Chapter 7

COMPONENTS OF PROGRAMMING

The idea of "writing a computer program" can sound like a difficult and complicated job, full of strange terminology and complex rules. There are many elements, though, that are universal to nearly all programming languages. Some of these common building blocks are the subject of this chapter. By understanding these components in general terms, you will be better able to think of the structure of your program independent from the syntax and grammar of particular languages.

What is a program?

If you followed along with the previous chapter, you have already written a program in the form of a shell script. There are many programming languages, each optimized for different trade-offs between ease of use, computational efficiency, portability, and specific tasks. The suite of popular languages has evolved over time, so that most of the widely used languages today were not even in existence twenty years ago. However, many key programming concepts and basic building blocks have remained largely the same.

Goals of the next few chapters

At first, writing a program may seem like taking a spelling test, a foreign language exam, a math test, and a car repair class—all at the same time. Learning the syntax of a programming language is the most conspicuous impediment to becoming a programmer, but it is actually more important to learn to think about your data analysis problems as a series of programmable tasks.

As an analogy, imagine you want to instruct someone in how to bake a pie. You might write instructions in English with U.S. units of measure—but you would be describing a set of underlying *tasks* that are themselves independent of language and measurement conventions. That is, the tasks required to bake the pie could be articulated in any language, regardless of syntax, vocabulary, or grammar. You

can think of the physical acts that go into cooking as similar to the building blocks of programming, and think of the words used to describe these acts as the terms of a programming language.

This book is not a cookbook—we do not provide you with lists of instructions for different tasks. This book is an introduction to how to make your own recipes. It might seem like a big step to go from not being able to cook at all to cooking from scratch without a recipe, but you can ease into it by starting simply, with only a few ingredients. You will grow familiar with programming by reusing the new skills you acquire and recycling and adapting bits of programs you write.

COMPILED VERSUS INTERPRETED PROGRAMS Programs are rarely written in a language that microprocessors can understand directly. Programs written in some languages, such as C and C++, are translated from human-writable and human-readable code (called **source code**) to computer-understandable instructions. Such translated programs are said to be **compiled**. Once the program has been compiled, it can be run again and again without being retranslated. The majority of programs that you use regularly are compiled in this way. The compiled program file is specific to a particular family of microprocessors and an operating system, and is not **portable** to another operating system. Also, other programmers can't modify the program or recompile it for a different type of computer unless they have the source code.

However, another category of programming languages exists in which programs typically are not typically compiled, but instead processed by an **interpreter** each time they are run. The interpreter is itself a compiled program that runs directly on the microprocessor. These languages are called interpreted languages, or **scripting languages**, and include Python, R, MATLAB, and Perl, among many others. There can be additional computational overhead to reinterpreting a program each time it is run, but even so interpreted programs have several advantages, especially for scientists. One such advantage is portability: the program file itself is the source code, so it is easier to modify, and can usually be run on any computer that has the correct interpreter. The shell scripts you wrote in previous chapters were interpreted programs, with the interpreter being `bash`. In the following chapters you will write programs for the `python` interpreter.

The distinctions between compiled and interpreted programming languages are discussed in greater detail in Chapter 21, along with instructions for compiling simple programs.

Practical programming

The scope of the next few chapters is modest compared to books dedicated to learning programming; we hope to convey a minimal set of skills that will get you started writing your own programs. These chapters are not intended as a comprehensive introduction to programming, or as a grounding in programming theory, but as a taste of practical programming. This will give you a footing for solving some of the simple problems you already face, as well as provide a departure point for getting started on more complex projects. The next step to taking on

these projects would likely include more in-depth instruction, via either online resources or any of the many excellent books available for beginners. At the end of the next few chapters you should have a much better sense of which tasks are best approached by writing new software, what's entailed in writing a new program or repurposing an old one, and what specifically you would need to learn to take on more complex challenges.

The present chapter focuses on the basic building blocks of programs and introduces some commonly used terms (Table 7.1). Most of this discussion is not specific to any particular computer language, but we do lean somewhat towards Python, the language we will focus on in the following chapters. If you plan to learn programming with a language other than Python, it is still important to become familiar with the concepts described here. In Appendix 5, you can see examples of how these elements are applied in a variety of programming languages. Although this treatment is neither comprehensive nor general to all computer languages, it will form a foundation for translating data-analysis tasks ("I need to reformat each line of this file into tab-delimited decimal values") into programming terms ("Open file, read in each line with a `while` loop, parse variables, save formatted output"). After this overview, the subsequent chapters will explain how to use these building blocks specifically in the context of Python.

TABLE 7.1 Glossary of program-related terms

Term	Definfition
Arguments	Values that are sent to a program at the time it is run
Code	*Noun*: A program or line of a program, sometimes called source code *Verb*: The act of writing a program
Execute	To begin and carry out the operation of a program; synonymous with *run*
Function	A subprogram that can be called repeatedly to perform the same task within a program
Parameters	Values that are sent to a function when it is called
Parse	To extract particular data elements from a larger block of text
Return	In a function, the act of sending back a value; the value can be assigned to a variable by referring to the function name (e.g., in `y = cos(x)`, the function `cos` calculates and returns the value of the cosine of `x`, which is assigned to `y`)
Run	To execute the sequence of commands in a program; can also refer to the processing of a file by a program that finds instructions within it
Statement	A line of a program or script, which can assign a value, do a comparison, or perform other operations

Variables

The anatomy of a variable

Anyone who has taken algebra knows about variables. In essence, each variable is a name that holds a value. The value can be a number, a series of characters, or something even more complex. As the name suggests, variables can vary, by changing what pieces of information they hold or point to.

Each variable has several attributes. First, there is its **name**, usually a plain word that gives some indication of the variable content, such as `Sequence`, `Plot_Num`, or `Mass`. The programmer designates the name of the variable at the time the program is written. Some languages require a special symbol or symbols to be associated with the name of the variable; in Perl, for instance, variable names begin with `$`. Other languages, such as Python, forbid punctuation within variable names. Almost all languages make it illegal for the first character of a variable name to be a digit. Sometimes single letters alone are used as names, especially when the variable is keeping track of some internal value just for controlling program flow. In general, though, it is best to pick variable names that are distinctive and memorable, even if it means a bit more typing. This is one of the simplest steps you can take to improve the readability of your programs so that they can be understood later, whether by you or by someone else. To improve clarity, most variable names in this book will begin with a capital letter, so that they can be easily distinguished from other program components. (The exception is variable names only a single letter long; we have left these lowercase, since they can't be confused with anything else.)

The second variable attribute addressed here may initially seem a bit more abstract. This is its **type**, a designation of the kind of information the variable contains. Variables in modern computer languages each have a type, accommodating numbers, strings made up of text, images, lists of data, and so on. It is also possible to design and use entirely new types.

A third attribute for a variable is its **value**, that is, the actual piece of information that it holds. Not only can a variable be assigned a value upon creation, but it can be reassigned different values throughout the course of a program. A close relationship exists between a variable's type and its value, such that a variable of a certain type can only contain certain kinds of values.

Variables have other attributes that we won't discuss in detail here, but they can become important as programs get more complex. One of these is **scope**, which designates where in a program a variable can be accessed.

Basic variable types

There are a few variable types that you will use again and again; examples of these are shown in Table 7.2. You will notice that there are several variable types for numbers. This is because there are trade-offs between the size and precision of a number and the computational resources needed to store and manipulate the number. Having several number types may seem unnecessarily confusing, but it allows programmers to write code that is more computationally efficient and

uses less memory; it also helps ensure that the program is behaving as expected.

TABLE 7.2 Commonly used variable types

Type	Example
Integer	`98`
Float	`98.6`
Boolean	`False`
String	`'Bargmannia elongata'`

Integer One of the most basic variable types is **integer**, which holds whole numbers without any fractional component (e.g., `0`, `-1`, and `255`). The limits on integer size are language-dependent, but in most languages integers can range from `-2,147,483,648` to `2,147,483,647`.[1] This range of possible values for an integer may seem so large that there is no chance that you could ever exceed it—and yet a timer that counts elapsed time in milliseconds would do so within a month. If you find yourself in a situation where the regular integer type is insufficient, you can go with one of the more specialized integer types: a variable of type **long** can hold much larger integers, but will take more memory, while a variable of type **unsigned** can go twice as high as a normal integer, but cannot hold negative integers.

Floating point Many scientific applications require numbers that either are extremely large, or else have decimal fractions. A useful type for these situations is **float**, shorthand for "floating point." This name signifies that the decimal point can float to any position, and not just lurk at the very end of the number. Like scientific notation, a float can represent a huge number or a tiny number using just a few digits (e.g., 4×10^{22}, the number of bacteria in a bioluminescent milky sea, or 1×10^{-15}, the diameter of a proton in meters). Floats are sometimes written as `4e22` or `4E22`. The limitations of floats mainly have to do with the number of significant digits they can accurately retain, rather than the size of the number. If a standard float isn't precise enough for your program, you can use a **double**, which occupies twice as much memory (hence the name) and has higher precision.

Boolean A variable of type **Boolean**[2] can have one of two values, `True` or `False`. Booleans are used for logical operations, such as the response to the question, "Is the value of variable `x` greater than variable `y`?" Booleans can also be interpreted as numbers. In this context, `True` is equivalent to `1` and `False` is equivalent to `0`.

Strings Most languages have a data type called **string** to handle a sequence of text characters. A string can include letters, digits, punctuation, white space (spaces, line endings, etc.), and other characters. Within a program, string values are almost always bounded by a pair of straight quotation marks, either single (') or double ("). This is to differentiate text that is placed in a string from variable names and commands. Here are some examples of assigning strings to variables:

---

[1] Such limits are determined by the amount of memory used to store a variable; in the case of an integer with these limits, this amount is 32 bits. See Appendix 6 for more specifics on how numbers are stored and how this affects the way computers can work with them.

[2] The name "Boolean" is derived from the 19th century mathematician George Boole.

```
SequenceName = "Bolinopsis infundibulum"
Primer1 = 'ATGTCTCATTCAAAGCAGG'
DateString = "18-Dec-1865\t13:05"
Location = "Pt. Panic, Oahu, Hawai'i"
```

In each case, the text within the quotation marks on the right is placed in the variable named on the left. Some programming languages allow strings to hold nearly any character, while others cannot include certain characters. Unfortunately for international users, support for extended character sets (for example, ø, ü, °, ß, and even カナ) can require special kinds of string variables that support Unicode text encoding. Strings can be of arbitrary length—that is, they can contain no characters at all, a single character, or many characters.

Variables as containers for other variables

Arrays and lists

An **array** or matrix is a collection of data that lives under the roof of a single variable. Arrays can consist of one dimension, such as the set of integers from 0 to 9; two dimensions, such as a grid describing the pixels of a black-and-white photograph, or a record of depth, temperature, and salinity over time; or three or more dimensions. One-dimensional arrays are often referred to as **lists** or **vectors**. Depending on the computer language, an array or list can contain a mixture of types, including other lists. For example, a list called `morphology` could include floats, integers, string descriptions, and lists:

```
Morphology=[1, 0, -2, 5.27, 'blue', [4,2,4]]
```

Lists of lists can store multidimensional data, such as two-dimensional matrices with columns and rows. Lists come with useful tools for finding particular values, extracting elements, sorting data, performing some function on each datum, and other similar tasks.

Each item in an array can be accessed by referring to its location within the matrix, usually with an index number contained in brackets. In many languages, the first element is index `0`, rather than `1`. For a one-dimensional list such as `[A,G,T,C]`, the index is just the number referring the position in the list (in this case, `0` for `A`, `1` for `G`, `2` for `T`, and `3` for `C`). With a two-dimensional matrix that looks like a checkerboard (Figure 7.1), you can specify an entire row or column of data, or just a single element by giving its row and column indices. To change elements in a list, you can usually use the same syntax that is used to access the values, with the list item on the left and the value to be assigned on the right. For example:

$$A=\begin{bmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{bmatrix}$$

FIGURE 7.1 A multidimensional array

```
MyArray[2] = 5
```

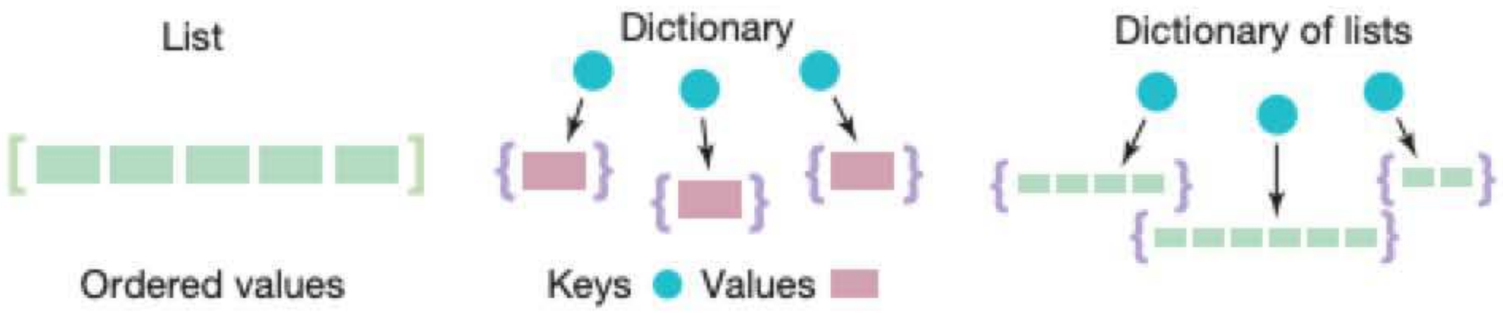

FIGURE 7.2 The structures of a list, a dictionary, and a dictionary of lists

Arrays are very important in scientific programming, making the code more efficient and easier to write and understand. If you find yourself repeating the same operation or calculation on a long series of values—especially if you are copying and pasting names like `Species1`, `Species2`, `Species3`—then this is the perfect time to learn how to store and process such variables as lists.

Dictionaries and associated arrays A **dictionary**, also known as an associative array, hash, or map, is another type of container for more than one variable. Unlike a list, which is simply a sequence of ordered values, a dictionary is a collection of names, or **keys**, with each key pointing to an associated **value** (Figure 7.2). The keys can be numbers, strings, or other types of variables. The keys for a dictionary must be unique, since each key can point to only one value, but different keys can have the same value. Here is some example Python code showing the creation of a dictionary, followed by the assigning of key-value pairs:

```
TreeDiam={} ← Create an empty dictionary with {}
TreeDiam['Kodiak'] = [68]
TreeDiam['Juneau'] = [85]
```

True to the dictionary analogy, values in dictionaries are looked up according to their keys, rather than by their position, as would happen in a list. For example, if you used a dictionary type to create and populate an actual dictionary, you could find the value for "cnidarian" using the word itself as the key, rather having to know that it was the 1024th item in the alphabetical list of definitions. In fact, the elements of data in a dictionary are in no particular order, and can be accessed *only* by looking up their key word.

Dictionaries are especially useful for gathering together data that describe some properties of a series of entities. For instance, you could create a dictionary containing all the molecular weights of amino acids, allowing you to easily retrieve the values for each amino acid by its name. Another handy trick is to combine dictionaries and lists, making a dictionary of lists. For instance, you could associate lists of DNA sequences from several different genes with particular specimen numbers.

Converting between types

Some programming languages, including C and C++, require the programmer to explicitly state variable types and then strictly enforce these types thereafter. In

other words, somewhere in the program before you use **x**, you have to specify that **x** is going to hold an integer. This makes for robust programs, since the computer never has to guess what type was intended, but it tends to reduce flexibility and require more code. Other languages, such as Python, set the type of the variable according to the values that are assigned to them, and only then strictly enforce these types. This is also quite robust, and means that the programmer still needs to think about variable types even though they aren't explicitly specified. Still other languages, for example Perl, try to take care of all type-related issues behind the scenes. The problem with this less restrictive approach is that code can break in very confusing ways when things go wrong. Although it can be convenient to have types handled entirely in the background, in many cases your intentions may be ambiguous and the computer's best guess is not what you expected.

This is particularly evident when you have a string containing a series of characters that could represent a number. For instance, take the string `"123"`. (Remember that the contents of a string are always contained in quotes.) To the computer, this is a sequence of characters, just like any other sequence, and not a number at all. A variable that contains this set of three text characters is very different from the integer 123; the latter is stored in the computer's memory as a binary representation of the number 123, not as a sequence of characters. If you were to add the integer 5 to the string `"123"`, for instance, would you expect to have the `"123"` treated as a number, resulting in 128, or the 5 treated as a string, resulting in `"1235"`?

In most programming languages, you would need to explicitly convert the variable types in this example to clarify the intended operation. That is, if you wanted the result to be the number 128, you would write the equivalent of, "Add `5` to a numerical interpretation of the string `'123'`"; whereas if you wanted the result to be the string `'1235'`, you would write the equivalent of, "Append a string interpretation of the integer `5` to the end of the string `'123'`." Because this kind of thing occurs often, there are commands built into nearly all languages to perform such conversions.

Variables in action

Programs need to do a lot more than just store information in variables. They also need to be able to do something useful with them. Often this consists of performing some kind of calculation to generate a new value. The tools for modifying or calculating new values include **operators** and **functions**.

Mathematical operators

Operators in computer programs include the mathematical operators that you are already familiar with: addition, subtraction, multiplication, division, power (x^y), and modulo (finding the remainder left after division).

The actions that operators perform depend on the language and the type of data they are operating on. Usually this is quite intuitive. Take the + operator:

If it is applied to two integers, it will return their sum. In many languages, + can also be applied to strings, as alluded to in the previous section, to produce a new string that contains the joined contents of the other strings. Although the operator is written as + in both cases, it is performing different tasks in each context.

The data that an operator acts on don't necessarily have to be of the same type. The + operator can add a float to an integer, for example, but not all combinations of data types are supported by all operators: the + operator cannot combine a string and an integer.

In some languages, including Python, operations on integers alone will often generate integers, perhaps even when you were expecting a float. The most trouble comes with the division operator, `/`. The precise result for `5/2` would of course be `2.5`, which needs to be stored as a floating point number. When the result is shoe-horned into an integer variable, however, it truncates the decimal portion, leaving the somewhat confusing result of 2, with the remainder having been discarded.

The operators discussed so far all do a calculation and return some new value as the result. One operator that we have taken for granted so far is the = operator, which assigns the value on the right side of = to the variable on the left side. This allows you to copy the value of one variable into another, or to store a new value.

Comparative and logical operators

Other operators make comparisons of variables and then return a Boolean value in the form of `True` or `False`. A common example is to test if one variable is greater than another. The results of operators can be assigned to a variable, or just used directly to help the program decide what to do.

Comparison operators can report whether two entities are the same or not. The equality operator is often written as ==, and returns `True` if the entities on the left and right side of the operator have the same value. This is different from a single =, which assigns values rather than comparing them. Though it is common to think of = alone doing both of these tasks, most languages will not allow its use in both contexts.

In some languages, equality operators can even apply to strings. The statement `GenusName == 'Falco'`, for example, would return `True` if the variable named `GenusName` contains a string with the value `'Falco'`. These kinds of equality comparisons have to be exact, including whether the characters in the string are uppercase or lowercase. If you want to test whether a bit of text partially matches another string, there are other string-related functions which work more like the regular expressions of Chapter 2 and the `grep` command in Chapter 5.

Another useful operator which returns a Boolean value is the `in` operator (some operators are words, not punctuation marks). For example, the expression `x in A` returns `True` if the value of `x` is contained among the items in list `A`. If `A` is a list of several lists, then `x` must be a list as well, not one of the elements of those lists.

Other logic-related operators work with Boolean variables themselves. The names of these operators (`and`, `or`, `not`) describe their mode of operation pretty

well. These are important in testing several comparisons simultaneously, for example (`Depth > 700 and Oxygen < 0.1`).

Operators are evaluated according to a hierarchical order, per normal algebraic rules. Usually the order is exponents; multiplication and division; addition and subtraction; comparisons of equality; and finally, logical operators. Within these

TABLE 7.3 Common operators and their symbols

Operator*	Common symbols
Mathematical	
Addition	+
Subtraction	-
Multiplication	*
Division	/
Power	**
Modulo (remainder after division)	%
Truncated division (result without remainder)	//
Comparative	
Equal to	==
Not equal to	!=, <>, ~=
Greater than	>
Less than	<
Greater or equal	>=
Less or equal	<=
Logical	
And	and, &, &&
Or	or, |, ||
Not	not, !, ~

*Not all languages support all operator symbols, and in some cases the name of the operator is used instead of a symbol.

categories, operators are evaluated left to right. This order can be modified with parentheses, grouping operations that should be performed first. The most commonly used operators are listed in Table 7.3.

Functions

Functions can be thought of as little stand-alone programs that are called from within your own program. Many functions for common tasks come built into programming languages. You can also write your own custom functions; these can be reused as often as you want, both within a program and potentially later on by other programs that you write. Functions can be defined locally in your program, or they can be stored in external files and loaded into your program as needed. Functions will be an important part of all but the simplest of the programs you write.

Functions can accept variables, which are then referred to as **parameters** of the function. These are usually sent to the function within parentheses, `()`. Functions can also return values. For example, the `round()` function, as used in the context of `y = round(2.718)`, takes a number with a fractional portion (in this case the floating point value `2.718`) and returns the value after the decimal is rounded off (here, rounded number is assigned to the variable `y`). Even when functions aren't designed to take any parameters, a set of trailing empty parentheses, e.g., `PrintHelpInfo()`, is still usually needed to differentiate the name as pointing to a function, rather than to a variable or other code element.

Flow control

Decisions with the `if` statement

At this point you know a bit about variables and how to obtain or create values based on their state, through comparisons and calculations. With just these tools alone, though, you couldn't write much more than a glorified calculator. The real power of programming becomes apparent with conditional decision-making—the ability for a program to follow different courses of action depending on the state of variables.

The `if` statement is the most widely used building block for such decision-making. It is like a sign at a fork in the road. If the statement on the sign is true, then you take one fork, and if it is false you take another route, eventually rejoining the main sequence of commands within the program. In programming terms, the `if` statement evaluates an expression that returns a Boolean. If that expression is true, it then executes a specified set of commands. You can use an `else` statement in conjunction with an `if` statement to execute another set of commands when the expression is false. You can also chain together a series of `if-else` statements to combine several of the logical conditions into a complex sequence of events. Figure 7.3 depicts both an `if` and an `if-else` statement.

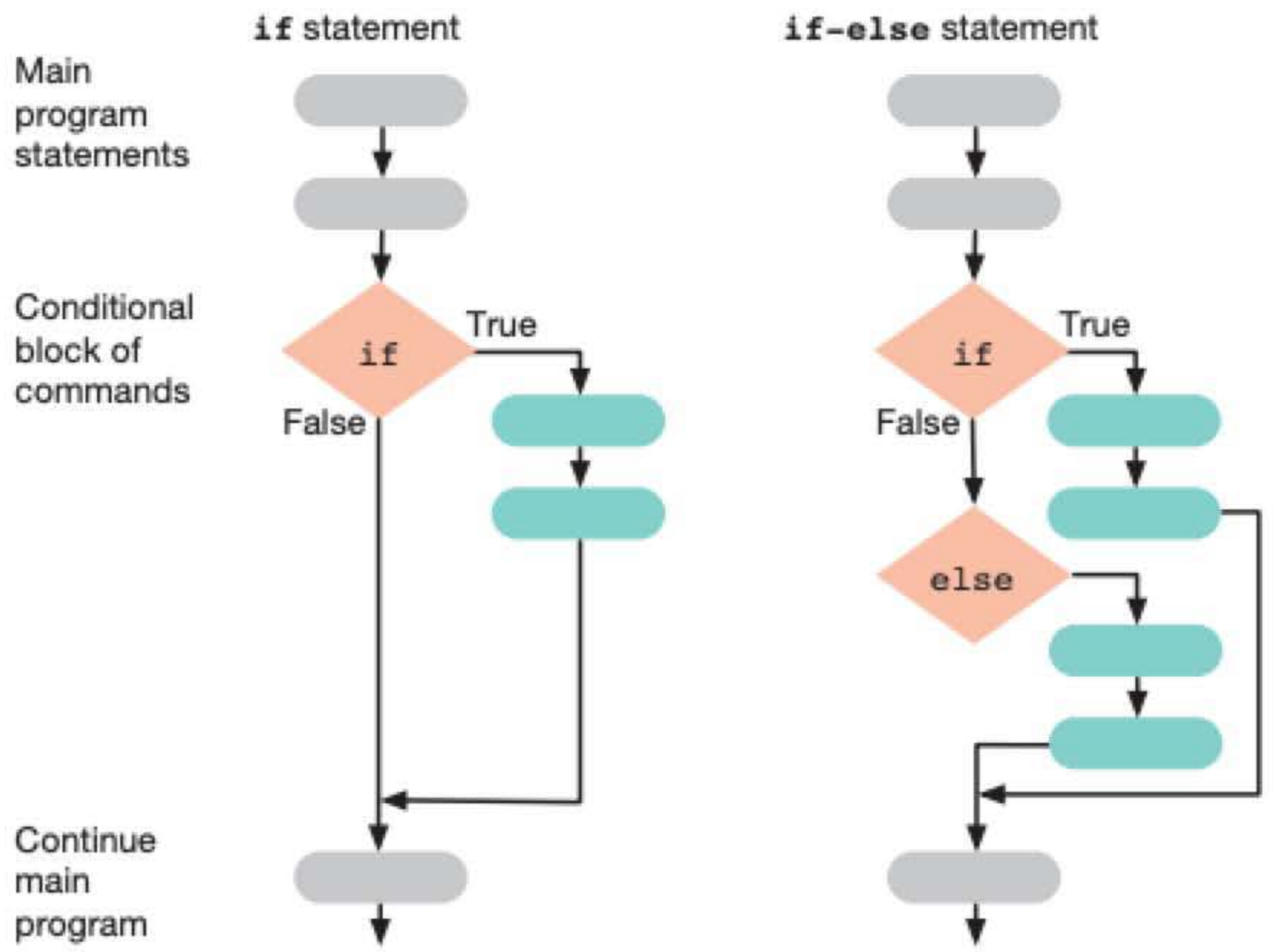

FIGURE 7.3 The flow of a program through conditional statements The ovals represent program statements and the diamonds represent decision points. Arrows represent the operation of the program under different circumstances. In an `if-else` statement, for example, if a statement is evaluated as `True`, you take the `if` fork; but if it evaluated as `False`, you take the `else` fork.

Below is an example of an `if-else` statement as it would be written in Python; other languages will have slightly different ways of stating the same thing:

```
A = 5
if A < 0:
    print "Negative number"
else:
    print "Zero or positive number"
```

Looping with `for` *and* `while`

The building blocks presented so far allow you to write a program consisting of a linear sequence of events to be executed from top to bottom, with the power to control which events in this sequence are executed. A linear program like this can automate a complicated sequence of analyses, but it isn't well-suited to repetitious tasks that require the same computations to be made over and over. This, however, is the exact kind of task you usually want to write a program for. The power of a program to plow through hundreds of calculations in only a few lines of code is realized by **looping** repeatedly through the same basic set of commands. A `for` loop and a `while` loop are illustrated schematically in Figure 7.4.

The `for` loop **Loops** are portions of programs that are repeatedly executed until some condition is met. Of these, by far the most widely used is the `for` loop. A `for` loop cycles through each item of a predefined collection, performing a series of commands each time it does so. The items it cycles through can be drawn

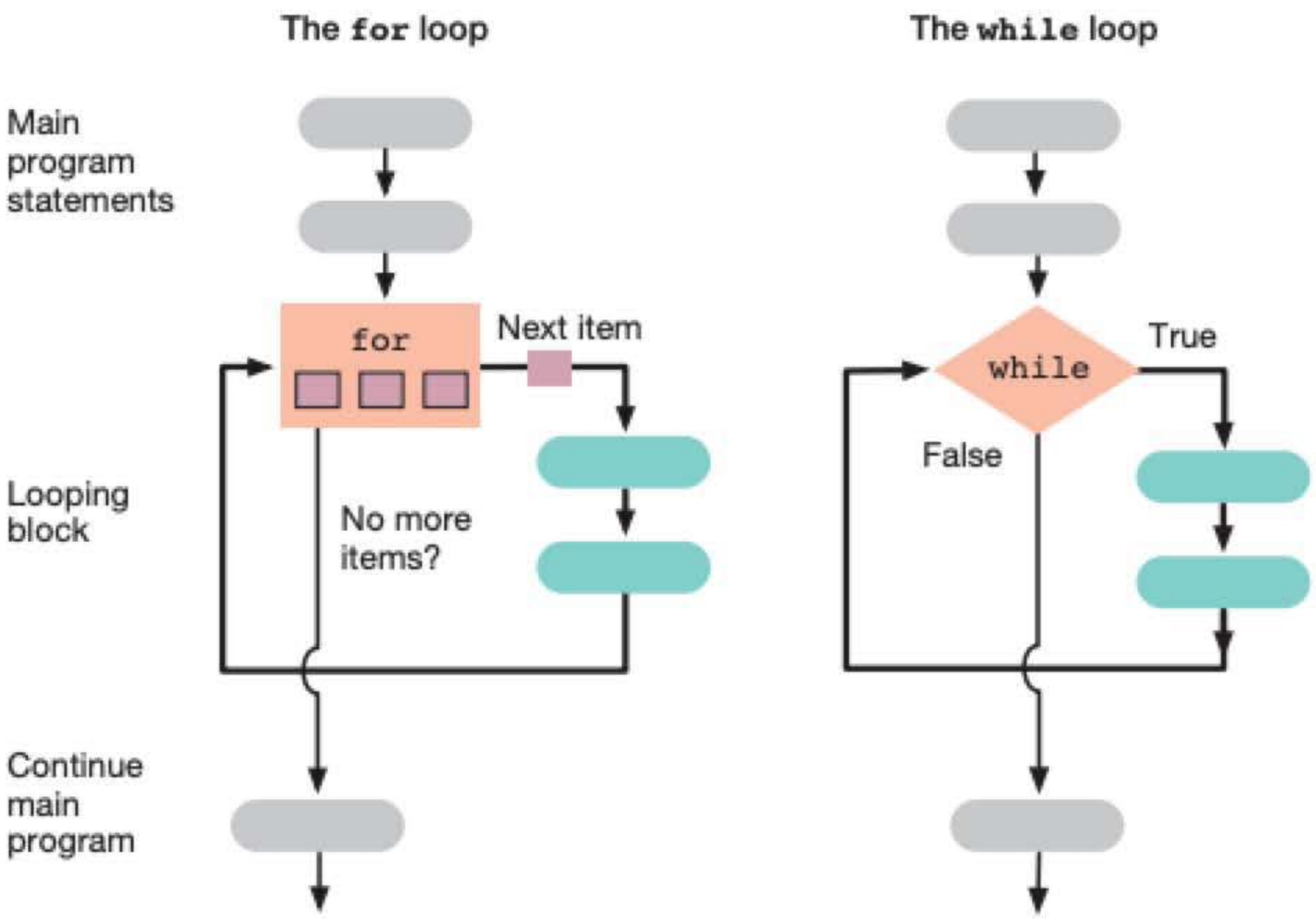

FIGURE 7.4 The flow of a program with a `for` or `while` loop The orange box represents a collection of objects which are taken one-by-one and operated on by the blue ovals.

from a variety of collections, such as a range of numbers from 1 to 100, a list of species names, an archive of protein sequences, the lines of a data file, and so on.

The power of a `for` loop comes from being able to repeat the same operation on each item in a long list. In fact, some programming languages have a specific `foreach` statement emphasizing this item-by-item capability. Loops can also be nested, or placed one inside the other. For instance, you might use three nested `for` loops to step through each amino acid of each protein sequence in each file of a folder.

As much as any programming element, the syntax used in `for` loops varies from one language to another. Many examples of this variation can be seen in Appendix 5. Two examples in particular are shown in Figure 7.5, one from Python and the other from C; note how verbose the C code is compared to the Python code. This may help you understand why we prefer Python for our work, and why we chose it as the programming language for this book.

(A) `for` loop in Python

```
for Num in range(10):
    print Num * 10
```

(B) `for` loop in C

```
for (Num=0; Num < 10; Num++) {
    printf("%d",Num * 10);
}
```

FIGURE 7.5 Code snippets showing a `for` loop in Python and in C

Innumerable situations can be tamed with `for` loops: processing every file that meets a certain criterion, splitting each line of a data file into its component parts, translating each gene sequence in a FASTA file, or performing a unit conversion on each entry of a column of numbers in a data-logger file.

The `while` loop In some situations, you enter a loop without having a predetermined list to cycle through. When a loop is open-ended like this, you can use a `while` loop to continue cycling through until some logical condition is evaluated as `False`. Like an `if` statement, the `while` loop begins with a logical test, but unlike the `if`, when the conditional block of code is complete, the program does not return to the main path, but loops back up and checks the condition again.

At least one variable involved in the test expression must be modified by the statements in the loop, or else the loop will keep cycling forever. This is the infamous "infinite loop." To avoid this situation, programmers sometimes link in a second fail-safe condition. Typically this is a counter—that is, a variable which keeps count of how many times the loop has been traversed. The loop can be made to exit when the count exceeds the expected maximum iterations.

Situations where `while` loops are useful include improving an approximation until it reaches a sufficient level of precision, waiting until a temperature or environmental condition is reached, continuously monitoring a sensor until a stop command is given, and reading lines from a file until the end is reached.

Using lists and dictionaries

Lists

Think of a list as a series of numbered boxes that start with box 0. You can create a list with a given number of boxes, and later append more boxes to the end of this series if you need more containers. A list that includes a box numbered 16 will necessarily have a box numbered 10. If you try to access box number 94 when there are only 70 boxes, though, an error will be generated. The data in a list can be accessed one at a time, or ranges of data can be accessed at once. You can peek inside any box you like, replace the contents of a box, remove or insert boxes into the beginning or end of the list (shifting the remaining boxes), or even reorder the boxes. After such modifications, the number used to access each box depends only on its position.

When do you use lists as opposed to singular variables? Pretty much any time you are handling more than one piece of closely related information. Imagine, for instance, a field site at which you measure the diameter of 5 trees. You might first think to load these values into a series of 5 variables named `Diameter1`, `Diameter2`, `...`, `Diameter5`, make calculations on each (`crosssection1=`, `crosssection2=`), and then print each result with five separate `print` statements. This is a cumbersome solution, since you would need to repeat a nearly identical set of commands to act on each variable independently for each tree. It is cleaner and easier to create a list variable called `Diameter` which holds the five

values in a list. By using a `for` loop, you can then easily write one set of commands which performs the same operations on each measurement in turn.

Lists also have the advantage of making your code more general. In our example, you may later want to use the same program to analyze a dataset with 6 measurements per site. If you used a different variable name for each measurement, you would have to add a variable to your program to accommodate the new dataset. If you use a list, however, your program would require no additional modifications. A list can handle thousands of values just as easily as it can handle ten.

Dictionaries

Dictionaries may seem a bit less intuitive than lists at first, but it is likely that you will find them to be equally useful, if not more so. Recall that the data entries of a dictionary are looked up with unique labels known as keys. If you wanted to perform an operation using each entry in a dictionary, you would first get a list of the keys, and then use those to retrieve each dictionary item one by one. Since in a dictionary, data elements don't have a fixed position, sorting the entries within the dictionary is not meaningful the way it would be with a list. However, sorting the keys and then retrieving the associated values in a particular order is often useful.

The unordered nature of dictionaries makes them more flexible than lists in several important ways. In a dictionary, you can create an entry for key `16` without having any data corresponding to keys `0` to `15`. This lets you fill in data as they become known. Although list indices must always be integers, dictionaries allow you to associate values with keys of many different variable types. For instance, you could use a dictionary where the keys are strings, each string being a specimen name:

```
TreeStat = {} ← Create an empty dictionary to build upon
TreeStat['Kodiak'] = [ 68, 57.8, -152.5]
TreeStat['Juneau'] = [ 85, 58.3, -134.5]
TreeStat['Barrow'] = [133, 71.3, -156.6]
```

Dictionaries and lists can be combined in useful ways. Imagine that each field site where you are measuring tree diameters has a unique name. You could create a dictionary `TreeStat` that uses the name of each site as the key, with a whole list of measurements and coordinates as the associated value for each entry. Such dictionaries of lists provide an easy way to store and return a large set of related measurements.

Other data types

One of the other basic data types, the **string**, is actually quite like a list in many respects. It is, after all, an ordered sequence of characters. You can step through each character of a string with a `for` loop, and you can even extract substrings or certain elements from a string in the same way you would extract subportions of a list. A difference is that you can't always modify individual elements of a string

the way you can those belonging to a list. Fortunately, there are other ways to carry out these kinds of operations on strings, as you will learn in later chapters.

Finally, there are other data types that hold collections of objects, such as sets and tuples in Python; however, these will not be addressed here.

Input and output

User interaction

Most programs provide for some means of user interaction. The vast majority of computer users have interacted with programs only through graphical user interfaces. Programs that are run at the command line (including the programs that you will soon write) also provide for several means of user interaction. Program options specified at the command line when a program is launched are called **arguments**. The command `ls`, for instance, can be run with several arguments that control what files are displayed, and in what format. These arguments are specified by listing them after the command when it is executed, for example `ls -a` or `ls *.txt`.

Arguments are a mode of user input for programs that first gather all the information they need, and then go about their business in one big push. This is good for the user because it isn't necessary to babysit the program while it runs, or to respond to queries during the execution. This is especially important for programs that take a long time to finish. If your program is configured with arguments, then it is also easier for other programs to control it. This will be important if you want to build up workflows that automatically shuttle data through a series of programs. Python and many other languages have built-in tools for passing arguments from the command line to variables within the programs you write.

Some command-line programs are designed to respond to user input while they are running. Command-line text editors, for example `nano`, are an elaborate example of this. Simple interactive interfaces can be desirable in some cases, and like arguments, they are simple to implement in your own software. For instance, you will create an interactive interface for the DNA analysis program described in the next chapters; this will allow you to calculate basic molecular properties of short sequences that you enter at the interactive prompt.

User interaction isn't just about getting input from the user; it is also about generating output, as well as giving the user feedback about the progress and results of the program. Typically, the command used to send output to the screen is either called `print` or some variation of that word. Output can also go directly into a file, a process we discuss next.

Files

Like variables, files associate some chunk of data with a name. Unlike variables, which have a fleeting existence while the program is running and thus live only in the computer's volatile memory (RAM), files reside on the disk drive. They are still there after the program stops, and even after the computer is turned off. While

variables are only accessible from within the program that created them, files are available to any program with access to the disk and permission to read them. Files therefore serve both as a long-term repository of data and a way to shuttle data between programs. Reading and writing files is an essential part of the programming you will do for your scientific research.

There are two separate levels of operations that must be completed to access data in a file. The first level consists of all of the mechanical details of gaining access to the contents of the file from within a program, such as finding it on the disk and loading its contents into memory. Fortunately, programming languages provide tools and commands that take care of all this work behind the scenes. In order to gain access to the contents of a file, you usually need to indicate only the path to the file and whether you want to read from it or write to it.

The second level of operations required to access data from a file is to establish relationships between the data in the file and variables in the program. This works in both directions. That is, when reading a file, you are **parsing** its data from a file into program variables, whereas when writing to a file, you are packaging data from program variables.

For example, a data file might contain lines of text like the following:

```
ConceptName     Depth  Latitude  Longitude Oxygen TempC  RecordedDate
Thalassocalyce   348.7  36.7180  -122.0574  1.48  7.165  1992-03-02 17:40:09
Thalassocalyce   520.3  36.7491  -122.0368  0.52  5.826  1992-05-05 20:01:10
Thalassocalyce   118.4  36.8385  -121.9676  1.52  7.465  1999-05-14 17:48:27
Thalassocalyce   100.9  36.7270  -122.0488  2.30  9.497  1999-08-09 20:12:11
Thalassocalyce  1509.6  36.5846  -122.5211  0.95  2.774  2000-04-17 20:04:23
```

This information, stored on the disk, will be most usable in the programming environment if each column can be loaded into list variables of appropriate types (strings, integers, floats, etc.). Conversely, once a calculation or conversion is performed, the variables will likely be written out to a file and saved.

Parsing and packaging file data usually comes down to string manipulation. The text is read from a file as strings, usually line-by-line, and the data are then extracted from these strings and placed in the appropriate variables. Regular expressions, the first tools you learned in this book, are often used for this process. Writing data to text files is the opposite process: data from variables are converted to strings, joined together with the appropriate formatting characters, and then written to the text file, once again line-by-line. The specifics of how you parse and package data will depend on the format of the data within the file, meaning that data from different sources, databases, or instruments will often require that you write or modify a small program to handle conversions and calculations. This, then, is where the heavy lifting of file handling occurs when you are programming.

In addition to storing data for analysis, files can also be a means of controlling program behavior. You have already seen how a text file can store a list of program commands as a script. Many programs can take their raw input and march-

ing orders from a file, after which they send their output directly to another file. Text files can also serve as a record of what a program did; in such cases, they are called log files, or just plain logs.

Libraries and modules

Computer languages have many built-in tools available out of the box. These include some of the basic building blocks we have mentioned—predefined variable types, basic functions, and simple mathematical and logical operators. For more specialized tasks, additional functions can be written and then bundled together in **modules**. By **importing** from one of these modules, you can gain access to the suite of functions stored within. Some of the most commonly used modules provide tools for more advanced mathematical functions (e.g., sin and log), interacting with the operating system (e.g., listing directory contents and executing shell commands), retrieving content from the Internet, performing regular expression searches, working with molecular sequences, and generating graphics. Python modules are explored in Chapter 12.

Comment statements

Programming languages provide for comments—that is, portions of text that are helpful to human beings but are to be ignored by the computer when the program is run. Comments are usually marked by starting the line with a special character, such as `#`, `//`, or `%`.

Comments serve several important functions for programmers. Commented blocks of text are used to provide documentation and a description of both what a program does and how to run it. Commented text at the top of a script usually includes a list of any arguments the program may take, and a revision history of changes, so that updated versions of the script can be distinguished.

Within the program, comments may be associated with individual lines or with subsections of code, to explain how they operate. These internal notes are important for future readers of your program (including yourself), who typically are trying to determine (or maybe just remember) how the program operates.

Comments are also useful for troubleshooting your program. You can isolate problems by turning portions of the code on and off with comments, to see if an error still occurs in the same way.

Objects

A full treatment of **objects** is beyond the scope of this book. However, objects are essential components of contemporary programming, and aspects of these components are woven into the fabric of many programming languages; thus, even a simple treatment needs to mention them.

An object is a sort of "super variable" that can contain several other variables within it as part of its definition. In fact, not only can objects contain variables in the traditional sense, but functions as well; in this context, these are called **methods**.

Consider a bicycle as an object. Your own personal bicycle may be just one example or instance of the platonic ideal of a bicycle. Likewise, your bike has many properties associated with it. In computer terms, you access the nested properties of an object using a **dot notation**, where the name of the property is joined to the name of the object with a period:

```
MyBike.color ← Color of your bike
MyBike.tires ← Properties of your tires
```

Used in this way, these statements might report the status of those properties, returning `'blue'` for color, as well as various values for the brand, air pressure, diameter and tread of your tires. These are examples of **nested** properties:

```
MyBike.tires.pressure ← A nested property of your tires
```

Methods (that is, functions) can also be associated with the object; if so, these are accessed with dot notation as well. These methods can be little programs, which take in values and apply them to the bicycle object:

```
MyBike.steer(-4) ← Steer 4° to the left
MyBike.color('red') ← You can often set and read with the same notation
MyBike.pedals.pedal(100) ← With the pedals, pedal at a speed of 100
```

The reason that these object-related ideas are important is that in many languages, even simple variables are objects and have their own methods and properties. Sometimes instead of using a separate function and sending your variable to it as a parameter, you can access a method from within the variable itself.

For example, imagine you have a string variable and you want to convert it to uppercase letters and print it. You might find a function `uppercaser()` and send your string into it:

```
MyString='abc'
print uppercaser(MyString)
```

Alternatively, the string variable itself might contain an uppercase function which you can access:

```
print MyString.upper()
```

There are many more implications and unique properties of object-oriented programming that are beyond the scope of this book. The main significance of objects for the topics we will address is that some functions you will use are methods of variables rather than stand-alone functions.

SUMMARY

You have learned:

- The differences between compiled and interpreted programs
- That variables have several attributes, including name, type, and value
- Some of the variable types widely used among programming languages
- That groups of variables can be stored in ordered lists and unordered dictionaries
- The basics of using variables with operators
- Key elements of the structure of programs, such as loops and functions