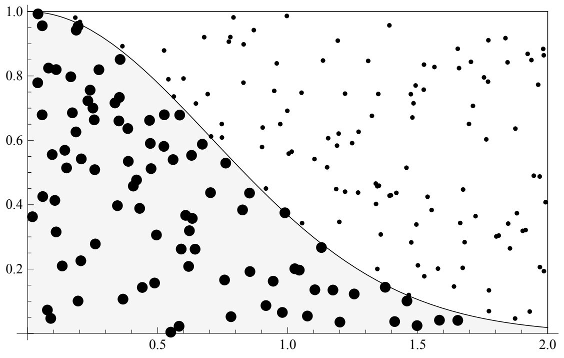

Figure 10.3: Darts thrown in box.

Since 88 of the 200 darts hit under the curve, and the total area of the box is 2, the area under the curve is approximately

$$\frac{88}{200} * 2 = 0.44 * 2 = 0.88$$

This technique is called Monte Carlo integration, and is an example of a more general class of techniques known as **Monte Carlo simulations**. Monte Carlo integration is used widely in computer graphics because the integrals that describe physically accurate lighting cannot be evaluated analytically.

Listing 10.1: Monte Carlo Integration

```python
# montecarlo.py

from random import uniform
from math import exp

def estimate_area(f, a, b, m, n=1000):
    # Estimate area under f over [a, b] for f positive and <= m.
    hits = 0
    total = m * (b - a)
    for i in range(n):
        x = uniform(a, b)
        y = uniform(0, m)
        if y <= f(x):
            hits += 1
    frac = hits / n
    return frac * total

```

```
18 def f(x):
19     return exp(-x ** 2)
20
21 def main():
22     print(estimate_area(f, 0, 2, 1))
23
24 main()
```

Most of this code should be familiar. The new features are the function calls to produce random values, the use of a default argument, and passing a function as a parameter.

Random Numbers

The main new feature we need in order to write simulations is a source of randomness. Most computer games also use some form of randomness. The `random` module in Python provides several functions that return pseudorandom values:

`random()`	Random float from $[0, 1)$.
`uniform(a, b)`	Random float from $[a, b]$.
`randint(a, b)`	Random integer from $[a, b]$.
`randrange(start, stop, step)`	Random integer from `range(start, stop, step)`.

Think of these as working in such a way that any of their possible return values are equally likely.

Default Arguments

Python allows a function to specify **default arguments** in its definition, as in line 6 of Listing 10.1. The default value is put after an equals sign that follows the parameter name, and is used if the argument is not specified at the time of the function call.

Functions as Parameters

Listing 10.1 illustrates the fact that Python allows functions as parameters to other functions. The name of the function to use is passed as an argument when the function is called. In line 22, we pass the function named `f` to `estimate_area`. That function is then used in line 13, where it is called with whatever the current value of `x` is. This is quite a powerful feature.

Exercises

10.1 Use Listing 10.1 to:

(a) Describe what the parameter `n` controls in the `estimate_area()` function.

(b) Describe the role of the variable `i` in the `estimate_area()` function. Is its value ever used?

(c) Explain why this version of `estimate_area()` does not work if the function `f` is not always positive.

(d) Explain what the **`if`** test does in line 13 in terms of the graphs and darts.

(e) Research the correct value of the area and compare it to values found by the simulation.

10.2 Modify Listing 10.1 so that the main program prints both the number of darts and the estimate for $n = 10, 100, 1000, \ldots 10^6$. Use a **`for`** loop. How quickly does the Monte Carlo algorithm seem to converge to the correct value?

10.3 Modify Listing 10.1 to estimate the area under $f(x) = x^3 + x^2$ between $x = -1$ and $x = 1$.

10.4 Given what you already know about computer programs, explain why it is difficult for a program to compute "random" numbers. Research the meaning of the term **pseudorandom** and report what you find.

10.5 Modify the centipede from Listing 6.1 to add randomness in some interesting way.

10.6 Write a `flip()` function that randomly returns either a 0 (representing heads) or a 1 (representing tails). Use this function to write a program `countflips.py` that asks the user for a number of trials and then simulates flipping a coin that many times. Print the number of heads and tails from the simulation. Does your program appear to be random?

10.7 Write a program `consecflips.py` that counts the number of coin flips necessary to reach a given number of heads in a row. Use the `flip()` function from Exercise 10.6. Ask the user for how many consecutive heads to wait for, and report the total number of flips needed at the end. Discuss the results of running your program.

10.8 Write a `roll()` function that simulates rolling a six-sided die by returning a random integer between 1 and 6 (inclusive). Use this function to write a program `countrolls.py` that asks the user for a number of trials and then simulates rolling a six-sided die that many times. Print the number of times each value from 1 to 6 appears. Does your program appear to be random?

10.9 Write a program `guess.py` that asks the user for an upper limit n, and then chooses a random number between 1 and n (inclusive). Then the program asks the user for a guess and responds with either "Too high," "Too low," or "Correct." The program should continue asking for guesses until the user is correct, and then report the number of guesses made. You do not need any functions other than `main()`.

10.10 Write a program `estpi.py` that uses a Monte Carlo simulation to estimate the value of π by throwing darts at a square centered at the origin. Imagine a circle inscribed within the square, also with center at the origin. The area of the square will be easy to calculate, and the fraction of darts within the circle will allow you to estimate the area of the circle. This will give you an estimate of π. How quickly does your program converge?

10.11 A game show contestant stands in front of three large doors. Behind one of the doors is a new car; the other two doors conceal goats. The contestant chooses a door. The host then reveals a goat behind one of the other two doors, and offers the player the chance to switch to the other remaining closed door or stay with his or her original choice. Write a simulation `gameshow.py` to help you determine the best strategy for the player. Report your results.

10.12 Develop a program `coinwar.py` to simulate playing a coin-tossing version of the card game War. In this game, one player is Same and the other is Diff. Each player starts with the same number of coins. During each round, each player flips his or her coin. If the result is the same (both heads or both tails), then player Same wins both coins; otherwise, player Diff takes both coins. Play continues until one player is out of coins.

Ask the user for the starting number of coins for each player. For each round, report the flips, who won that round, and each player's new number of coins. At the end, report the winner.

Project: Visualization

Consider whether or not the images in the previous chapter helped you to understand Monte Carlo simulation. Would lists of numbers have been as helpful? **Visualization** has become a key application of computer technology in the recent past, thanks largely to the availability of inexpensive graphics cards. In this project, we will use turtle graphics to create an animated visualization of the Monte Carlo simulation from Chapter 10, where the dots appear randomly as they are generated by the simulation.

Chapter 6 introduced both the Python `turtle` module and the online Python documentation. The exercises will lead you to develop this visualization in stages. Use the documentation to find new turtle functions as you need them.

Exercises

1. Begin the program `visual.py` by writing a `line(x0, y0, x1, y1)` function that draws a straight line from $(x0, y0)$ to $(x1, y1)$. In `main()`, set the world coordinates to have lower-left corner $(-0.5, -0.5)$ and upper-right corner $(2, 2)$. Then use the `line()` function to draw x and y axes, as well as a horizontal line at height $y = 1$.

2. Write a `plot(f, x0, x1, n)` function to draw a plot of $y = f(x)$ over the interval $[x0, x1]$ using n line segments. Most software draws function graphs in this way, as a sequence of very short straight lines. While you could use the `line()` function from the previous exercise to do this, use the following direct algorithm instead:

   ```
   Calculate dx, the width in the x direction of each segment.
   Begin at x = x0.
   Move to (x, f(x)) without drawing.
   For each segment:
       Increase x by dx.
       Draw to the next point (x, f(x))
   ```

   Use your function to plot $f(x) = e^{-x^2}$ on $[0, 2]$. Experiment with the number of segments to get a nice image.

3. Write a `placedot(f, x, y)` function that draws the correct dot at (x, y): a large blue dot if the point is below the graph of f, or a plain dot if it is above. Use your function to finish the visualization by drawing 50 random dots within the box. Hide the turtle at the end, and close the window when the user clicks in it.