



Data and Definite Loops

Introduction

Now that you know something about the basic structure of Python programs, you are ready to learn how to solve more complex problems. For the time being we will still concentrate on programs that produce output, but we will begin to explore some of the aspects of programming that require problem-solving skills.

The first half of this chapter fills in two important areas. First, it examines expressions, which are used to perform simple computations in Python, particularly those involving numeric data. Second, it discusses program elements called variables that can change in value as the program executes.

The second half of the chapter introduces your first control structure: the `for` loop. You use this structure to repeat actions in a program. This is useful whenever you find a pattern in a task such as the creation of a complex figure, because you can use a `for` loop to repeat the action to create that particular pattern. The challenge is finding each pattern and figuring out what repeated actions will reproduce it.

The `for` loop is a flexible control structure that can be used for many tasks. In this chapter we use it for definite loops, where you know exactly how many times you want to perform a particular task. In Chapter 5 we will discuss how to write indefinite loops, where you don't know in advance how many times to perform a task.

2.1 Basic Data Concepts

- Types
- Expressions
- Literals
- Arithmetic Operators
- Precedence
- Mixing and Converting Types

2.2 Variables

- A Program with Variables
- Increment/Decrement Operators
- Printing Multiple Values

2.3 The `for` Loop

- Using a Loop Variable
- Details about Ranges
- String Multiplication and Printing Partial Lines
- Nested `for` Loops

2.4 Managing Complexity

- Scope
- Pseudocode
- Constants

2.5 Case Study: Hourglass Figure

- Problem Decomposition and Pseudocode
- Initial Structured Version
- Adding a Constant

Chapter Summary

Self-Check Problems

Exercises

Programming Projects

2.1 Basic Data Concepts

Programs manipulate information, and information comes in many forms. For example, a program to keep track of library book rentals might store each book’s title, author, ISBN number, date it was checked out, name of the person to whom it was checked out, and more. Each of these pieces of data is a different kind of data: the title and author are words of text, the ISBN is an integer, the checkout time is a calendar date, and so forth. The idea of different pieces of data coming in different forms and having different allowed sets of values is related to the notion of data types, which we will explore in this section.

Types

Python programs can manipulate data, such as performing numerical computations similar to a calculator ($1 + 1$ equals 2) or searching through a body of text for a given keyword (display all dictionary words that begin with the letter “k”). Every piece of data that you manipulate in a Python program will be of a certain *type*, where a type describes a set of related values along with a set of operations you can perform on those values. One example of a type is integers, which includes values like 0, 1, 2, -4 , 65536, and so on; and operations such as addition, subtraction, and multiplication. As you write code you will often find yourself thinking about what types of data you intend to use.

Data Type
A name for a category of data values that are all related, such as type `int` in Python that represents integer values.

Some programming languages have a syntax that insists on the programmer explicitly referring to types of data throughout the code. Python has a shorter and simpler syntax where the type of each piece of data you mention in your program is generally automatically inferred from the data value itself. For example, if you write a Python program that asks to compute the result of $1 + 1$, the Python interpreter infers that you are performing a calculation on integers.

Python includes a wide variety of built-in types. We will not explore all of them in this chapter or even in this text as a whole, since some of them are not needed for basic programs. Table 2.1 lists some of Python’s built-in data types.

Let’s begin our exploration of data and types by looking at calculations with numbers. We will focus on two numeric data types in Python: `int`, which represents integers

Table 2.1 Commonly Used Data Types in Python

Type	Description	Examples
<code>int</code>	integers (whole numbers)	42, 3, 18, 20493, 0
<code>float</code>	real numbers	7.35, 14.9, 19.83423, 6.022e23
<code>str</code>	sequences of text characters (strings)	"hello", 'X', "abc 1 2 3!", ""
<code>bool</code>	logical values	True, False

(whole numbers) such as 42; and `float`, which represents real numbers with a decimal point such as 3.14. The type names `int` and `float` are Python keywords, and we will explore the usage of those keywords later in this chapter.

It may seem odd to use one type for integers and another type for real numbers. Isn't every integer a real number? The answer is yes, but these are fundamentally different types of numbers. The difference is so great that we make this distinction even in English. We don't ask, "How much sisters do you have?" or "How many do you weigh?" We realize that sisters come in discrete integer quantities (0 sisters, 1 sister, 2 sisters, 3 sisters, and so on), and we use the word "many" for integer quantities ("How many sisters do you have?"). Similarly, we realize that weight can vary by tiny amounts (175 pounds versus 175.5 pounds versus 175.25 pounds, and so on), and we use the word "much" for these real-number quantities ("How much do you weigh?").

In programming, this distinction is even more important, because integers and real numbers are represented in different ways in the computer's memory: Integers are stored exactly, while real numbers are stored as approximations with a limited number of digits of accuracy. You will see that storing values as approximations can lead to round-off errors when you use real number values.

The name `float` for real values is not very intuitive. It's an accident of history in much the same way that we still talk about "dialing" a number on our telephones even though modern telephones don't have dials. Real numbers in computing are often referred to as *floating-point numbers* because of the way that a computer's central processing unit (CPU) represents and handles them. In 1972 the C programming language introduced a data type called `float` for storing real numbers; the name caught on and was used by later languages as well. A more intuitive name might be `real`, which is what they're called in some languages, but the old C name of `float` is familiar and used by Python and many other languages. So programmers will continue to use the word `float` for real numbers, and people will still talk about "dialing" people on the phone even if they've never touched a telephone dial.

Expressions

When you write programs, you will often need to include values and calculations. The technical term for these elements is *expressions*.

Expression

A simple value or a set of operations that produces a value.

The simplest expression is a specific value, like 42 or 28.9. We call these "literal values," or *literals* for short. More complex expressions involve combining simple values. Suppose, for example, that you want to know how many bottles of water you have. If you have two 6-packs, four 4-packs, and two individual bottles, you can compute the total number of bottles with the following expression:

$$(2 * 6) + (4 * 4) + 2$$

Notice that we use an asterisk to represent multiplication and that we use parentheses to group parts of the expression. The computer determines the value of an expression by *evaluating* it.

Evaluation

The process of obtaining the value of an expression.

The value obtained when an expression is evaluated is called the *result*. Complex expressions are formed using *operators*.

Operator

A special symbol (like + or *) that is used to indicate an operation to be performed on one or more values.

The values used in the expression are called *operands*. For example, consider the following simple expressions:

3 + 29
4 * 5

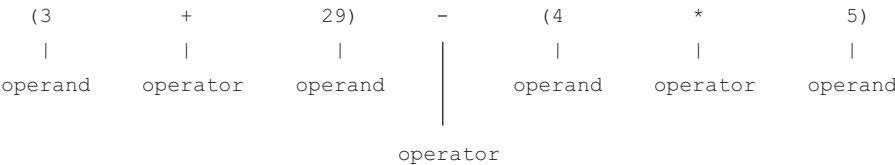
The operators here are the + and *, and the operands are simple numbers.



When you form complex expressions, these simpler expressions can in turn become operands for other operators. For example, consider the following expression:

(3 + 29) - (4 * 5)

This expression has two levels of arithmetic operators:



The addition operator has simple operands of 3 and 29 and the multiplication operator has simple operands of 4 and 5, but the subtraction operator has operands that are each parenthesized expressions with operators of their own. Thus, complex expressions can be built from smaller expressions. At the lowest level, you have simple numbers. These are used as operands to make more complex expressions, which in turn can be used as operands in even more complex expressions.

There are many things you can do with expressions. One of the simplest things you can do is to type expressions into the Python Shell, which causes the interpreter to evaluate them and display their results. This is a great way to learn more about expressions and operators:

```
>>> 1 + 1
2
>>> (3 + 29) - (4 * 5)
12
```

If you are writing a program to be saved in a file, you can print the result of an expression using a `print` statement. For example, the following three `print` statements produce the following three lines of output:

```
print(75)
print(2 + 2)
print((3 + 29) - (4 * 5))
```

```
75
4
12
```

Notice that for the second `print`, the computer evaluates the expression (adding 2 and 2) and prints the result (in this case, 4). For the third `print`, the computer evaluates all of the arithmetic operators and prints the result. You will see many different operators as you progress through this book, all of which can be used to form expressions. Expressions can be arbitrarily complex, with as many operators as you like. For that reason, when we tell you, “An expression can be used here,” we mean that you can use arbitrary expressions that include complex expressions as well as simple values.

The spacing in expressions is optional; you can write `3+4` or `3 + 4` and achieve the same result. We prefer to put spaces between an operator and its operands, and that is the style we’ll follow in this textbook. The designers of Python have written a Python official style guide “PEP” that also recommends a single space on each side of an operator.

Literals

The simplest expressions refer to values directly using what are known as *literals*. An integer literal (considered to be of type `int`) is a sequence of digits with or without a leading + or – sign:

```
3      482      -29434      0      92348      +9812
```

A real number literal (considered to be of type `float`) is any number that includes a decimal point:

```
298.4      0.284      207.      .2843      42.0      -17.452      -.98
```

Notice that `207.` and `42.0` are considered to be type `float` even though they coincide with integers, because of the decimal point. Literals of type `float` can also be expressed in scientific notation (a number followed by `e` followed by an integer):

```
2.3e4      1e-5      3.84e92      2.458e12
```

The first of these numbers represents 2.3 times 10 to the 4th power, which equals 23,000. Even though this value happens to coincide with an integer, it is considered to be of type `float` because it is expressed in scientific notation. The second number represents 1 times 10 to the negative 5th power, which is equal to 0.00001. The third number represents 3.84 times 10 to the 92nd power. The fourth number represents 2.458 times 10 to the 12th power.

We have seen that textual information can be represented as strings that contain a sequence of characters. Strings in quotation marks are considered literal values of type `str`. In later chapters we will explore strings in more detail, including how to examine and manipulate the characters of a string. As we saw previously, a string literal consists of zero or more characters enclosed in single or double quotation marks:

```
'abc'      "hello"      "I'm happy!"      'Michael "Air" Jordan'      "x"      ""
```

Finally, the type `bool` stores logical information. We won't be exploring the use of type `bool` until we reach Chapter 4 and see how to introduce logical tests into our programs, but for completeness, we include the `bool` literal values here. Logic deals with just two possibilities: `True` and `False`. These two Python keywords are the two literal values of type `bool`:

```
True      False
```

Arithmetic Operators

The basic arithmetic operators are shown in Table 2.2. The addition and subtraction operators will, of course, look familiar to you, as should the asterisk as a multiplication operator and the forward slash as a division operator.

Table 2.2 Arithmetic Operators in Python

Operator	Meaning	Example	Result
+	addition	2 + 2	4
-	subtraction	53 - 18	35
*	multiplication	3 * 8	24
/	division	9 / 2	4.5
//	integer division	9 // 2	4
%	remainder or mod	19 % 5	4
**	exponentiation	3 ** 4	81

The `**` syntax for exponentiation may seem unusual, but the behavior matches what you would expect. An expression such as `3 ** 5` means 3^5 , or 3 raised to the 5th power, which is `3 * 3 * 3 * 3 * 3`, which evaluates to 243.

Division is the most complex of the basic arithmetic operations, so it deserves further discussion. Dividing with the `/` operator produces a real number result represented as a `float` value. The result will be of type `float` even if both operands are integers. The following interaction in the Python Shell shows some examples of this:

```
>>> 11.0 / 4.0
2.75
>>> 1 / 2
0.5
>>> 12 / 2
6.0
>>> 119 / 5
23.8
```

But as we'll see throughout this textbook, there are often situations where we want to perform integer division and produce an integer (`int`) quotient. As you learned from doing long division in school, the results of integer division can be expressed as two integers, a quotient and a remainder:

119 divided by 5 = 23 (quotient) with 4 (remainder)

You can compute the integer quotient and remainder from division using the `//` and `%` operators, respectively. Here are some examples in the Python Shell that show these arithmetic operators:

```
>>> 119 // 5
23
>>> 119 % 5
4
```

These two division operators should be familiar if you recall how long-division calculations are performed. Consider the result you’d get when dividing 1079 by 34 longhand on paper:

$$\begin{array}{r} 31 \\ 34 \overline{)1079} \\ \underline{102} \\ 59 \\ \underline{34} \\ 25 \end{array}$$

Here, dividing 1079 by 34 yields 31 with a remainder of 25. Using arithmetic operators, the problem would be described like this:

```
1079 // 34 evaluates to 31
1079 % 34 evaluates to 25
```

It takes a while to get used to integer division in Python. When you are using the integer division operator (`//`), the key thing to keep in mind is that it truncates (discards) anything after the decimal point. So, if you imagine computing an answer on a calculator, just think of ignoring anything after the decimal point, as shown in the following examples in the Python Shell. Notice that the number is never rounded up; even if its fractional component is above 0.5, the integer division result is always rounded down by discarding the portion after the decimal point.

```
>>> 19 // 5      # 19 / 5 is 3.8 on a calculator
3
>>> 207 // 10    # 207 / 10 is 20.7 on a calculator
20
>>> 7 // 8       # 7 / 8 is 0.875 on a calculator
0
```

The operator `%` computes the remainder left over from integer division. It is usually referred to as the “modulus” or “mod” operator. The mod operator lets you know how much was left unaccounted for by the truncating `//` division operator. For example, given the previous examples, you’d compute the mod results as shown in Table 2.3.

In each case, you figure out how much of the number is accounted for by the truncating division operator. The mod operator gives you any excess (the remainder). When you put this into a formula, you can think of the mod operator as behaving as follows:

$$x \% y = x - (x // y) * y$$

Table 2.3 Examples of % Mod Operator

Mod problem	First divide	What does division account for?	How much is left over?	Answer
19 % 5	19 // 5 is 3	3 * 5 is 15	19 - 15 is 4	4
207 % 10	207 // 10 is 20	20 * 10 is 200	207 - 200 is 7	7
7 % 8	7 // 8 is 0	0 * 8 is 0	7 - 0 is 7	7

It is possible to get a result of 0 for the mod operator. This happens when one number divides evenly into another. For example, each of the following expressions evaluates to 0 because the second number goes evenly into the first number:

```
>>> 28 % 7
0
>>> 95 % 5
0
>>> 44 % 2
0
```

A few special cases are worth noting because they are not always immediately obvious to novice programmers:

- **Numerator smaller than denominator:** In this case division produces 0 and mod produces the original number.
- **Numerator of 0:** In this case both division and mod produce 0.
- **Denominator of 0:** In this case, both division and mod are undefined and produce a runtime error. For example, a program that attempts to evaluate any of `7 / 0`, `7 // 0`, or `7 % 0` will produce an error.

The following interactions in the Python Shell demonstrate these special cases:

```
>>> 7 // 10      # numerator smaller than denominator
0
>>> 7 % 10
7
>>> 0 // 10      # numerator of 0
0
>>> 0 % 10
0
>>> 7 // 0       # denominator of 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

The mod operator has many useful applications in computer programs. Here are just a few:

- Testing whether a number is even or odd (`number % 2` is 0 for evens, and `number % 2` is 1 for odds).
- Finding individual digits of a number (e.g., `number % 10` is the final digit).
- Finding the last four digits of a social security number (`number % 10000`).

The remainder operator can be used with `floats` as well as with integers, and it works similarly: You consider how much is left over when you take away as many “whole” values as you can. For example, the expression `10.1 % 2.4` evaluates to `0.5` because you can take away four `2.4` values from `10.1`, leaving you with `0.5` left over. But the use of `%` is much more common with integers than `float` values.

The integer division operator `//` can also be used with `floats`. It performs an exact division but then truncates the fractional component, replacing it with `.0`. For example, the expression `7.0 // 2.0` evaluates to `3.0`.

Precedence

Python expressions are like complex noun phrases in English in that they are subject to ambiguity. For example, consider the phrase, “the man on the hill by the river with the telescope.” Is the river by the hill or by the man? Is the man holding the telescope, or is the telescope on the hill, or is the telescope in the river? We don’t know how to group the various parts together.

You can get the same kind of ambiguity if parentheses aren’t used to group the parts of a Python expression. For example, the expression `2 + 3 * 4` has two operators. Which operation is performed first? You could interpret this as `(2 + 3) * 4`, which would evaluate to `5 * 4` or `20`. Or it could be `2 + (3 * 4)`, which would be `2 + 12` or `14`.

To deal with this kind of ambiguity, Python has a set of rules called *precedence* that determine the order in which various parts of an expression will be grouped together and evaluated.

Precedence

The binding power of an operator, which determines how to group and evaluate parts of an expression.

The computer applies rules of precedence when the grouping of operators in an expression is ambiguous. An operator with high precedence is evaluated first, followed by operators of lower precedence. Within a given level of precedence the operators are evaluated in one direction, usually left to right.

Table 2.4 Python Operator Precedence

Description	Operators
exponentiation	**
unary operators	+, −
multiplicative operators	*, /, //, %
additive operators	+, −

For the expressions we have seen, parentheses receive the highest level of precedence. After that, the exponentiation operator `**` has the highest precedence. The multiplicative operators `*`, `/`, `//`, `%` are next, followed by the additive operators `+` and `−`.

Table 2.4 lists these levels of precedence in descending order. (Placing a `+` or `−` sign in front of a number is considered a “unary” operator that modifies only a single operand, and those unary operators are listed in the table for completeness.) The order of precedence is the same as in the “PEMDAS” acronym many students are taught in school, short for “Parentheses, Exponents, Multiplication / Division, Addition / Subtraction.”

Thus, the expression `2 + 3 * 4` is evaluated to `14` as follows:

$$\begin{array}{ccccccc} 2 & + & 3 & * & 4 & & \\ & & & \underbrace{} & & & \\ 2 & + & & 12 & & & \\ \underbrace{} & & & & & & \\ & & & & & & 14 \end{array}$$

Within the same level of precedence, arithmetic operators are evaluated from left to right. This often doesn’t make a difference in the final result, but occasionally it does. Consider, for example, the expression `40 − 25 − 9`, which evaluates to `6` as follows:

$$\begin{array}{ccccccc} 40 & - & 25 & - & 9 & & \\ \underbrace{} & & & & & & \\ & 15 & & - & 9 & & \\ & \underbrace{} & & & & & \\ & & & & & & 6 \end{array}$$

The expression would produce a different result if the second subtraction were evaluated first.

You can always override precedence with parentheses. For example, if you really want the second subtraction to be evaluated first, you can force that to happen by

introducing parentheses and writing `40 - (25 - 9)`. The expression would then evaluate as follows:

$$\begin{array}{r} 40 - (25 - 9) \\ \qquad \underbrace{\hspace{1.5cm}} \\ 40 - 16 \\ \underbrace{\hspace{1.5cm}} \\ 24 \end{array}$$

Another concept in arithmetic is *unary* plus and minus, which take a single operand, as opposed to the binary operators we have seen thus far (e.g., `*`, `/`, and even binary `+` and `-`), all of which take two operands. For example, we can find the negation of 8 by asking for `-8`. These unary operators have a higher level of precedence than the multiplicative operators. Consequently, we can form expressions like `12 * -8`, which evaluates to `-96`.

We will see many other types of operators in the next few chapters. As we introduce more operators, we'll update the precedence listing in Table 2.4 to include them as well.

Before we leave this topic, let's look at a complex expression and see how it is evaluated step by step. Consider the following expression:

$$13 * 2 + 239 // 10 \% 5 - 2 * 2$$

It has a total of six operators: two multiplications, one integer division, one mod, one subtraction, and one addition. The multiplication, division, and mod operations will be performed first, because they have higher precedence, and they will be performed from left to right because they are all at the same level of precedence. Now we evaluate the additive operators from left to right. The final result of the long expression is 25.

$$\begin{array}{r} 13 * 2 + 239 // 10 \% 5 - 2 * 2 \\ \underbrace{\hspace{1.5cm}} \\ 26 + 239 // 10 \% 5 - 2 * 2 \\ \qquad \underbrace{\hspace{1.5cm}} \\ 26 + 23 \% 5 - 2 * 2 \\ \qquad \underbrace{\hspace{1.5cm}} \\ 26 + 3 - 2 * 2 \\ \qquad \underbrace{\hspace{1.5cm}} \\ 26 + 3 - 4 \\ \underbrace{\hspace{2.5cm}} \qquad \underbrace{\hspace{1.5cm}} \\ 29 - 4 \\ \underbrace{\hspace{3.5cm}} \\ 25 \end{array}$$

Mixing and Converting Types

You'll often find yourself mixing values of different types and wanting to convert from one type to another. Python has simple rules to avoid confusion and provides a mechanism for requesting that a value be converted from one type to another.

Two types that are frequently mixed are `ints` and `floats`. You might, for example, ask Python to compute `2 * 3.6`. This expression includes the `int` literal `2` and the `float` literal `3.6`. In this case, Python converts the `int` into a `float` and performs the computation entirely with `float` values. This is always the rule when Python encounters an `int` where it was expecting a `float`. These rules hold true even if there is an equivalent `int` that could have been used to represent the result. For example, `2 * 3.0` evaluates to the `float` value of `6.0` and not the `int` value of `6`.

The preceding is called an *implicit conversion* between types. But sometimes you want to perform an *explicit conversion* between types, such as turning a `float` into an `int`. You can ask Python for this conversion by writing the desired type name followed by the value or expression to convert in parentheses. The general syntax template for converting one type to another is:

```
type(expression)
```

Syntax template: Type conversion

For example, the expression `int(4.75)` will convert the `float` value of `4.75` into the `int` value `4`. When you convert a `float` value to an `int`, it does not round to the nearest integer; it simply truncates anything after the decimal point. There is also a `float` conversion that turns an integer value into a real number.

```
>>> int(4.75)      # convert to int
4
>>> int(17.3)
17
>>> int(3.14159)
3
>>> float(42)      # convert to float
42.0
```

Perhaps it's unclear why you would want to explicitly convert `4.75` into `4`. Why not just write `4` in the code in the first place? The more likely case where you'd use a type conversion is when you're computing the result of a more complex expression.

If you want to convert the result of an expression, you have to be careful to use the parentheses properly. For example, suppose that you have some books that are each

0.15 feet wide and you want to know how many of them will fit in a bookshelf that is 2.5 feet wide. You could do a straight division of `2.5 / 0.15`, but that evaluates to a `float` result that is between 16 and 17. Americans use the phrase “16 and change” as a way to express the idea that a value is larger than 16 but not as big as 17. Even if you use the integer division `//` operator and write `2.5 // 0.15`, the result is a `float` result of `16.0`. In this case, we don’t care about the “change”; we only want to compute the 16 part. You might form the following expression:

```
>>> int(2.5) / 0.15
13.333333333333334
```



Unfortunately, this expression evaluates to the wrong answer because the type conversion is applied to whatever is in the parentheses (here, the value `2.5`). This converts `2.5` into the integer `2`, divides by `0.15`, and evaluates to 13 and change, which isn’t an integer and isn’t the right answer. Instead, you want to form this expression:

```
>>> int(2.5 / 0.15)
16
```



This expression first performs the division to get 16 and change, and then converts that value to an `int` by truncating it. It thus evaluates to the `int` value 16, which is the answer you’re looking for.

Some languages call a conversion between types a type *cast*, as in, “casting a value in a different light.”

2.2 Variables

As you write larger programs with data and expressions, you will find that you end up with calculations that you want to use multiple times in your program. For example, you might write the following code to calculate the cost of some purchased items before and after taxes:

```
# cost of items before/after 10% tax
print("Subtotal:")
print(30 + 22 + 17 + 46)
```

Continued on next page

Continued from previous page

```
print("Taxes:")
print((30 + 22 + 17 + 46) * 0.1)
print("Total:")
print((30 + 22 + 17 + 46) * 1.10)
```

```
Subtotal:
115
Taxes:
11.5
Total:
125.5
```

Notice that the subtotal amount of $(30 + 22 + 17 + 46)$ is used three times in the code, which is redundant. We'd like to eliminate the redundancy by calculating the subtotal amount a single time and then referring to the resulting value throughout the code. Python has a feature called variables that is made for just this kind of situation. A **variable** is a location in the computer's memory that is given a name and a value. The program can store a value and retrieve it for use later.

Variable

A memory location with a name that stores a value.

Think of the computer's memory as being like a giant spreadsheet that has many cells where data can be stored. When you create a variable in Python, you are asking it to set aside one of those cells for this new variable. When you define the variable, you store an initial value in the cell. And as with a spreadsheet, you will have the option to change the value in that cell later.

When you create a variable, you have to decide on a name to use to refer to this memory location. The normal rules of Python identifiers apply (the name must start with a letter or underscore, which can be followed by any combination of letters, underscores, and digits). The standard convention in Python is for variable names to consist of lowercase letters, as in `number` or `digits`. If you want a multi-word variable name, place an underscore in front of any subsequent words, as in `number_of_students` or `average_age`.

To use a variable in a Python program, they must state its name and the value you want to store in it. The line of code that creates a variable and gives a value to it is called a **variable definition**.

Variable Definition

A request to set aside memory for a new variable with a given name and value.

A variable definition uses the following syntax:

```
name = expression
```

When the Python interpreter reaches a variable definition, it first calculates the result of the expression on the right side of the = sign. Then it stores that value into the computer's memory and associates it with the `name` on the left side of the = sign.

Once you have defined a variable, you can use that variable in your code. If you write the variable's name in an expression, it is equivalent to writing the value stored in that variable. For example, the following code has the same output as the previous code but uses a variable named `subtotal` to store the cost of the items rather than rewriting and recalculating that expression three times:

```
# cost of items before/after 10% tax (using a variable)
subtotal = 30 + 22 + 17 + 46
print("Subtotal:")
print(subtotal)
print("Taxes:")
print(subtotal * 0.1)
print("Total:")
print(subtotal * 1.10)
```

You can also try out variables in the Python Shell. If you define a variable and then type its name afterward, the shell will report its value to you:

```
>>> x = 1 + 1
>>> x
2
>>> y = x + 3
>>> y
5
>>> x * 3 + y
11
```

It is an error to refer to a variable that has not been defined. If there is no variable named `x`, the following code will result in an error:

```
# bug: try to print a variable
# that has not been defined
print("The value of x is:")
print(x)
```



```
The value of x is:
Traceback (most recent call last):
  File "undefined.py", line 7, in <module>
    main()
  File "undefined.py", line 5, in main
    print(x)
NameError: name 'x' is not defined
```

One very common variable definition statement that points out the difference between algebraic relationships and program statements is:

```
x = x + 1
```

Remember not to think of this as “ x equals $x + 1$.” There are no numbers that satisfy that mathematical equation. We read this as, “ x ’s value should be updated to become the value of x plus one.” This may seem a rather odd statement, but you should be able to decipher it given the rules outlined earlier. Suppose that the current value of x is 19. To execute the statement, you first evaluate the expression to obtain the result 20. The computer stores this value in the variable named on the left, x . Thus, this statement adds one to the value of the variable. We refer to this as *incrementing* the value of x . It is a fundamental programming operation because it is the programming equivalent of counting (1, 2, 3, 4, and so on). The following statement is a variation that counts down, which we call *decrementing* a variable:

```
x = x - 1      # decrement a variable (decrease its value)
```

We will discuss incrementing and decrementing in more detail later in the following section.

You can also define several variables in a single statement. The syntax is to write their names separated by commas, followed by an equals sign, followed by their values in the same order separated by commas:

```
name, name, ..., name = expression, expression, ..., expression
```

Syntax template: Defining multiple variables in a single statement

The following example defines three variables named `height`, `weight`, and `age`. It sets `height` to store 70, sets `weight` to store 195, and sets `age` to store 40.

```
# define several variables and set their values
height, weight, age = 70, 195, 40
```

The authors find the preceding syntax hard to read and prefer to define only a single variable on each line.

A Program with Variables

To explore more uses of variables, let's examine a larger program that computes a value called the *basal metabolic rate* (BMR), which is the number of calories a person burns in 24 hours if their body is completely at rest. Of course, a person does not remain completely at rest for 24 straight hours, and an active person burns more calories in a day than their BMR. But BMR can still be an interesting estimation of a lower bound for the minimum number of calories a person would need to eat in a day to sustain their vital organs.

Given an individual's height, weight, age, and sex, we can compute that person's BMR using the following formulas:

- men: $\text{BMR} = 10 \times (\text{weight in kg}) + 6.25 \times (\text{height in cm}) - 5 \times (\text{age in years}) + 5$
- women: $\text{BMR} = 10 \times (\text{weight in kg}) + 6.25 \times (\text{height in cm}) - 5 \times (\text{age in years}) - 161$

Or if you use American units of measurement, as we'll do in the rest of this section, the formulas are the following:

- men: $\text{BMR} = 4.54545 \times (\text{weight in lb}) + 15.875 \times (\text{height in inches}) - 5 \times (\text{age in years}) + 5$
- women: $\text{BMR} = 4.54545 \times (\text{weight in lb}) + 15.875 \times (\text{height in inches}) - 5 \times (\text{age in years}) - 161$

The height, weight, and age are important parts of the BMR formula. A program to calculate BMR, then, would naturally have three variables for these three pieces of information. There are several details that we need to discuss about variables, but it can be helpful to look at a complete program first to see the overall picture. The following program computes and prints the BMR for a 40-year-old male and female who are 5 feet 10 inches tall and weigh 195 pounds:

```

1  # This program calculates a person's Basal Metabolic Rate (BMR),
2  # which is the number of calories burned when at rest for 24 hours.
3
4  def main():
5      # define variables
6      height = 70
7      weight = 195
8      age = 40
9
10     # compute BMR for male and female
11     bmr_m = 4.54545 * weight + 15.875 * height - 5 * age + 5
12     bmr_f = 4.54545 * weight + 15.875 * height - 5 * age - 161
13

```

Continued on next page

Continued from previous page

```

14     # print results
15     print("Your Basal Metabolic Rate (BMR) is the number")
16     print("of calories that your body burns when you are")
17     print("at rest for 24 hours.")
18     print()
19     print("Current BMR (male):")
20     print(bmr_m)
21     print("Current BMR (female):")
22     print(bmr_f)
23
24     main()

```

Notice that the program includes blank lines to separate the sections and comments to indicate what the different parts of the program do. It produces the following output:

```

Your Basal Metabolic Rate (BMR) is the number
of calories that your body burns when you are
at rest for 24 hours.

Current BMR (male):
1802.61275
Current BMR (female):
1636.61275

```

Let's now examine the details of this program to understand how its variables work. For example, in the following code, the first line defines a variable named `height` and sets its value to 70. The second line defines a variable named `weight` and sets its value to 195. The third line defines a variable named `age` and sets its value to 40.

```

height = 70
weight = 195
age = 40

```

Once a variable is defined, Python sets aside a memory location to store its value. Our variable definition stores the value 70 in the memory location for the variable `height`, indicating that this person is 70 inches tall (5 feet 10 inches).

We sometimes draw pictures of variables as boxes to indicate the memory used for them, with the variable's value inside the box. After the Python interpreter executes the three statements above, the memory looks like this:

```

height  70      weight  195      age  40

```

You can think of a variable as being a named alias for a given value. By setting `height` to be `70`, you can now refer to `height` later in your program and the program will substitute the value `70` in its place.

Variable definitions can appear anywhere a statement can occur. Every variable stores a value of a particular type. Our variable named `height` stores the value `70`, which is of type `int`. We don't have to explicitly write the type of the variable in the code; the Python interpreter figures it out based on the context of the code. Some other programming languages require the programmer to explicitly specify what type of data the variable will hold, but Python does not.

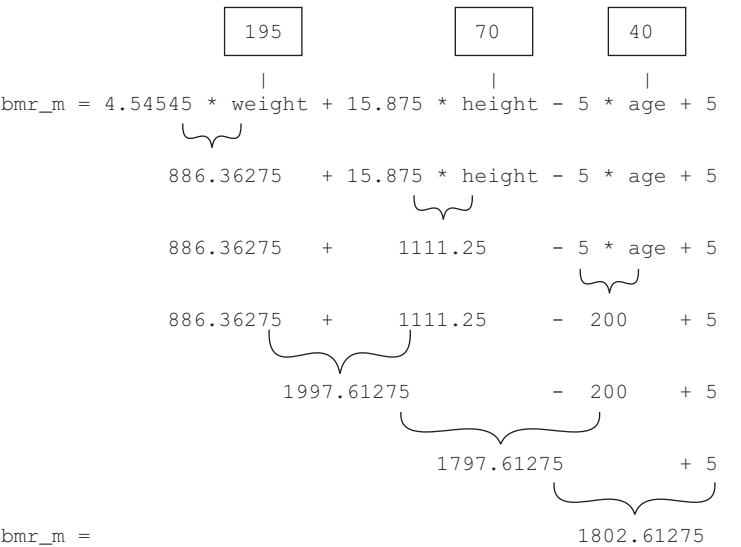
The value stored in a variable does not have to be a simple literal value. You can write a more complex expression; that expression will be evaluated, and its result will be stored into the variable. For example, the following variable definition defines the variable `height` to store the value `77`:

```
height = 44 + 3 * 11
```

When the statement executes, the computer first evaluates the expression on the right side of the `=` sign; then, it stores the result in the memory location for the given variable. The next two lines of our program contain two more definitions for variables named `bmr_m` and `bmr_f`. These definitions each use a formula (an expression to be evaluated):

```
bmr_m = 4.54545 * weight + 15.875 * height - 5 * age + 5
bmr_f = 4.54545 * weight + 15.875 * height - 5 * age - 161
```

The Python interpreter calculates the value of each expression and stores the result into each variable. When the expression refers to a variable such as `weight`, `height`, or `age`, the interpreter substitutes the value of that variable as defined previously. All of the same operator behavior and precedence still apply when using variables in an expression. The following diagram breaks down the evaluation of the value for the `bmr_m` variable:



So, after the computer has executed the `bmr_m` variable definition, the memory looks like this:

height	70	weight	195	age	40	bmr_m	1802.61275
--------	----	--------	-----	-----	----	-------	------------

The `bmr_f` variable is calculated in a similar way. The last lines of the program report the BMR result using `print` statements:

```
print("Current BMR (male):")
print(bmr_m)
print("Current BMR (female):")
print(bmr_f)
```

Notice that we can include a variable in a `print` statement the same way that we include literal values and other expressions to be printed.

As its name implies, a variable can take on different values at different times. The first time a variable is defined is called an *initialization*, while a change to the value of a variable is called an *assignment*.

Initialization

Defining a variable for the first time in your program.

Assignment

Storing a new value into an already existing variable, replacing the previous value.

For example, consider the following variation of the BMR program, which computes a new BMR for a male user, assuming the person lost 15 pounds (going from 195 pounds to 180 pounds).

```
1 # This program calculates a male user's Basal Metabolic Rate
2 # (BMR) both before and after losing 15 pounds.
3
4 def main():
5     # define variables
6     height = 70
7     weight = 195
8     age = 40
9
```

Continued on next page

Continued from previous page

```

10     # compute and print BMR for male user before weight loss
11     bmr_m = 4.54545 * weight + 15.875 * height - 5 * age + 5
12     print("Previous BMR (male):")
13     print(bmr_m)
14
15     # compute and print BMR for male user after weight loss
16     weight = 180
17     bmr_m = 4.54545 * weight + 15.875 * height - 5 * age + 5
18     print("Current BMR (male):")
19     print(bmr_m)
20
21     main()

```

The program begins the same way, setting the variables to the following values and reporting this initial value for the male BMR:

height	70	weight	195	age	40	bmr_m	1802.61275
--------	----	--------	-----	-----	----	-------	------------

But the new program then includes the following definition:

```
weight = 180
```

This changes the value of the `weight` variable:

height	70	weight	180	age	40	bmr_m	1802.61275
--------	----	--------	-----	-----	----	-------	------------

You might think that this would also change the value of the `bmr_m` variable. After all, earlier in the program we said that the following should be true:

```
bmr_m = 4.54545 * weight + 15.875 * height - 5 * age + 5
```

This is a place where the spreadsheet analogy is not as accurate. A spreadsheet can store formulas in its cells and when you update one cell it can cause the values in other cells to be updated. The same is not true in Python.

You might also be misled by the use of an equals sign for assignment. Don't confuse this statement with a statement of equality. The assignment statement does not represent an algebraic relationship. In algebra, you might say:

$$x = y + 2$$

In mathematics you state definitively that `x` is equal to `y` plus two, a fact that is true now and forever. If `x` changes, `y` will change accordingly, and vice versa. Python’s variable definition statement is very different.

The definition statement is a command to perform an action at a particular point in time. It does not represent a lasting relationship between variables. That’s why we usually say “stores” or “becomes” rather than saying “equals” when we read variable definition statements.

Getting back to the program, resetting the variable called `weight` does not reset the variable called `bmr_m`. To recompute `bmr_m` based on the new value for `weight`, we must include the second assignment statement:

```
weight = 180
bmr_m = 4.54545 * weight + 15.875 * height - 5 * age + 5
```

Otherwise, the variable `bmr_m` would store the same value as before. That would be an incorrect outcome to report to someone who’s just lost 15 pounds. By including both of these statements, we reset both the `weight` and `bmr_m` variables so that memory looks like this:



The output of the new version of the program is:

```
Previous BMR (male):
1802.61275
Current BMR (male):
1734.431
```

Increment/Decrement Operators

In addition to standard assignment with `=`, Python has several special operators that are useful for a particular family of operations that are common in programming. As we mentioned earlier, you will often find yourself increasing the value of a variable by a particular amount, an operation called *incrementing*. You will also often find yourself decreasing the value of a variable by a particular amount, an operation called *decrementing*. To accomplish this, you write statements like the following:

```
x = x + 1
y = y - 1
z = z + 2
```

Likewise, you'll frequently find yourself wanting to double or triple the value of a variable or to reduce its value by a factor of 2, in which case you might write code like the following:

```
x = x * 2
y = y * 3
z = z // 2
```

Python has a shorthand for these situations. You glue together the operator character (+, -, *, etc.) with the equals sign to get a special *assignment operator* (+=, -=, *=, etc.). This variation allows you to rewrite assignment statements like the previous ones as follows:

```
x += 1
y -= 1
z += 2

x *= 2
y *= 3
z /= 2
```

This convention is yet another detail to learn about Python, but it can make the code easier to read. Think of a statement like `x += 2` as saying, “add 2 to x.” That’s more concise than saying `x = x + 2`.

Some other languages have special operators ++ and -- for the operation of incrementing or decrementing a variable by 1. (The C++ language got its name because it is one “increment” above its predecessor language, C.) Python does not include ++ or -- operators because the language designers felt the existing += and -= operators were more clear.

Printing Multiple Values

You saw in Chapter 1 that you can output string literals using `print`. You can also output numeric expressions and variables’ values using `print`. The statement in the following code causes the interpreter first to evaluate the expression, which yields the value 14, and then to write that value to the console window.

```
print(12 + 3 - 1)
```

You have also seen that you can also include the value of a variable in a `print` statement. The following code prints the number of years until a person of a given age can retire from their job at age 65:

```
age = 40
years_until_retirement = 65 - age
```

Continued on next page

Continued from previous page

```
print("You will retire in this many years:")  
print(years_until_retirement)
```

```
You will retire in this many years:  
25
```

You'll often want to output more than one value on a line, such as a variable or expression's value with text before it. In our preceding example it might be better to print the retirement information on a single line, such as:

```
You will retire in 25 years.
```

To do so, Python allows you to supply multiple values in a `print` statement, separated by commas. All of the values or expressions you provide will be printed, with a blank space between them. The general syntax is the following:

```
print(expression, expression, ..., expression)
```

Syntax template: Printing multiple values

For example, to print the desired single line of retirement output, we can write:

```
age = 40  
years_until_retirement = 65 - age  
print("You will retire in", years_until_retirement, "years.")
```

```
You will retire in 25 years.
```

You need to pay close attention to the quotation marks and commas in a line like this to keep track of which parts are “inside” a string literal and which are outside. This line of output begins with the text “You will retire in”, then a space, then the value of the variable `years_until_retirement`, then another space, and finally the text “years.” By contrast, the following line is incorrect and does not insert the variable's value because it is inside the quotation marks of the string:

```
print("You will retire in, years_until_retirement, years.")
```



Providing multiple values in a `print` statement is related to the concept of function *parameters*, which we will discuss in detail in the next chapter. You will often use the

multiple-value `print` syntax in conjunction with variables. For example, consider the following program that computes the number of hours, minutes, and seconds in a standard year: Notice that the three `print` commands at the end each have a string literal followed by a comma and a variable name.

```

1  def main():
2      hours = 365 * 24
3      minutes = hours * 60
4      seconds = minutes * 60
5
6      print("Hours   in a year =", hours)
7      print("Minutes in a year =", minutes)
8      print("Seconds in a year =", seconds)
9
10 main()
```

```

Hours   in a year = 8760
Minutes in a year = 525600
Seconds in a year = 31536000
```

You can print arbitrarily complex expressions. For example, if you had variables named `x`, `y`, and `z`, you might want to display their values in coordinate format with parentheses and commas as shown in the following code. Notice the careful placement of quotation marks and commas. If `x`, `y`, and `z` had the values 8, 19, and 23, respectively, this statement would produce the following output:

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

```
( 8 , 19 , 23 )
```

By default the multiple values in a `print` statement are separated by single spaces. If you want to change the separator between values, you can do so using a special syntax. Just before the closing `)` parenthesis in the `print` statement, you write `sep=` followed by the string to print between the multiple values. The general syntax is the following:

```
print(expression, expression, ..., expression, sep="text")
```

Syntax template: Printing with a custom separator between values

For example, to print a date in year / month / day format, you could write the following code:

```

year = 2021
month = 1
day = 20
print("Today is", year, month, day, sep="/")

```

```
Today is 2021/1/20
```

We won't use the `sep=` syntax much in this text, but it is useful in the cases where you need precise control over your output format.

Another syntax for printing multiple values is to use the `str` function, which converts any value into a string, and then concatenate these strings together using `+` operators. The following code is equivalent to the previous example but using `str`:

```

year = 2021
month = 1
day = 20
print("Today is " + str(year) + "/" + str(month) + "/" + str(day))

```

```
Today is 2021/1/20
```

The authors prefer to avoid the use of `str` in most cases and will tend to favor the comma-separated style whenever reasonably possible. But you may see code examples online and in other texts that use `str` heavily, partly because the `str` style is more similar to the way multiple values are printed in other languages such as Java and C++.

2.3 The for Loop

Programming often involves specifying redundant tasks. The `for` loop helps to avoid such redundancy by repeatedly executing a sequence of statements over a particular range of values.

Suppose you want to write a program that repeats a printed message several times, like a chant being repeated by sports fans at your university. If your school's team is the Wildcats, you could write a program like this:

```

1 # This program prints a repeating chant.
2 def main():
3     print("Go, Cats, Go!")
4     print("Go, Cats, Go!")

```

Continued on next page

Continued from previous page

```

5     print("Go, Cats, Go!")
6     print("Go, Cats, Go!")
7     print("BEAR DOWN!")
8
9     main()

```

The program produces the following output:

```

Go, Cats, Go!
Go, Cats, Go!
Go, Cats, Go!
Go, Cats, Go!
BEAR DOWN!

```

This code approach is tedious because the program has four `print` statements that are exactly the same. You could put the `print` statement into a function and call that function four times instead, but now the function calls themselves are redundant. The more repeated copies of the chant, the worse this redundancy would get.

Python's `for` loop is made for exactly this kind of situation. A `for` loop is a statement that instructs the interpreter to execute a line or lines a given number of times. The `for` loop has the following general syntax:

```

for name in range(max):
    statements

```

Syntax template: for loop

For example, to print the same output as the previous program with a `for` loop, you could write the following code:

```

1  # This program prints a repeating chant using a for loop.
2  def main():
3      for i in range(4):
4          print("Go, Cats, Go!")
5      print("BEAR DOWN!")
6
7      main()

```

An oversimplified mental model of the `for` loop is, whatever number you write in the parentheses after `range` is the number of times the statement in the loop will repeat. If you changed the preceding program to say `range(10)`, the message would print 10 times. Notice that the last line printing “BEAR DOWN!” is not indented; it is not part of the `for` loop and will print only a single time.

You can try out `for` loops in the Python Shell. If you type the loop's header and press Enter, the shell will show a second line with a `"..."` in front of it, and you can type the loop's indented body on that line. After typing a blank line, the shell will run the loop and display its output:

```
>>> for i in range(3):
...     print("Hello, Python Shell")
...
Hello, Python Shell
Hello, Python Shell
Hello, Python Shell
```

The `for` loop is the first example of a *control structure* that we will study. A control structure is a language element that controls other statements.

Control Structure

A syntactic structure that controls other statements.

The initial line that contains the word `for` is called the *header* or *heading* of the loop, and the indented statement or statements being repeated are called the *body* of the loop. A `for` loop can repeat one statement or multiple statements. For example, the following program has a `for` loop containing two statements:

```
1 # This program prints part of a song.
2 # It demonstrates a for loop with two lines in its body.
3
4 def main():
5     print("My Coding Song:")
6     for i in range(3):
7         print("'C' is for 'Coding'")
8         print("That's good enough for me!")
9     print("Oh!")
10    print("Coding, Coding, Coding starts with 'C'!")
11    main()
```

The `print` statement that comes before the `for` loop, and the two `print` statements after the `for` loop, will not be repeated. These statements are not indented

and are not included as part of the loop's body. The program produces the following output:

```
My Coding Song:
'C' is for 'Coding'
That's good enough for me!
'C' is for 'Coding'
That's good enough for me!
'C' is for 'Coding'
That's good enough for me!
Oh!
Coding, Coding, Coding starts with 'C'!
```

Notice the order of the lines in the output. The two indented lines inside the `for` loop are repeated three times. The repeated lines appear in pairs; it prints the first line, then the second line, then the first, then the second, and so on.

Be careful to use consistent indentation to indicate controlled statements. All consecutive indented lines after the loop header are considered to be part of the body of that loop.

A program can contain multiple `for` loops. The following new version of our song program produces similar output to the original, but with two repetitions of the final line of the song:

```
1 # This program prints part of a song.
2 # It demonstrates a for loop with two lines in its body.
3
4 def main():
5     print("My Coding Song:")
6     for i in range(3):
7         print("'C' is for 'Coding'")
8         print("That's good enough for me!")
9     print("Oh!")
10    for i in range(2):
11        print("Coding, Coding, Coding starts with 'C'!")
12
13    main()
```

Common Programming Error**Forgetting to Indent**

You should use indentation to indicate the body of a `for` loop. If you intend to have a loop that contains two statements, it's easy to forget the additional lines of the body after the first one. Suppose, for example, that you want to print 20 alternating copies of the strings "Hi!" and "Ho!". You might mistakenly write the following incorrect code:

```
for i in range(20):  
    print("Hi!")  
print("Ho!")
```



The indentation indicates to the interpreter which statements are part of the loop body. Since we did not indent the second `print` statement, it is not included in the loop body and is therefore not repeated. The code would produce 20 lines of output that all say "Hi!" followed by one line of output that says "Ho!". To include both `print` statements in the loop body and therefore repeat both of them, you need to indent both lines:

```
for i in range(20):  
    print("Hi!")  
    print("Ho!")
```

**Using a Loop Variable**

A `for` loop can help you write code to process a range of numbers. We'll take advantage of that in this next program. Suppose you want to write out the squared values of the integers 0 through 5. You could write a redundant program like this:

```
1 def main():  
2     print(0, "squared =", 0 * 0)  
3     print(1, "squared =", 1 * 1)  
4     print(2, "squared =", 2 * 2)  
5     print(3, "squared =", 3 * 3)  
6     print(4, "squared =", 4 * 4)  
7     print(5, "squared =", 5 * 5)  
8  
9 main()
```

```
0 squared = 0
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

But this approach is tedious because the program has six statements that are very similar. They are all of the form:

```
print(number, "squared =", number * number)
```

In this case, `number` is either 0, 1, 2, 3, 4, or 5. What we are really trying to say here is, “execute this `print` statement for each of the values 0 through 5.” The `for` loop can help us do exactly that.

What a `for` loop actually does is to execute a piece of code once for each element in a range of integers. The expression `range(max)` includes the range of integers that begins with 0 and ends just before `max`. For example, the expression `range(4)` includes 0, 1, 2, and 3, but not 4 itself. Note that `range(max)` is a range that contains exactly `max` integers in it.

The `for` loop also contains a name that we have called `i` in our examples. The `i` here is an example of a *loop variable* (also called a *control variable*), which is a special variable that exists inside a loop that you can use in the loop’s code.

Loop Variable

The variable defined in the heading of a `for` loop, which can be used in the loop’s code and changes on each repetition of the loop.

When we say the following, the loop executes the statement once for each integer in the range, temporarily storing that integer as a variable named `i`:

```
for i in range(max):
    statement
```

- Define a variable named `i` storing the value 0, then run the statement.
- Now assign `i` to be 1, then run the statement again.
- Now assign `i` to be 2, then run the statement again.
- ...
- Now assign `i` to be `max - 2`, then run the statement again.
- Now assign `i` to be `max - 1`, then run the statement again.

The key insight about a loop variable is that you can use it in the statement inside the loop. For example, consider the following code:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

The `for` loop is running the code once for each number in the range 0 through 4, referring to that number as `i` each time. Figure 2.1 shows essentially what the `for` loop expands to become when it executes.


Each execution of the controlled statement of a loop is called an *iteration* of the loop, as in, “The loop finished executing after four iterations.” Iteration also refers to looping in general, as in, “I solved the problem using iteration.”

With all of this in mind, here is a version of our squared numbers program that uses a `for` loop to eliminate the redundant statements:

```
1 def main():
2     for i in range(6):
3         print(i, "squared =", i * i)
4
5 main()
```

All of our examples have used a loop variable named `i`. But the loop variable’s name can be any legal identifier. By convention, we often use names like `i`, `j`, and `k` for loop variables because those names are short and meaningless. But if the numbers you are printing represent something important, you can give the loop variable a more

```
for i in range(5):
    print(i)
```



```

i = 0
print(i)
i = 1
print(i)
i = 2
print(i)
i = 3
print(i)
i = 4
print(i)
```

Figure 2.1 Expansion of `for` loop using loop variable `i`

descriptive name, such as `a` or `z_coordinate` or `student_id`. Here is an example of a `for` loop with a loop variable named `num`:

```
# loop with 'num' as variable name instead of 'i'
for num in range(10):
    print(num)
```

Details about Ranges

The `for` loops we have written interacted with ranges of numbers produced from a command called `range`. The `range` command is actually a function that your code calls that creates a range of numbers for your program to process. The form we have seen so far was to supply a maximum integer in parentheses, which would produce a range from 0 (inclusive) through that maximum (exclusive). But there are situations where you don't want to start at 0, and/or when you don't want to process every integer in the range. In such cases you can supply additional information to the `range` function to customize the range of numbers produced. Table 2.5 shows the various forms of the `range` function.

For example, if you want to print the integers 1–10 inclusive, you could write the following code. Note that the max is exclusive, so we write 11 if we want 10 to be the last number included in the range.

```
# print the integers 1-10
for i in range(1, 11):
    print(i)
```

There is also a form of `range` where you supply a minimum value, a maximum value, and a step value to indicate how much the loop variable should increase on each iteration. For example, if the step is 2, the loop will process the numbers `min`, `min+2`, `min+4`, The following code prints a sequence of numbers in a song:

```
# range with a step of 2
for i in range(2, 9, 2):
    print(i)
print("Who do we appreciate?")
```

Continued on next page

Table 2.5 Ways to Create Ranges

Range Form	Description	Example	Numbers in Range
<code>range(max)</code>	range from 0 (inclusive) to max (exclusive)	<code>range(5)</code>	0, 1, 2, 3, 4
<code>range(min, max)</code>	range from min (inclusive) to max (exclusive)	<code>range(3, 7)</code>	3, 4, 5, 6
<code>range(min, max, step)</code>	range from min (inclusive) to max (exclusive), increasing by step each time	<code>range(4, 22, 3)</code>	4, 7, 10, 13, 16, 19

Continued from previous page

```
2
4
6
8
Who do we appreciate?
```

The `for` loop is running the code once for each number in the range 2 (inclusive) through 9 (exclusive, meaning that the range stops at 8), stepping upward by 2 each time. The code refers to each number as `i`. In a previous section we showed an expansion of a basic `for` loop; Figure 2.2 shows a similar expansion of this loop.

Sometimes we want to process a range of numbers in reverse order, counting down rather than up. You can achieve this by providing a negative number as the step to your range. You can accomplish this by using a decrement rather than an increment, so we sometimes refer to this as a *decrementing loop*. For example, the following code counts down from 5 to 1. Notice that the second value is written as 0 rather than 1, to make sure that 1 is included in the range.

```
# range with a negative step (count down)
for i in range(5, 0, -1):
    print(i)
print("Kaboom!")
```

```
5
4
3
2
1
Kaboom!
```

```
for i in range(2, 9, 2):
    print(i)
```

```
i = 2
print(i)
i += 2
print(i)
i += 2
print(i)
i += 2
print(i)
i += 2
print(i)
```

Figure 2.2 Expansion of for loop with a step

The values used in the range do not have to be integer literals. You can write any arbitrary integer expression:

```
# range using variables and expressions
a = 17
b = 2
c = 3
for i in range(c, a + 1, b * 2):
    print(i)
```

```
3
7
11
15
```

This loop will use a min of 3, a max of 18, and a step of 4, producing the integers 3, 7, 11, 15.

It is also possible to provide a combination of integers that iterates only once, or no times at all. The following range contains only a single integer, the number 42, so the loop prints only a single line of output:

```
# range containing a single value
for i in range(42, 43):
    print(i)
```

```
42
```

The following range does not contain any integers at all, so no output is produced. This loop performs no iterations at all. It will not cause an error; it just won't execute the `print` statement in the loop body.

```
# empty range (prints no output)
for i in range(7, 7):
    print(i)
```

The integers we write inside the parentheses when creating a range are called **parameters**. Parameters are values that are provided to a function that modify or customize its behavior. In this case, the numbers we write in the parentheses customize the range of numbers that the function creates. We will explore parameters in much more detail in the next chapter.

Common Programming Error**Off-By-One Bug (OBOB) in Range Boundary**

The fact that Python's `range` function has an inclusive min value and an exclusive max value is hard to remember for new programmers. This can lead to frequent bugs where your code loops over a range of numbers that is off by 1 from the correct range. We find that students are especially likely to make this mistake when using the form of `range` where you supply both a minimum and maximum value. For example, if you want to print the integers from 1 through 4 inclusive, you might write the code below:

```
# print integers from 1-4 inclusive (incorrect!)
for i in range(1, 4):
    print(i)
```



```
1
2
3
```

When writing these kinds of loops, remember that if you want to include a value `max` in your output, your loop must specify its maximum value as `max + 1`.

```
# print integers from 1-4 inclusive (correct)
for i in range(1, 5):
    print(i)
```

**Common Programming Error****Float as Range Boundary**

Many students who are new to `for` loops accidentally try to loop over a non-integer range. The `range` function raises an error if you write a real number as the range boundary:

```
>>> range(3.14159)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

The more subtle and common case is when the student computes the loop boundary using expressions on variables. It is easy to forget that the `/` operator on integers

Continued on next page

Continued from previous page

produces a float result, not int. The following code crashes because the variable `class_size` is a float with the value 5.4, which is not a valid range boundary:

```
1 # This buggy program contains a for loop that uses
2 # a float value as its loop range boundary.
3 def main():
4     courses = 5
5     students = 27
6     class_size = students / courses
7
8     for i in range(class_size):
9         print(i)
10
11 main()
```

Traceback (most recent call last):

```
File "float_loop.py", line 11, in <module>
    main()
File "float_loop.py", line 8, in main
    for i in range(class_size):
TypeError: 'float' object cannot be interpreted as an integer
```

The program will work properly if you convert the class size to an `int` explicitly, and/or if you use the `//` operator when dividing integers.

String Multiplication and Printing Partial Lines

We have talked about the `*` operator for performing multiplication on integers and real numbers. Interestingly, Python also allows you to use `*` on a string, which replicates the string a given number of times:

```
"text" * int
```

Syntax template: String multiplication

This operation is called *string multiplication*. For example, the expression `"hello" * 3` replicates the string `"hello"` three times, evaluating to a result of `"hellohellohello"`. The following interaction in the Python Shell shows more examples:

```
>>> "Go Cats " * 4
"Go Cats Go Cats Go Cats Go Cats "
>>> "x" * 10
```

```

"xxxxxxxxxx"
>>> "Python" * 1
"Python"
>>> "times zero!" * 0
""

```

You can print a multiplied string to see repeated text patterns in the program's output. For example:

```
print("hello" * 3)
```

```
hellohellohello
```

If you want to repeat part of a line of output, you can use the syntax shown in this chapter where multiple comma-separated values are provided in a `print` statement. For example, in the following song code, the word “la” needs to be repeated, but the initial word “Fa” does not. So we supply them separately with commas in the `print` statement on that line of output:

```
print("Deck the halls with boughs of holly")
print("Fa", "la " * 8)
```

```
Deck the halls with boughs of holly
Fa la la la la la la la la
```

You can use the combination of `for` loops and string multiplication to produce interesting patterns of characters. For example, the following code prints five lines of output, each of which contains ten `#` characters. Notice that we name our loop variable `line` here because each iteration of the loop produces a line of a figure:

```
for line in range(5):
    print("#" * 10)
```

```
#####
#####
#####
#####
#####
```

You can use the loop variable to alter the number of characters printed. In the previous code, the `for` loop always does exactly the same thing: It prints exactly 10 characters on a line of output. But if we change the code to make use of the loop's control variable `i`, the output is very different. The following code prints 1 character on line 1, 2 characters on line 2, and so on, producing a triangular figure as output.

```
for line in range(1, 6):
    print("#" * line)
```

```
#
##
###
####
#####
```

Suppose we want to print a more complex pattern, such as the following figure of output:

```
#++#
##++++##
###+++++###
####++++++####
#####+++++++#####
```

This pattern is hard to print in a single statement because each line contains a complex pattern. To help us break down the task, we will use a new variation of the `print` statement that prints a partial line of output. This new form prints the text you provide, but it does not drop the output down to the next line. The consequence is that you can place several `print` statements in sequence and all of their output will appear on the same line on the console.

The general syntax for printing a partial line of output is to place a comma followed by `end=""` inside the parentheses of your `print` statement:

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

Syntax template: Printing partial line of output

The `end=""` syntax looks a bit odd; here's what it is really doing. The `print` statement has a notion of an `end` marker that it will print after whatever text you write in parentheses. By default the end marker is `"\\n"`, which is a line break character. This means that after printing the text you provide, a `"\\n"` is printed, causing the console to move to the next line. By writing `end=""` we are changing the `end` marker into an empty string, because we don't want any line break or other characters to print after the text or value

we're printing. This causes the console to remain on the same line after printing. You can actually set the `end` marker to be any arbitrary string, but that is not a common style and we won't use it in this text.

If we use our previous code as a starting point, it contains a loop with a `range(1, 6)` and a loop variable named `i` to print exactly `i` copies of the `"#"` character on each line. For this new figure we want that same number of `"#"` characters, followed by twice that many `"+"` signs, followed by the original number of `"#"` characters a second time. To print all three of those character sequences on the same line, we will use three `print` statements inside the `for` loop as follows:

```
# printing a complex pattern using string multiplication
# and partial lines of output
for line in range(1, 6):
    print("#" * line, end="")
    print"+" * (2 * line), end="")
    print("#" * line)
```

```
#++#
##++++##
###+++++###
####++++++####
#####+++++++#####
```

Notice that the last `print` statement does not include `end=""` because we do want to end the line of output after the final sequence of `#` characters has printed. If we accidentally did include `end=""` on the final `print` statement, the output would be the following jumbled mess with no line breaks:

```
#####
```



Earlier in this chapter we saw that we can write multiple values to print, separated by commas. We also saw the `sep="text"` notation for controlling what characters should appear between each of those values. An alternative style for printing the previous figure would be to write all three sequences of characters (the initial `#` signs; the `+` signs; and the final `#` signs) of each line in a single `print` statement. We want these sequences of characters to appear with no separation between them, indicated by `sep=""`. In this form, you no longer need the `end=""`, because the single `print` statement contains the entire contents of the line of output to print.

```
# printing multiple sequences of repeated characters
# in a single print statement
for line in range(1, 6):
    print("#" * line, "+" * (2 * line), "#" * line, sep="")
```

Though this latter form is shorter and requires fewer lines of code, the authors find it difficult to read and consider it poor style. We will favor using individual `print` statements for each repeated sequence of characters, and we recommend that you do the same in your programs.

Nested for Loops

Suppose you want to print a multiplication table such as the following:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45

You can print such a table using standard `for` loops, but the code is lengthy and redundant:

```

1  # This redundant program prints a multiplication table using loops.
2  def main():
3      for x in range(1, 10):
4          print(1 * x, end="\t")
5          print()
6
7      for x in range(1, 10):
8          print(2 * x, end="\t")
9          print()
10
11     for x in range(1, 10):
12         print(3 * x, end="\t")
13         print()
14
15     for x in range(1, 10):
16         print(4 * x, end="\t")
17         print()
18
19     for x in range(1, 10):
20         print(5 * x, end="\t")
21         print()
22
23  main()
```



Earlier in this chapter we discussed string multiplication, but that feature can't help us print this particular output. String multiplication helps us when we are repeating the exact same character, but in this case the numbers are changing.

Notice the pattern in the redundant lines. There is a trio of lines containing a `for` loop header, the loop body, and a `print` statement to end the line of output. This trio of lines repeats five times in the program, with the only difference being the integer to multiply by `x` each time. This integer, which we could call `y`, takes on the values 1, 2, 3, 4, and 5 in our code.

Luckily we can remove the redundancy by embedding the trio of repeated lines inside a second loop. Such a loop is called a *nested loop*. The `for` loop controls a statement, and the `for` loop is itself a statement, which means that one `for` loop can control another `for` loop.

The following program prints the same multiplication table using nested `for` loops. We take advantage of the `end` marker in our `print` statement, setting it to `"\t"` to separate each number by a tab character so that they line up nicely on the console.

```
1 # This program prints a multiplication table using nested loops.
2 def main():
3     for y in range(1, 6):
4         for x in range(1, 10):
5             print(y * x, end="\t")
6         print()
7
8 main()
```



The behavior of this code is consistent with how standard `for` loops behave. The outer loop executes once for each value in the specified numeric range. The outer loop's range is specified as `range(1, 6)` and the outer loop variable's name is `y`, which means that the inner code should execute with `y = 1`, then with `y = 2`, and so on, up to `y = 5`. Figure 2.3 shows an expansion of what the nested loops are doing along with the console output produced by each pass of the outer loop.

The outer loop is the more long-lived of the two, the one that takes longer for each of its iterations to finish. The outer loop defines `y` to be 1 and then executes the entire inner loop. Then the outer loop assigns `y` to be 2 and then executes the entire inner loop again, and so on.

Nested loops and string multiplication both involve repetition, but nested loops are more versatile and powerful. Many programming languages don't support string multiplication, but you can achieve a similar effect using a nested loop. For example, we previously saw the following program for printing a triangular figure:

```
# print triangular figure w/ string multiplication
for i in range(1, 6):
    print("#" * i)
```

```
for y in range(1, 6):
    for x in range(1, 10):
        print(y * x, end="\t")
    print()
```

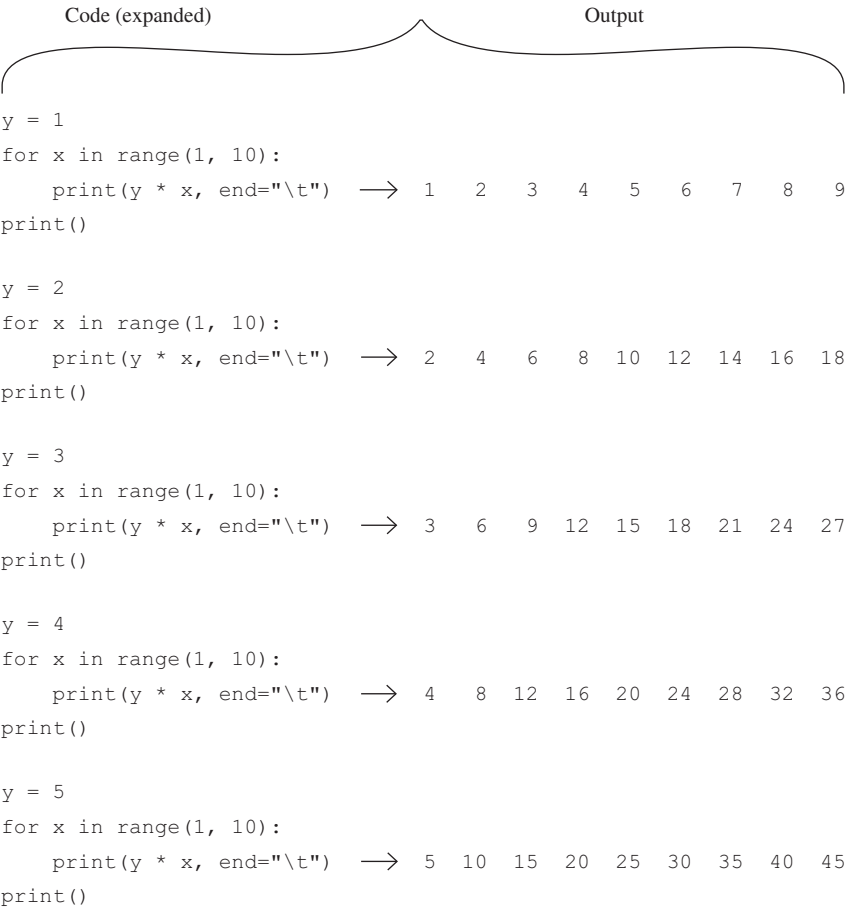


Figure 2.3 Nested loop expansion

The purpose of the string multiplication was to print a given character a given number of times. If Python did not support string multiplication, you could still print the same figure using a nested loop as follows, where each iteration of the inner loop prints a single character:

```
# print triangular figure w/out string multiplication
for i in range(1, 6):
    for j in range(i):
        print("#")
```

```
#
##
###
####
#####
```

Of course, the string multiplication version of the code is cleaner and simpler, so we will favor that approach over a nested loop when it is viable to do so.

2.4 Managing Complexity

You’ve learned about several new programming constructs in this chapter, and it’s time to put the pieces together to solve some complex tasks. As Brian Kernighan, one of the coauthors of *The C Programming Language*, once said, “Controlling complexity is the essence of computer programming.” In this section we will examine several techniques that computer scientists use to solve complex problems without being overwhelmed by complexity.

Scope

As programs get longer, it is increasingly likely that different parts of the program will interfere with each other. Python helps us to manage this potential problem by enforcing rules of *scope*.

Scope

The part of a program in which a particular definition is valid.

You’ve seen that when it comes to defining functions, you can put them in any order whatsoever. The scope of a function is the entire program file in which it appears. Variables work differently. The simple rule is that the scope of a variable definition extends from the point where it is defined to the end of the function that encloses it.

This scope rule has several implications. Consider first what it means for different functions. Each function has its own set of statements to be executed when the function is called. Any variables defined inside a function’s indented set of statements won’t be available outside that function. We refer to such variables as *local variables*.

Local Variable

A variable defined inside a function that is accessible only in that function.

Let's look at a simple example involving two functions. In this example, the `main` function defines local variables `x` and `y` and gives them initial values. Then it calls the function `compute_sum`, which tries to use the values of `x` and `y` to compute a sum. However, because the variables `x` and `y` are local to the `main` function and are not visible inside of the `compute_sum` function, this doesn't work, so the interpreter displays a runtime error called a `NameError`.

```

1  # This program produces an error because
2  # a variable is used out of scope.
3
4  def compute_sum():
5      sum = x + y          # error!
6      print("sum =", sum)
7
8  def main():
9      x = 3
10     y = 7
11     compute_sum()
12
13  main()
```

```

Traceback (most recent call last):
  File "scopel.py", line 13, in <module>
    main()
  File "scopel.py", line 11, in main
    compute_sum()
  File "scopel.py", line 5, in compute_sum
    sum = x + y
NameError: name 'x' is not defined
```

A loop variable defined in the header of a `for` loop can be accessed after the loop. Its value after the loop will be equal to the value the loop variable held on the last iteration of the loop. For example, in the following code, the loop variable `i` has the value 3 as it exits the loop:

```

# access a loop variable after the loop
for i in range(4):
    print("inside loop:", i)
print("after loop:", i)
```

```

inside loop: 0
inside loop: 1
inside loop: 2
inside loop: 3
after loop: 3
```

In general you will want to define variables inside of functions. It is possible, though, to define variables outside of functions. Such variables, called *global variables*, are visible to the entire program. At first glance, this sounds like an incredibly useful feature. You might wonder, why not just define every variable as global?

Global Variable

A variable defined outside of your functions that is accessible to the entire program. Global variables are generally considered to be poor style and are discouraged from use.

Using global variables certainly sounds simpler and more powerful. But remember that if you write large programs, you will inevitably spend a lot of time trying to find and fix bugs. A very common category of software bugs involves accidentally setting a variable to the wrong value. How do you find and fix such a bug? You need to examine all of the parts of the program that might have corrupted the variable's value. If the variable is local, to find the bug you only need to examine the function in which the variable was defined. But if the variable is global, literally any part of your program could have caused the bug. Localizing variables provides more security to your code because it minimizes the amount of your program that could modify a given variable's value.

As a nonprogramming analogy, consider the use of refrigerators in student dormitories. A dormitory building might have a large shared communal refrigerator that anybody can use. The last time we were in a dorm we noticed that most of the rooms also had individual refrigerators in them. This seems redundant, since everyone could use the shared refrigerator to store all of their food, but the reason for the duplication is obvious. Having your own private refrigerator protects your important food items from being accessed or modified (eaten) by a roommate. If you put a sandwich into the dorm's shared refrigerator and it goes missing, the culprit could be anyone, and it will be difficult to figure out who took it.

Python programs use variables to store values just as students use refrigerators to store ice cream, drinks, and other valuables. If you want to guarantee the security of something, you put it where nobody else can access it. You will use local variables in your programs in much the same way. If each individual function has its own local variables to use, you don't have to consider possible interference from other parts of the program.

(Don't worry about the loss of the power of global variables. In the next chapter we will learn about a technique called parameters that will allow us to selectively share data values from one function to another.)

Pseudocode

As you write more complex algorithms, you will find that you can't just write the entire algorithm correctly all at once. If the desired behavior or output is very complex, you will need to reason about the proper code before starting to write it. As Brian Kernighan, one of the coauthors of *The C Programming Language*, once said,

“Controlling complexity is the essence of computer programming.” Experienced programmers use several techniques to break down and solve complex problems without being overwhelmed by their complexity. In this section we will explore one such technique called *pseudocode*, which is when you write out a rough description of the program in plain text rather than in actual Python code.

Pseudocode

English-like descriptions of algorithms. Programming with pseudocode involves successively refining an informal description until it is easily translated into Python.

For example, you can describe the problem of drawing a box as the following:

Print a box with 50 lines and 30 columns of asterisks.

While this statement describes the figure, it does not give specific instructions about how to draw it (that is, what algorithm to use). Do you draw the figure line by line or column by column? In Python, figures like these must be generated line by line, because once a `print` statement has been performed on a line of output, that line cannot be changed. There is no command for going back to a previous line in the output. Therefore, you must output the first line in its entirety, then the second line in its entirety, and so on. As a result, your decompositions for figures such as these will be line-oriented at the top level. Thus, a pseudocode that is closer to Python would be:

*for each of 50 lines:
 Print a line of 30 asterisks.*

Using pseudocode, you can gradually convert an English description into something that is easily translated into a Python program. The simple examples we’ve looked at so far are hardly worth the application of pseudocode, so we will now examine the problem of generating a more complex figure:

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

This figure must also be generated line by line:

*for each of 5 lines:
 Print one line of the triangle.*

Unfortunately, each line is different. Therefore, you must come up with a general rule that fits all the lines. The first line of this figure has a series of asterisks on it with no leading spaces. Each of the subsequent lines has a series of spaces followed by a series of asterisks. Using your imagination a bit, you can say that the first line has 0 spaces on it followed by a series of asterisks. This allows you to write a general rule for making this figure:

```
for each of 5 lines:
    Print some spaces (possibly 0) on the output line.
    Print some asterisks on the output line.
End the output line.
```

In order to proceed, you must determine a rule for the number of spaces and a rule for the number of asterisks. Assuming that the lines are numbered 1 through 5, looking at the figure, you can fill in Table 2.6 with the number of each kind of character on each line.

Table 2.6 Analysis of Triangle Output Figure

Line	Spaces	Asterisks	Output
1	0	9	*****
2	1	7	*****
3	2	5	*****
4	3	3	***
5	4	1	*

You want to find a relationship between line number and the other two columns. This is simple algebra, because these columns are related in a linear way. The second column is easy to get from the line number. It equals (line - 1). The third column is a little tougher. Because it goes down by 2 every time and the first column goes up by 1 every time, you need a multiplier of -2. Then you need an appropriate constant. The number 11 seems to do the trick, so you can make the third column equal (11 - 2 * line). You can improve your pseudocode, then, as follows:

```
for each line from 1 through 5:
    Print (line - 1) spaces on the output line.
    Print (11 - 2 * line) asterisks on the output line.
End the output line.
```

This pseudocode is simple to turn into a program:

```
1 # This program draws a downward
2 # triangular figure using loops.
3 def main():
4     for line in range(1, 6):
5         print(" " * (line - 1), end="")
```

Continued on next page

Suppose you wanted to draw both of these figures in a single program. It would be tempting to try to come up with a single `for` loop that draws both figures, but there is no simple way to make a single range that contains both an increasing and decreasing sequence of numbers. Instead, you could turn each figure into a function and call both of them from `main` with a blank line between them:

```

1  # This program draws two triangular
2  # figures using loops.
3
4  # Draws a 5-line downward-facing triangle of stars.
5  def downward_triangle():
6      for line in range(1, 6):
7          print(" " * (line - 1), end="")
8          print("*" * (11 - 2 * line))
9
10 # Draws a 5-line upward-facing triangle of stars.
11 def upward_triangle():
12     for line in range(5, 0, -1):
13         print(" " * (line - 1), end="")
14         print("*" * (11 - 2 * line))
15
16 def main():
17     downward_triangle()
18     print()
19     upward_triangle()
20
21 main()

```

```

*****
*****
****
***
**
*

*
***
****
*****
*****
*****

```

Once you get practice at writing pseudocode, you'll find that it is often simple to translate well-written pseudocode into correct Python code. This is partly because of Python's clean and simple syntax. Some programmers even jokingly call Python

“executable pseudocode” since its syntax is so similar to the way a person might write down an algorithm in pseudocode format.

Constants

The triangle-drawing programs in the previous section draw a figure where each region has five lines. How would you modify the code to produce similar triangle figures but with only three lines in each part? Your first thought might be to simply change the occurrences of 5 in the code to 3. However, making that change to the most recent version of the program would produce the following incorrect output:

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



If you work through the geometry of the figure, you will discover that the problem is with the use of the number 11 in the expressions that calculate the number of asterisks to print. The number 11 actually comes from this formula:

$$2 * (\text{number of lines}) + 1$$

So when the number of lines is 5, the appropriate value is 11, but when the number of lines is three, the appropriate value is 7. Programmers call numbers like these *magic numbers*. They are magic in the sense that they seem to make the program work, but their definition is not always obvious. Glancing at the `draw_triangle` program, one is apt to ask, “Why 5? Why 11? Why 3? Why 7? Why me?”

To make programs more readable and more adaptable, you should try to avoid magic numbers whenever possible. You do so by storing the magic numbers as variables. You could use a local variable to store these values, but there are two problems with that approach. The first problem is that any variable you define in one function, such as `downward_triangle`, won’t be visible in the other function due to scope. The second problem is that programmers expect that variables might have values that change over time, but we don’t want this value to be modified after it is defined.

In cases like these, we define special variables called *constants* that are expected not to be modified after they are defined. We most often define *global constants*, which have a large scope so that they can be accessed throughout the entire program.

Constant

A variable that is defined once and never to be changed afterward. A global constant can be accessed anywhere in the program (i.e., its scope is the entire program file).

You can choose a descriptive name for a constant that explains what it represents. You can then use that name instead of referring to the specific value to make your programs more readable and adaptable. For example, in the *draw_triangle* program, you might want to introduce a constant called `LINES` that represents the number of lines. (We follow the common convention of using all uppercase letters for constant names.) You can use that constant in place of the magic number 5 and as part of an expression to calculate a value. This approach allows you to replace the magic number 11 with the formula from which it is derived ($2 * \text{LINES} + 1$).

Python does not have any special syntax for defining constants; they are just like any other variable. What differs is where you define the constant; rather than defining it indented inside a function, it is defined near the top of your program, unindented, before any of your functions. For example:

```
LINES = 5
```

You can define a constant anywhere you can define a variable, but because constants are often used by several different functions, we generally define them outside functions. We can avoid using a magic number in the *draw_triangle* program by instead referring to our constant for the number of lines. We can replace the 5 in the outer loop with this constant and replace the 11 in the second inner loop with the expression $2 * \text{LINES} + 1$. The result is the following program:

```
1 # This program draws two triangular
2 # figures using loops.
3 LINES = 3
4
5 # Draws a downward-facing triangle of stars.
6 def downward_triangle():
7     for line in range(1, LINES + 1):
8         print(" " * (line - 1), end="")
9         print("*" * (2 * LINES + 1 - 2 * line))
10
11 # Draws an upward-facing triangle of stars.
12 def upward_triangle():
13     for line in range(LINES, 0, -1):
14         print(" " * (line - 1), end="")
15         print("*" * (2 * LINES + 1 - 2 * line))
16
17 def main():
18     downward_triangle()
19     print()
20     upward_triangle()
21
22 main()
```

```

*****
 ***
  *

  *
 ***
*****

```

This new program is more adaptable and can produce figures of different sizes with only a single change. If the `LINES` constant is changed to 7, the program's output becomes the following:

```

*****
*****
*****
*****
*****
 ***
  *

  *
 ***
*****
*****
*****
*****
*****
*****
*****

```

Some programming languages allow the programmer to specify that a given variable is a constant, which then produces an error if the code tries to modify the constant's value after it has been defined. Unfortunately Python does not include this feature. By defining our constant in a global scope, Python does provide us with a bit of protection against code that tries to change the constant's value. For example, if you try to modify the value of a constant from inside a function, the interpreter raises an error:

```

# Bad code; tries to modify a global constant.
def upward_triangle():
    LINES += 3    # error!
    for line in range(LINES, 0, -1):
        ...

```



```
+-----+
Traceback (most recent call last):
  File "hourglass2_error.py", line 43, in <module>
    main()
  File "hourglass2_error.py", line 39, in main
    draw_top()
  File "hourglass2_error.py", line 14, in draw_top
    LINES += 3
UnboundLocalError: local variable 'LINES'
                    referenced before assignment
```

But unfortunately the protection is imperfect; if we instead wrote `LINES = 7`, the code would run successfully and would draw the upward_triangle with a `LINES` value of 7 rather than 4. Therefore Python constants are technically just global variables and are held constant only through convention and through the good behavior of the programmer writing the code. Modifying the value of a global constant variable is considered very poor style and strongly discouraged. The expectation followed by Python programmers is that if a variable is named in `UPPERCASE`, it is to be treated as a constant, and its value is not to be modified after it is defined. The language cannot force us to obey this convention, but we are expected to take care to do so when writing our code.

2.5 Case Study: Hourglass Figure

Now we'll consider an example that is even more complex. This program will draw an "hourglass" figure made out of patterns of repeating characters. The desired output is the following:

```
+-----+
|\12345678/|
| \123456/ |
|  \1234/  |
|   \12/   |
|    \/\   |
|   /\     |
|  /21\    |
| /4321\   |
|/654321\  |
|/87654321\|
+-----+
```

To solve it, we will follow three basic steps:

1. Decompose the task into subtasks, each of which will become a function.
2. For each subtask, make a table for the figure and compute formulas for each column of the table in terms of the line number.
3. Convert the tables into actual `for` loops and code for each function.

Problem Decomposition and Pseudocode

To generate this figure, you have to first break it down into subfigures. In doing so, you should look for lines that are similar in one way or another. The first and last lines are exactly the same. The five “top half” lines after the first line all fit one pattern, and the five “bottom half” lines after that fit another. Figure 2.4 shows the patterns of characters.

```

+-----+   line

|\12345678/|
| \123456/ |   top half
|  \1234/  |
|   \12/   |
|    \ /    |

|    /\     |
|   /21\    |
|  /4321\   |   bottom half
| /654321\  |
|/87654321\|

+-----+   line

```

Figure 2.4 Hourglass figure character patterns

Thus, you can break down the overall problem into the following pseudocode:

```

Draw a solid line.
Draw the top half of the hourglass.
Draw the bottom half of the hourglass.
Draw a solid line.

```

You should solve each subproblem independently. Eventually you’ll want to incorporate a constant to make the program more flexible, but let’s first solve the problem without worrying about the use of a constant.

The “Draw a solid line” task can be further specified as:

```

Draw a solid line:
    Print a plus on the output line.
    Print 10 dashes on the output line.
    Print a plus on the output line.
    End the line of output.

```

This set of instructions translates easily into a function:

```

# Prints a solid line of dashes.
def draw_line():

```

Continued on next page

Continued from previous page

```
print("+", end="")
print("-" * 10, end="")
print("+")
```

The top half of the hourglass is more complex. Here is a typical line:

```
| \1234/ |
```

There are four individual characters, separated by spaces and numbers.

		\	1234	/		
bar	spaces	backslash	numbers	slash	spaces	bar

Thus, a first approximation in pseudocode might look like this:

```
for each of 5 lines:
    Print a bar.
    Print some spaces.
    Print a backslash.
    Print some numbers.
    Print a slash.
    Print some spaces.
    Print a bar.
End the line of output.
```

Again, you can make a table to figure out the required expressions. Writing the individual characters will be easy enough to translate into Python, but you need to be more specific about the spaces and numbers. Each line in this group contains two sets of spaces and one set of numbers. Table 2.7 shows how many to use. We do not list the |, \, and / characters in the table because they always appear exactly once on each line.

The two sets of spaces go from 0 to 4 when the line number goes from 1 to 5; this can be expressed as (line - 1). The range of numbers on each line is more complicated.

Table 2.7 Analysis of Hourglass Figure

Line	Spaces	Numbers	Spaces	Output
1	0	1--8	0	\12345678/
2	1	1--6	1	\123456/
3	2	1--4	2	\1234/
4	3	1--2	3	\12/
5	4	none	4	\ /

As the line number goes up 1 to 2 to 3 and so on, the max value appearing in the output goes down by 2 each time from 8 to 6 and so on. One way of figuring out the pattern would be to think of this as an algebraic equation between the line number and the max number value. You could also think about what the max number value would be if there were a line number 0; by the pattern we've seen, the max would be 10. Therefore, the general formula for the range's max is $(10 - 2 * \text{line})$. But we have to write our loop using a Python range, and ranges exclude the max value you write, so we need to shift our formula by +1 to account for this. That means we actually want a `range(1, 11 - 2 * line)`.

```
for each line from 1 through 5:
    Print a bar.
    Print (line - 1) spaces.
    Print a backslash.
    Print the range of integers from 1 to (11 - 2 * line).
    Print a slash.
    Print (line - 1) spaces.
    Print a bar.
    End the line of output.
```

Initial Structured Version

The pseudocode for the top half of the hourglass is easily translated into a function called `draw_top`. A similar solution exists for the bottom half of the hourglass, which we will call `draw_bottom`. The `main` function calls the functions to draw the top and bottom halves with lines around them.

Our code uses the `end=""` modifier for many of its `print` statements so that we can print each chunk of a line's complex sequence of characters using a separate `print` statement.

We can produce most of the repeated sequences of characters using string multiplication with the `*` operator. The one exception is the sequences of integers in the middle of each line, which require us to use a nested `for` loop.

Put together, the program looks like this:

```
1 # This program draws an hourglass figure
2 # of characters and numbers using nested loops.
3
4 # Prints a solid line of dashes.
5 def draw_line():
6     print("+", end="")
7     print("-" * 10, end="")
8     print("+")
```

Continued on next page

Continued from previous page

```

9
10 # Produces the top half of the hourglass figure.
11 def draw_top():
12     for line in range(1, 6):
13         print("|", end="")
14         print(" " * (line - 1), end="")
15         print("\\", end="")
16         for i in range(1, 11 - 2 * line):
17             print(i, end="")
18             print("/", end="")
19             print(" " * (line - 1), end="")
20         print("|")
21
22 # Produces the bottom half of the hourglass figure.
23 def draw_bottom():
24     for line in range(1, 6):
25         print("|", end="")
26         print(" " * (5 - line), end="")
27         print("/", end="")
28         for i in range(2 * line - 2, 0, -1):
29             print(i, end="")
30             print("\\", end="")
31             print(" " * (5 - line), end="")
32         print("|")
33
34 def main():
35     draw_line()
36     draw_top()
37     draw_bottom()
38     draw_line()
39
40 main()

```

Adding a Constant

The hourglass program produces the desired output, but it is not very flexible. What if we wanted to produce a similar figure of a different size? The original problem involved an hourglass figure that had five lines in the top half and five lines in the bottom half. What if we wanted the following output, with three lines in the top half and three lines in the bottom half?

```
+-----+
|\1234/|
| \12/ |
|  \/  |
| /\   |
| /21\  |
|/4321\|
+-----+
```

Obviously the program would be more useful if we could make it flexible enough to produce either output. We do so by eliminating the magic numbers with the introduction of a constant. You might think that we need to introduce two constants, one for the height and one for the width, but because of the regularity of this figure, the height is determined by the width and vice versa. Consequently, we only need to introduce a single constant. Let’s use the height of the hourglass halves:

```
SUB_HEIGHT = 5
```

We’ve called the constant `SUB_HEIGHT` rather than `HEIGHT` because it refers to the height of each of the two halves, rather than the figure as a whole. Notice how we use the underscore character to separate the different words in the name of the constant.

So, how do we modify the original program to incorporate this constant? We look through the program for any magic numbers and insert the constant or an expression involving the constant where appropriate. For example, both the `draw_top` and `draw_bottom` functions have a `for` loop that executes 5 times to produce 5 lines of output. We change this to 3 to produce 3 lines of output, and more generally, we change it to `SUB_HEIGHT` to produce `SUB_HEIGHT` lines of output.

In other parts of the program we have to update our formulas for the number of dashes, spaces, and dots. Sometimes we can use educated guesses to figure out how to adjust such a formula to use the constant. If you can’t guess a proper formula, you can use the table technique to find the appropriate formula. Using this new output with a subheight of 3, you can update the various formulas in the program. We also show what the formulas would be for a subheight of 4. Table 2.8 shows the various formulas.

Table 2.8 Analysis of Different Height Figures

Sub Height	Dashes in line	Spaces in top	Numbers in top	Spaces in bottom	Numbers in bottom
3	6	line-1	range(1, 7-2*line)	3-line	range(2*line-2, 0, -1)
4	8	line-1	range(1, 9-2*line)	4-line	range(2*line-2, 0, -1)
5	10	line-1	range(1, 11-2*line)	5-line	range(2*line-2, 0, -1)

We then go through each formula (each column in the table) and figure out how to replace it with a new formula involving the constant:

- The number of dashes increases by 2 when the subheight increases by 1, so the general expression is twice the subfigure height, or $2 * \text{SUB_HEIGHT}$.
- The number of spaces in `draw_top` does not change when the subheight changes, so the expression does not need to be altered.
- The range of numbers in `draw_top` involves the number 7 for a subheight of 3, the number 9 for a subheight of 4, and the number 11 for a subheight of 5. The general expression for this is $2 * \text{SUB_HEIGHT} + 1$, and substituting this into the original range formula leads to an overall range of `range(1, 2 * SUB_HEIGHT + 1 - 2 * line)`.
- The number of spaces in `draw_bottom` involves the value 3 for a subheight of 3, the value 4 for a subheight of 4, and the value 5 for a subheight of 5. This is clearly just the subfigure height, and substituting it into the expression yields $\text{SUB_HEIGHT} - \text{line}$.
- The range of numbers in `draw_bottom` does not change when subheight changes, so the expression does not need to be altered.

Here is the new version of the program with a constant for the subheight. It uses a `SUB_HEIGHT` value of 3, but we could change this to other values to produce figures of other sizes.

```

1  # This program draws an hourglass figure
2  # of characters and numbers using nested loops.
3  # This version uses a global constant for the figure size.
4  SUB_HEIGHT = 4
5
6  # Prints a solid line of dashes.
7  def draw_line():
8      print("+", end="")
9      print("-" * (2 * SUB_HEIGHT), end="")
10     print("+")
11
12  # Produces the top half of the hourglass figure.
13  def draw_top():
14      for line in range(1, SUB_HEIGHT + 1):
15          print("|", end="")
16          print(" " * (line - 1), end="")
17          print("\\", end="")
18          for i in range(1, 2 * SUB_HEIGHT + 1 - 2 * line):
19              print(i, end="")
20          print("/", end="")

```

Continued on next page

Continued from previous page

```

21         print(" " * (line - 1), end="")
22         print("|")
23
24     # Produces the bottom half of the hourglass figure.
25     def draw_bottom():
26         for line in range(1, SUB_HEIGHT + 1):
27             print("|", end="")
28             print(" " * (SUB_HEIGHT - line), end="")
29             print("/", end="")
30             for i in range(2 * line - 2, 0, -1):
31                 print(i, end="")
32             print("\\", end="")
33             print(" " * (SUB_HEIGHT - line), end="")
34             print("|")
35
36     def main():
37         draw_line()
38         draw_top()
39         draw_bottom()
40         draw_line()
41
42     main()

```

Notice that the `SUB_HEIGHT` constant is defined at the top of the program, giving it program-wide scope, rather than locally in the individual functions. While localizing variables is a good idea, the same is not true for constants. We localize variables to avoid potential interference, but that argument doesn't hold for constants, since they are guaranteed not to change. Another argument for using local variables is that it makes functions more independent. That argument has some merit when applied to constants, but not enough. It is true that constants introduce dependencies between functions, but often that is what you want. For example, the three functions in the program should not be independent of each other when it comes to the size of the figure. Each subfigure has to use the same size constant. Imagine the potential disaster if each function had its own `SUB_HEIGHT`, each with a different value; none of the pieces would fit together.

Chapter Summary

- Python groups data into types. Some of Python's built-in types are `int` for integers, `float` for real numbers, `str` for sequences of text characters, and `bool` for logical values.
- Values and computations are called expressions. The simplest expressions are individual values, also called literals. Some example literals are: `42`, `3.14`, `"Q"`, and

`False`. Expressions may contain operators, as in `(3 + 29) - 4 * 5`. The division operation is split into exact division (`/`), integer division (`//`), and remainder (`%`) operators. You can test expressions and operators in the Python Shell.

- Rules of precedence determine the order in which multiple operators are evaluated in complex expressions.

Multiplication and division are performed before addition and subtraction. Parentheses can be used to force a particular order of evaluation.

- Variables are memory locations in which values can be stored. A variable is defined with a name and an initial value. The variable's value can be used later in the program or modified.
- Data can be printed on the console using the `print` function, just like text strings. Multiple comma-separated values can be printed to produce a more complex line of output, or you can use the `str` function to convert values into strings for printing.
- A loop is used to execute a group of statements several times. The `for` loop is one kind of loop that can

be used to apply the same statements over a range of numbers or to repeat statements a specified number of times. A loop can contain another loop, called a nested loop.

- A variable exists from the line where it is defined to the end of the function that encloses it. This range, also called the scope of the variable, constitutes the part of the program where the variable can legally be used. A program can also contain constants, which are variables defined in a global scope that are not supposed to be modified after they are defined.
- An algorithm can be easier to write if you first write an English description of it. Such a description is also called pseudocode.

Self-Check Problems

Section 2.1: Basic Data Concepts

1. Trace the evaluation of the following expressions, and give their resulting values:

- $2 + 3 * 4 - 6$
- $14 // 7 * 2 + 30 // 5 + 1$
- $(12 + 3) // 4 * 2$
- $(238 \% 10 + 3) \% 7$
- $(18 - 7) * (43 \% 10)$
- $2 + 19 \% 5 - (11 * (5 // 2))$
- $813 \% 100 // 3 + 2.4$
- $26 \% 10 \% 4 * 3$
- $22 + 4 ** 1 * 2$
- $23 \% 8 \% 3$
- $12 - 2 ** 2 - 3 ** 2$
- $6/2 + 7//3$
- $6 * 7 \% 4$
- $3 * 4 + 2 * 3$
- $177 \% 100 \% 10 // 2$
- $89 \% (5 + 5) \% 5$
- $392 // 10 \% 10 // 2$
- $8 * 2 - 7 // 4$
- $37 \% 20 \% 3 * 4$
- $17 \% 10 // 4$

2. Trace the evaluation of the following expressions, and give their resulting values:

- $4.0 / 2 * 9 / 2$
- $2.5 * 2 + 8 / 5.0 + 10 // 3$
- $12 // 7 * 4.4 * 2 // 4$

- d. $4 * 3 // 8 + 2.5 * 2$
- e. $(5 * 7.0 / 2 - 2.5) / 5 * 2$
- f. $41 \% 7 * 3 // 5 + 5 // 2 * 2.5$
- g. $10.0 / 2 / 4$
- h. $8 // 5 + 13 // 2 / 3.0$
- i. $(2.5 + 3.5) / 2$
- j. $9 // 4 * 2.0 - 5 // 4$
- k. $9 / 2.0 + 7 // 3 - 3.0 / 2$
- l. $813 \% 100 // 3 + 2.4$
- m. $27 // 2 / 2.0 * (4.3 + 1.7) - 8 // 3$
- n. $53 // 5 / (0.6 + 1.4) / 2 ** 1 + 13 // 2$
- o. $2.5 * 2 + 8 / 5.0 + 10 // 3$
- p. $2 * 3 // 4 * 2 / 4.0 + 4.5 - 1$
- q. $89 \% 10 // 4 * 2.0 / 5 + (1.5 + 1.0 / 2) * 2$
- r. $1 ** 5 + 7 ** 2 / 2.0$

Section 2.2: Variables

3. Which of the following choices is the correct syntax for defining a real number variable named `grade` and initializing its value to 4.0?

- a. `int grade = 4.0`
- b. `grade = float 4.0`
- c. `float grade = 4.0`
- d. `grade = 4`
- e. `grade = 4.0`

4. Suppose you have a variable called `number` that stores an integer. What Python expression produces the last digit of the number (the 1s place)?

5. The following program contains 4 mistakes! What are they?

```
def main():
    x = 2
    print("x is" x)
    x = 15.2    # set x to 15.2
    print("x is now , x")
    y = 0       # set y to 1 more than x
    y = int x + 1
    print("x and y are " , x , and , y)
main()
```

6. Suppose you have a variable called `number` that stores an integer. What Python expression produces the second-to-last digit of the number (the 10s place)? What expression produces the third-to-last digit of the number (the 100s place)?

7. What are the values of `a`, `b`, and `c` after the following statements?

```
a = 5
b = 10
c = b

a = a + 1
b = b - 1
c = c + a
```


8. What are the values of `first` and `second` at the end of the following code? How would you describe the net effect of the code statements in this exercise? Consider the following code:

```
first = 8
second = 19
first = first + second
second = first - second
first = first - second
```

9. Rewrite the code from the previous exercise to be shorter, by defining the variables together and by using the special assignment operators (e.g., `+=`, `-=`, `*=`, and `/=`) as appropriate.

10. What are the values of `i`, `j`, and `k` after the following statements?

```
i = 2
j = 3
k = 4
x = i + j + k
```

```
i = x - i - j
j = x - j - k
k = x - i - k
```

11. What is the output from the following code?

```
max = 0
min = 10
max = 17 - 4 // 10
max = max + 6
min = max - min
print(max * 2)
print(max + min)
print(max)
print(min)
```

12. The following program redundantly repeats the same expressions many times. Modify the program to remove all redundant expressions using variables.

```
def main():
    # Calculate pay at work based on hours worked each day
    print("My total hours worked:")
    print(4 + 5 + 8 + 4)

    print("My hourly salary:")
    print("8.75")

    print("My total pay:")
    print((4 + 5 + 8 + 4) * 8.75)

    print("My taxes owed:") # 20% tax
    print((4 + 5 + 8 + 4) * 8.75 * 0.20)

main()
```

Section 2.3: The for Loop

13. Complete the following code, replacing the “FINISH ME” parts with your own code:

```
def main():
    for i in range("FINISH ME"):
        print("FINISH ME")
main()
```

to produce the following output:

```
2 times 1 = 2
2 times 2 = 4
2 times 3 = 6
2 times 4 = 8
```

14. Assume that you have a variable called `count` that will take on the values 1, 2, 3, 4, and so on. You are going to formulate expressions in terms of `count` that will yield different sequences. For example, to get the sequence 2, 4, 6, 8, 10, 12, ... , you would use the expression `(2 * count)`. Fill in the following table, indicating an expression that will generate each sequence.

Sequence	Expression
2, 4, 6, 8, 10, 12, ...	
4, 19, 34, 49, 64, 79, ...	
30, 20, 10, 0, 210, 220, ...	
−7, −3, 1, 5, 9, 13, ...	
97, 94, 91, 88, 85, 82, ...	

15. Complete the code for the following for loop:

```
for i in range(1, 7):
    # your code here

−4
14
32
50
68
86
```

The loop should print the following numbers, one per line:

16. What is the output of the following `odd_stuff` function?

```
def odd_stuff():
    number = 4
    for count in range(1, number + 1):
        print(number)
        number = number // 2
```

17. What is the output of the following loop?

```
total = 10
for number in range(1, total // 2):
    total = total - number
    print(total, number)
```

18. What is the output of the following loop?

```
print("+---+")
for i in range(1, 4):
    print("\    /")
    print("/    \")
print("+---+")
```

19. What is the output of the following loop?

```
print("T-minus ", end="")
for i in range(5, 0, -1):
    print(i, end=", ")
print("Blastoff!")
```

20. What is the output of the following sequence of loops?

```
for i in range(1, 6):
    for j in range(1, 11):
        print(i * j, end=" ")
    print()
```

21. What is the output of the following sequence of loops?

```
for i in range(1, 3):
    for j in range(1, 4):
        for k in range(1, 5):
            print("*", end="")
        print("!", end="")
    print()
```

Section 2.4: Managing Complexity

22. Suppose that you have a variable called `line` that will take on the values 1, 2, 3, 4, and so on, and a constant named `SIZE` that takes one of two values. You are going to formulate expressions in terms of `line` and `SIZE` that will yield different sequences of numbers of characters. Fill in the table below, indicating an expression that will generate each sequence.

<code>line</code> Value	Constant <code>SIZE</code> Value	Number of Characters	Expression
a. 1, 2, 3, 4, 5, 6, ...	1	4, 6, 8, 10, 12, 14, ...	
1, 2, 3, 4, 5, 6, ...	2	6, 8, 10, 12, 14, 16, ...	
b. 1, 2, 3, 4, 5, 6, ...	3	13, 17, 21, 25, 29, 33, ...	
1, 2, 3, 4, 5, 6, ...	5	19, 23, 27, 31, 35, 39, ...	
c. 1, 2, 3, 4, 5, 6, ...	4	10, 9, 8, 7, 6, 5, ...	
1, 2, 3, 4, 5, 6, ...	9	20, 19, 18, 17, 16, 15, ...	

23. Write a table that determines the expressions for the number of each type of character on each of the 6 lines in the following output.

```
!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!//
\\\\!!!!!!!!!!!!!!!!//
\\\\\\\\!!!!!!!!!!!!//
\\\\\\\\\\\\!!!!!!!!//
\\\\\\\\\\\\\\\\!!!!//
\\\\\\\\\\\\\\\\\\!//
```

24. Suppose that a program has been written that produces the output shown in the previous problem. Now the author wants the program to be scalable using a constant called `SIZE`. The previous output used a constant height of 6, since there were 6 lines. The following is the output for a constant height of 4. Create a new table that shows the expressions for the character counts at this new size of 4, and compare these tables to figure out the expressions for any size using the `SIZE` constant.

```
!!!!!!!!!!!!
\\!!!!!!!!!!//
\\\\\\\\!!!!!!//
\\\\\\\\\\\\!!!!//
```

Exercises

1. Write a `for` loop that produces the following output:

```
1 4 9 16 25 36 49 64 81 100
```

For added challenge, try to modify your code so that it does not need to use the `*` multiplication operator. (It can be done! Hint: Look at the differences between adjacent numbers.)

2. The Fibonacci numbers are a sequence of integers in which the first two elements are 1, and each following element is the sum of the two preceding elements. The mathematical definition of each *k*th Fibonacci number is the following:

$$F(k) = \begin{cases} F(k - 1) + F(k - 2), & k > 2 \\ 1, & k \leq 2 \end{cases}$$

The first 12 Fibonacci numbers are

```
1 1 2 3 5 8 13 21 34 55 89 144
```

Write a `for` loop that computes and prints the first 12 Fibonacci numbers.

3. Write `for` loops to produce the following output:

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

4. Write `for` loops to produce the following output:

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

5. Write `for` loops to produce the following output:

```
1
22
333
4444
55555
666666
7777777
```

6. It's common to print a rotating, increasing list of single-digit numbers at the start of a program's output as a visual guide to number the columns of the output to follow. With this in mind, write nested `for` loops to produce the following output, with each line 60 characters wide:

```
| 1234567890123456789012345678901234567890123456789012345678901234567890
|
```

7. Modify your code from the previous exercise so that it could easily be modified to display a different range of numbers (instead of 1234567890) and a different number of repetitions of those numbers (instead of 60 total characters), with the vertical bars still matching up correctly. Use constants instead of “magic numbers.” Here are some example outputs that could be generated by changing your constants:

```
| 1234012340123401234012340123401234012340123401234012340
|
| 1234567012345670123456701234567012345670123456701234567012345670
|
```

8. Write a function called `print_design` that produces the following output. Use `for` loops to capture the structure of the figure.

```
-----1-----
----333----
---5555---
--777777--
-99999999-
```

9. Write a Python program that produces the following output. Use `for` loops and string multiplication to capture the structure of the figure.

```
!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!//
\\\\\\!!!!!!!!!!!!!!///
\\\\\\\\!!!!!!!!!!!!///
\\\\\\\\\\!!!!!!!!!!!!///
\\\\\\\\\\\\!!!!!!!!!!!!///
\\\\\\\\\\\\\\!!!!!!!!!!!!///
```

10. Modify your program from the previous exercise to use a constant for the figure’s height. (You may want to make loop tables first.) The previous output used a constant height of 6. The following are the outputs for constant heights of 4 and 8:

Height 4	Height 8
!!!!!!!!!!!!	!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!//	\\!!!!!!!!!!!!!!!!!!//
\\\\\\!!!!!!///	\\\\\\!!!!!!!!!!!!!!///
\\\\\\\\\\!!!!///	\\\\\\\\\\!!!!!!!!!!!!///
	\\\\\\\\\\\\\\!!!!///
	\\\\\\\\\\\\\\\\\\!!!!///
	\\\\\\\\\\\\\\\\\\\\\\!!!!///
	\\\\\\\\\\\\\\\\\\\\\\!!!!///
	\\\\\\\\\\\\\\\\\\\\\\!!!!///
	\\\\\\\\\\\\\\\\\\\\\\!!!!///

11. Write a Python program that produces the following output. Use `for` loops and string multiplication to capture the structure of the figure. Once you get it to work, add a constant so that the size of the figure can be changed simply by changing the constant's value.

```

+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+

```

12. Write a Python program that produces the following output. Use `for` loops to capture the structure of the figure.

```

////////////////\
////////////////\*****\
////////////////\*****\
////////////////\*****\
////////////////\
*****

```

13. Modify the program from the previous exercise to use a constant for the figure's height. (You may want to make loop tables first.) The previous output used a constant height of 5. The following are the outputs for constant heights of 3 and 6:

Height 3	Height 6
////////////////\	////////////////\
////*****\	////////////////*****\
*****	////////////////*****\
	/////////*****\
	////*****\

Programming Projects

1. Write a program that produces the following output using nested `for` loops and string multiplication:

```
***** /////////////// *****
***** //////////////\  *****
****  //////////////\   ****
***   //////////////\   ***
**    //////////////\   **
*     //////////////\   *
      //////////////\
      \\\\\\\\\\\\\\\
```

2. Write a program that produces the following output using nested `for` loops:

```
+-----+
|      *      |
|   /*\      |
|  /**\      |
| //*\      |
| ///*\      |
| \\\*///    |
|  \*\      |
|   \*/      |
|      *      |
+-----+
|  \\\*///    |
|   \*\      |
|    \*/      |
|      *      |
|      *      |
|   /*\      |
|  /**\      |
| //*\      |
| ///*\      |
+-----+
```


3. Write a program that produces the following hourglass figure as its output using nested `for` loops and string multiplication:

```
| " " " " " " " " |
 \ : : : : : : /
  \ : : : : : /
   \ : : : : /
    \ : : /
     | |
    / : : \
   / : : : \
  / : : : : \
 / : : : : : \
/ : : : : : : \
| _____ |
```

4. Write a program that produces the following rocket ship figure as its output using nested `for` loops and string multiplication. Use a constant to make it possible to change the size of the rocket (the following output uses a size of 3).

```
  / ** \
 / / ** \ \
 / / / ** \ \ \
 / / / / ** \ \ \ \
 / / / / / ** \ \ \ \ \
+ = * = * = * = * = * +
| . . / \ . . . / \ . . |
| . / \ / \ . . / \ / \ . |
| / \ / \ / \ / \ / \ |
| \ / \ / \ / \ / \ / |
| . \ \ / . . \ \ / . |
| . . \ / . . . \ / . . |
+ = * = * = * = * = * +
| \ \ / \ / \ / \ / \ |
| . \ \ / . . \ \ / . |
| . . \ / . . . \ / . . |
| . . / \ . . . / \ . . |
| . / \ / \ . . / \ / \ . |
| / \ / \ / \ / \ / \ |
+ = * = * = * = * = * +
  / ** \
 / / ** \ \
 / / / ** \ \ \
 / / / / ** \ \ \ \
 / / / / / ** \ \ \ \ \
```

5. Write a program that produces the following figure (which vaguely resembles the Seattle Space Needle) as its output using nested `for` loops and string multiplication. Use a constant to make it possible to change the size of the figure (the following output uses a size of 4).

The diagram illustrates the iterative construction of a fractal curve, specifically a Koch curve. It shows the replacement of a line segment with four segments, each scaled by 1/3, arranged in a specific pattern. The process starts with a single vertical segment and branches out into a more complex, self-similar structure. The diagram includes a legend for the replacement rule: a segment is replaced by four segments, each scaled by 1/3, arranged in a specific pattern. The final stage shows a highly detailed, fractal-like curve.

-

-