1

VARIABLES AND SIMPLE DATA TYPES

In this chapter you'll learn about the different kinds of data you can work with in your Python programs. You'll also learn how to store your data in variables and how to use those variables in your programs.

What Really Happens When You Run hello_world.py

Let's take a closer look at what Python does when you run *hello_world.py*. As it turns out, Python does a fair amount of work, even when it runs a simple program:

hello_world.py

```
print("Hello Python world!")
```

When you run this code, you should see this output:

```
Hello Python world!
```

When you run the file *hello_world.py*, the ending *.py* indicates that the file is a Python program. Your editor then runs the file through the *Python interpreter,* which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word `print`, it prints to the screen whatever is inside the parentheses.

As you write your programs, your editor highlights different parts of your program in different ways. For example, it recognizes that `print` is the name of a function and displays that word in blue. It recognizes that "Hello Python world!" is not Python code and displays that phrase in orange. This feature is called *syntax highlighting* and is quite useful as you start to write your own programs.

Variables

Let's try using a variable in *hello_world.py*. Add a new line at the beginning of the file, and modify the second line:

```
message = "Hello Python world!"
print(message)
```

Run this program to see what happens. You should see the same output you saw previously:

```
Hello Python world!
```

We've added a *variable* named `message`. Every variable holds a *value*, which is the information associated with that variable. In this case the value is the text "Hello Python world!"

Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the text "Hello Python world!" with the variable `message`. When it reaches the second line, it prints the value associated with `message` to the screen.

Let's expand on this program by modifying *hello_world.py* to print a second message. Add a blank line to *hello_world.py*, and then add two new lines of code:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Now when you run *hello_world.py*, you should see two lines of output:

```
Hello Python world!
Hello Python Crash Course world!
```

You can change the value of a variable in your program at any time, and Python will always keep track of its current value.

Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable *message_1* but not *1_message*.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, *greeting_message* works, but *greeting message* will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word `print`. (See "Python Keywords and Built-in Functions" on page 489.)
- Variable names should be short but descriptive. For example, *name* is better than *n*, *student_name* is better than *s_n*, and *name_length* is better than *length_of_persons_name*.
- Be careful when using the lowercase letter *l* and the uppercase letter *O* because they could be confused with the numbers *1* and *0*.

It can take some practice to learn how to create good variable names, especially as your programs become more interesting and complicated. As you write more programs and start to read through other people's code, you'll get better at coming up with meaningful names.

NOTE *The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but it's a good idea to avoid using them for now.*

Avoiding Name Errors When Using Variables

Every programmer makes mistakes, and most make mistakes every day. Although good programmers might create errors, they also know how to respond to those errors efficiently. Let's look at an error you're likely to make early on and learn how to fix it.

We'll write some code that generates an error on purpose. Enter the following code, including the misspelled word *mesage* shown in bold:

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A *traceback* is a record of where the interpreter ran into trouble when trying to execute your code. Here's an example of the traceback that Python provides after you've accidentally misspelled a variable's name:

```
Traceback (most recent call last):
❶  File "hello_world.py", line 2, in <module>
❷    print(mesage)
❸ NameError: name 'mesage' is not defined
```

The output at ❶ reports that an error occurs in line 2 of the file *hello_world.py*. The interpreter shows this line to help us spot the error quickly ❷ and tells us what kind of error it found ❸. In this case it found a *name error* and reports that the variable being printed, `mesage`, has not been defined. Python can't identify the variable name provided. A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name.

Of course, in this example we omitted the letter *s* in the variable name `message` in the second line. The Python interpreter doesn't spellcheck your code, but it does ensure that variable names are spelled consistently. For example, watch what happens when we spell *message* incorrectly in another place in the code as well:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

In this case, the program runs successfully!

```
Hello Python Crash Course reader!
```

Computers are strict, but they disregard good and bad spelling. As a result, you don't need to consider English spelling and grammar rules when you're trying to create variable names and writing code.

Many programming errors are simple, single-character typos in one line of a program. If you're spending a long time searching for one of these errors, know that you're in good company. Many experienced and talented programmers spend hours hunting down these kinds of tiny errors. Try to laugh about it and move on, knowing it will happen frequently throughout your programming life.

NOTE *The best way to understand new programming concepts is to try using them in your programs. If you get stuck while working on an exercise in this book, try doing something else for a while. If you're still stuck, review the relevant part of that chapter. If you still need help, see the suggestions in Appendix C.*

TRY IT YOURSELF

Write a separate program to accomplish each of these exercises. Save each program with a filename that follows standard Python conventions, using lowercase letters and underscores, such as *simple_message.py* and *simple_messages.py*.

2-1. Simple Message: Store a message in a variable, and then print that message.

2-2. Simple Messages: Store a message in a variable, and print that message. Then change the value of your variable to a new message, and print the new message.

Strings

Because most programs define and gather some sort of data, and then do something useful with it, it helps to classify different types of data. The first data type we'll look at is the string. Strings are quite simple at first glance, but you can use them in many different ways.

A *string* is simply a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

name.py

```
name = "ada lovelace"
print(name.title())
```

Save this file as *name.py*, and then run it. You should see this output:

```
Ada Lovelace
```

In this example, the lowercase string `"ada lovelace"` is stored in the variable `name`. The method `title()` appears after the variable in the `print()` statement. A *method* is an action that Python can perform on a piece of data. The dot (`.`) after `name` in `name.title()` tells Python to make the `title()` method act on the variable `name`. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty.

`title()` displays each word in titlecase, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values `Ada`, `ADA`, and `ada` as the same name, and display all of them as `Ada`.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

This will display the following:

```
ADA LOVELACE
ada lovelace
```

The `lower()` method is particularly useful for storing data. Many times you won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

Combining or Concatenating Strings

It's often useful to combine strings. For example, you might want to store a first name and a last name in separate variables, and then combine them when you want to display someone's full name:

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = first_name + " " + last_name

print(full_name)
```

Python uses the plus symbol (+) to combine strings. In this example, we use + to create a full name by combining a `first_name`, a space, and a `last_name` ❶, giving this result:

```
ada lovelace
```

This method of combining strings is called *concatenation*. You can use concatenation to compose complete messages using the information you've stored in a variable. Let's look at an example:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name

❶ print("Hello, " + full_name.title() + "!")
```

Here, the full name is used at ❶ in a sentence that greets the user, and the `title()` method is used to format the name appropriately. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

You can use concatenation to compose a message and then store the entire message in a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name

❶ message = "Hello, " + full_name.title() + "!"
❷ print(message)
```

This code displays the message "Hello, Ada Lovelace!" as well, but storing the message in a variable at ❶ makes the final `print` statement at ❷ much simpler.

Adding Whitespace to Strings with Tabs or Newlines

In programming, *whitespace* refers to any nonprinting character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.

To add a tab to your text, use the character combination `\t` as shown at ❶:

```
>>> print("Python")
Python
❶ >>> print("\tPython")
    Python
```

To add a newline in a string, use the character combination `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

You can also combine tabs and newlines in a single string. The string `"\n\t"` tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Newlines and tabs will be very useful in the next two chapters when you start to produce many lines of output from just a few lines of code.

Stripping Whitespace

Extra whitespace can be confusing in your programs. To programmers `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.

It's important to think about whitespace, because often you'll want to compare two strings to determine whether they are the same. For example, one important instance might involve checking people's usernames when they log in to a website. Extra whitespace can be confusing in much simpler situations as well. Fortunately, Python makes it easy to eliminate extraneous whitespace from data that people enter.

Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the `rstrip()` method.

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
  'python '
❸ >>> favorite_language.rstrip()
  'python'
❹ >>> favorite_language
  'python '
```

The value stored in `favorite_language` at ❶ contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value ❷. When the `rstrip()` method acts on the variable `favorite_language` at ❸, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, you can see that the string looks the same as when it was entered, including the extra whitespace ❹.

To remove the whitespace from the string permanently, you have to store the stripped value back into the variable:

```
  >>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
  >>> favorite_language
  'python'
```

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then store that value back in the original variable, as shown at ❶. Changing a variable's value and then storing the new value back in the original variable is done often in programming. This is how a variable's value can change as a program is executed or in response to user input.

You can also strip whitespace from the left side of a string using the `lstrip()` method or strip whitespace from both sides at once using `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
  ' python'
❸ >>> favorite_language.lstrip()
  'python '
❹ >>> favorite_language.strip()
  'python'
```

In this example, we start with a value that has whitespace at the beginning and the end ❶. We then remove the extra space from the right side at ❷, from the left side at ❸, and from both sides at ❹. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Avoiding Syntax Errors with Strings

One kind of error that you might see with some regularity is a syntax error. A *syntax error* occurs when Python doesn't recognize a section of your program as valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Here's how to use single and double quotes correctly. Save this program as *apostrophe.py* and then run it:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

```
One of Python's strengths is its diverse community.
```

However, if you use single quotes, Python can't identify where the string should end:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

You'll see the following output:

```
  File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
                             ^❶
SyntaxError: invalid syntax
```

In the output you can see that the error occurs at ❶ right after the second single quote. This *syntax error* indicates that the interpreter doesn't recognize something in the code as valid Python code. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct. If you get stuck on a particularly stubborn error, see the suggestions in Appendix C.

NOTE *Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.*

Printing in Python 2

The `print` statement has a slightly different syntax in Python 2:

```
>>> python2.7
>>> print "Hello Python 2.7 world!"
Hello Python 2.7 world!
```

Parentheses are not needed around the phrase you want to print in Python 2. Technically, `print` is a function in Python 3, which is why it needs parentheses. Some Python 2 `print` statements do include parentheses, but the behavior can be a little different than what you'll see in Python 3. Basically, when you're looking at code written in Python 2, expect to see some `print` statements with parentheses and some without.

TRY IT YOURSELF

Save each of the following exercises as a separate file with a name like *name_cases.py*. If you get stuck, take a break or see the suggestions in Appendix C.

2-3. Personal Message: Store a person's name in a variable, and print a message to that person. Your message should be simple, such as, "Hello Eric, would you like to learn some Python today?"

2-4. Name Cases: Store a person's name in a variable, and then print that person's name in lowercase, uppercase, and titlecase.

2-5. Famous Quote: Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:

> *Albert Einstein once said, "A person who never made a mistake never tried anything new."*

2-6. Famous Quote 2: Repeat Exercise 2-5, but this time store the famous person's name in a variable called `famous_person`. Then compose your message and store it in a new variable called `message`. Print your message.

2-7. Stripping Names: Store a person's name, and include some whitespace characters at the beginning and end of the name. Make sure you use each character combination, `"\t"` and `"\n"`, at least once.

Print the name once, so the whitespace around the name is displayed. Then print the name using each of the three stripping functions, `lstrip()`, `rstrip()`, and `strip()`.

Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they are being used. Let's first look at how Python manages integers, because they are the simplest to work with.

Integers

You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

The spacing in these examples has no effect on how Python evaluates the expressions; it simply helps you more quickly spot the operations that have priority when you're reading through the code.

Floats

Python calls any number with a decimal point a *float*. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number. Every programming language must

be carefully designed to properly manage decimal numbers so numbers behave appropriately no matter where the decimal point appears.

For the most part, you can use decimals without worrying about how they behave. Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

But be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally. Just ignore the extra decimal places for now; you'll learn ways to deal with the extra places when you need to in the projects in Part II.

Avoiding Type Errors with the str() Function

Often, you'll want to use a variable's value within a message. For example, say you want to wish someone a happy birthday. You might write code like this:

birthday.py

```
age = 23
message = "Happy " + age + "rd Birthday!"

print(message)
```

You might expect this code to print the simple birthday greeting, `Happy 23rd birthday!` But if you run this code, you'll see that it generates an error:

```
Traceback (most recent call last):
  File "birthday.py", line 2, in <module>
    message = "Happy " + age + "rd Birthday!"
❶ TypeError: Can't convert 'int' object to str implicitly
```

This is a *type error*. It means Python can't recognize the kind of information you're using. In this example Python sees at ❶ that you're using a variable that has an integer value (`int`), but it's not sure how to interpret that

value. Python knows that the variable could represent either the numerical value 23 or the characters *2* and *3*. When you use integers within strings like this, you need to specify explicitly that you want Python to use the integer as a string of characters. You can do this by wrapping the variable in the `str()` function, which tells Python to represent non-string values as strings:

```
age = 23
message = "Happy " + str(age) + "rd Birthday!"

print(message)
```

Python now knows that you want to convert the numerical value 23 to a string and display the characters `2` and `3` as part of the birthday message. Now you get the message you were expecting, without any errors:

```
Happy 23rd Birthday!
```

Working with numbers in Python is straightforward most of the time. If you're getting unexpected results, check whether Python is interpreting your numbers the way you want it to, either as a numerical value or as a string value.

Integers in Python 2

Python 2 returns a slightly different result when you divide two integers:

```
>>> python2.7
>>> 3 / 2
1
```

Instead of `1.5`, Python returns `1`. Division of integers in Python 2 results in an integer with the remainder truncated. Note that the result is not a rounded integer; the remainder is simply omitted.

To avoid this behavior in Python 2, make sure that at least one of the numbers is a float. By doing so, the result will be a float as well:

```
>>> 3 / 2
1
>>> 3.0 / 2
1.5
>>> 3 / 2.0
1.5
>>> 3.0 / 2.0
1.5
```

This division behavior is a common source of confusion when people who are used to Python 3 start using Python 2, or vice versa. If you use or create code that mixes integers and floats, watch out for irregular behavior.

TRY IT YOURSELF

2-8. Number Eight: Write addition, subtraction, multiplication, and division operations that each result in the number 8. Be sure to enclose your operations in `print` statements to see the results. You should create four lines that look like this:

```
print(5 + 3)
```

Your output should simply be four lines with the number 8 appearing once on each line.

2-9. Favorite Number: Store your favorite number in a variable. Then, using that variable, create a message that reveals your favorite number. Print that message.

Comments

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A *comment* allows you to write notes in English within your programs.

How Do You Write Comments?

In Python, the hash mark (#) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter. For example:

comment.py

```
# Say hello to everyone.
print("Hello Python people!")
```

Python ignores the first line and executes the second line.

```
Hello Python people!
```

What Kind of Comments Should You Write?

The main reason to write comments is to explain what your code is supposed to do and how you are making it work. When you're in the middle of working on a project, you understand how all of the pieces fit together. But when you return to a project after some time away, you'll likely have forgotten some of the details. You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach in clear English.

If you want to become a professional programmer or collaborate with other programmers, you should write meaningful comments. Today, most software is written collaboratively, whether by a group of employees at one company or a group of people working together on an open source project. Skilled programmers expect to see comments in code, so it's best to start adding descriptive comments to your programs now. Writing clear, concise comments in your code is one of the most beneficial habits you can form as a new programmer.

When you're determining whether to write a comment, ask yourself if you had to consider several approaches before coming up with a reasonable way to make something work; if so, write a comment about your solution. It's much easier to delete extra comments later on than it is to go back and write comments for a sparsely commented program. From now on, I'll use comments in examples throughout this book to help explain sections of code.

TRY IT YOURSELF

2-10. Adding Comments: Choose two of the programs you've written, and add at least one comment to each. If you don't have anything specific to write because your programs are too simple at this point, just add your name and the current date at the top of each program file. Then write one sentence describing what the program does.

The Zen of Python

For a long time, the programming language Perl was the mainstay of the Internet. Most interactive websites in the early days were powered by Perl scripts. The Perl community's motto at the time was, "There's more than one way to do it." People liked this mind-set for a while, because the flexibility written into the language made it possible to solve most problems in a variety of ways. This approach was acceptable while working on your own projects, but eventually people realized that the emphasis on flexibility made it difficult to maintain large projects over long periods of time. It was difficult, tedious, and time-consuming to review code and try to figure out what someone else was thinking when they were solving a complex problem.

Experienced Python programmers will encourage you to avoid complexity and aim for simplicity whenever possible. The Python community's philosophy is contained in "The Zen of Python" by Tim Peters. You can access this brief set of principles for writing good Python code by entering `import this` into your interpreter. I won't reproduce the entire "Zen of

Python" here, but I'll share a few lines to help you understand why they should be important to you as a beginning Python programmer.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

Python programmers embrace the notion that code can be beautiful and elegant. In programming, people solve problems. Programmers have always respected well-designed, efficient, and even beautiful solutions to problems. As you learn more about Python and use it to write more code, someone might look over your shoulder one day and say, "Wow, that's some beautiful code!"

```
Simple is better than complex.
```

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

```
Complex is better than complicated.
```

Real life is messy, and sometimes a simple solution to a problem is unattainable. In that case, use the simplest solution that works.

```
Readability counts.
```

Even when your code is complex, aim to make it readable. When you're working on a project that involves complex coding, focus on writing informative comments for that code.

```
There should be one-- and preferably only one --obvious way to do it.
```

If two Python programmers are asked to solve the same problem, they should come up with fairly compatible solutions. This is not to say there's no room for creativity in programming. On the contrary! But much of programming consists of using small, common approaches to simple situations within a larger, more creative project. The nuts and bolts of your programs should make sense to other Python programmers.

```
Now is better than never.
```

You could spend the rest of your life learning all the intricacies of Python and of programming in general, but then you'd never complete any projects. Don't try to write perfect code; write code that works, and then decide whether to improve your code for that project or move on to something new.

As you continue to the next chapter and start digging into more involved topics, try to keep this philosophy of simplicity and clarity in mind. Experienced programmers will respect your code more and will be happy to give you feedback and collaborate with you on interesting projects.

TRY IT YOURSELF

2-11. Zen of Python: Enter `import this` into a Python terminal session and skim through the additional principles.

Summary

In this chapter you learned to work with variables. You learned to use descriptive variable names and how to resolve name errors and syntax errors when they arise. You learned what strings are and how to display strings using lowercase, uppercase, and titlecase. You started using whitespace to organize output neatly, and you learned to strip unneeded whitespace from different parts of a string. You started working with integers and floats, and you read about some unexpected behavior to watch out for when working with numerical data. You also learned to write explanatory comments to make your code easier for you and others to read. Finally, you read about the philosophy of keeping your code as simple as possible, whenever possible.

In Chapter 3 you'll learn to store collections of information in variables called *lists*. You'll learn to work through a list, manipulating any information in that list.

2
INTRODUCING LISTS

In this chapter and the next you'll learn what lists are and how to start working with the elements in a list. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

What Is a List?

A *list* is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you want into a list, and

the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as `letters`, `digits`, or `names`.

In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Because this isn't the output you want your users to see, let's learn how to access the individual items in a list.

Accessing Elements in a List

Lists are ordered collections, so you can access any element in a list by telling Python the position, or *index*, of the item desired. To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets.

For example, let's pull out the first bicycle in the list `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0])
```

The syntax for this is shown at ❶. When we ask for a single item from a list, Python returns just that element without square brackets or quotation marks:

```
trek
```

This is the result you want your users to see—clean, neatly formatted output.

You can also use the string methods from Chapter 2 on any element in a list. For example, you can format the element `'trek'` more neatly by using the `title()` method:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

This example produces the same output as the preceding example except `'Trek'` is capitalized.

Index Positions Start at 0, Not 1

Python considers the first item in a list to be at position 0, not position 1. This is true of most programming languages, and the reason has to do with how the list operations are implemented at a lower level. If you're receiving unexpected results, determine whether you are making a simple off-by-one error.

The second item in a list has an index of 1. Using this simple counting system, you can get any element you want from a list by subtracting one from its position in the list. For instance, to access the fourth item in a list, you request the item at index 3.

The following asks for the bicycles at index `1` and index `3`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

This code returns the second and fourth bicycles in the list:

```
cannondale
specialized
```

Python has a special syntax for accessing the last element in a list. By asking for the item at index `-1`, Python always returns the last item in the list:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

This code returns the value `'specialized'`. This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well. The index `-2` returns the second item from the end of the list, the index `-3` returns the third item from the end, and so forth.

Using Individual Values from a List

You can use individual values from a list just as you would any other variable. For example, you can use concatenation to create a message based on a value from a list.

Let's try pulling the first bicycle from the list and composing a message using that value.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = "My first bicycle was a " + bicycles[0].title() + "."

print(message)
```

At ❶, we build a sentence using the value at `bicycles[0]` and store it in the variable `message`. The output is a simple sentence about the first bicycle in the list:

```
My first bicycle was a Trek.
```

TRY IT YOURSELF

Try these short programs to get some firsthand experience with Python's lists. You might want to create a new folder for each chapter's exercises to keep them organized.

3-1. Names: Store the names of a few of your friends in a list called `names`. Print each person's name by accessing each element in the list, one at a time.

3-2. Greetings: Start with the list you used in Exercise 3-1, but instead of just printing each person's name, print a message to them. The text of each message should be the same, but each message should be personalized with the person's name.

3-3. Your Own List: Think of your favorite mode of transportation, such as a motorcycle or a car, and make a list that stores several examples. Use your list to print a series of statements about these items, such as "I would like to own a Honda motorcycle."

Changing, Adding, and Removing Elements

Most lists you create will be dynamic, meaning you'll build a list and then add and remove elements from it as your program runs its course. For example, you might create a game in which a player has to shoot aliens out of the sky. You could store the initial set of aliens in a list and then remove an alien from the list each time one is shot down. Each time a new alien appears on the screen, you add it to the list. Your list of aliens will decrease and increase in length throughout the course of the game.

Modifying Elements in a List

The syntax for modifying an element is similar to the syntax for accessing an element in a list. To change an element, use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have.

For example, let's say we have a list of motorcycles, and the first item in the list is `'honda'`. How would we change the value of this first item?

motorcycles.py

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ motorcycles[0] = 'ducati'
  print(motorcycles)
```

The code at ❶ defines the original list, with `'honda'` as the first element. The code at ❷ changes the value of the first item to `'ducati'`. The output shows that the first item has indeed been changed, and the rest of the list stays the same:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

You can change the value of any item in a list, not just the first item.

Adding Elements to a List

You might want to add a new element to a list for many reasons. For example, you might want to make new aliens appear in a game, add new data to a visualization, or add new registered users to a website you've built. Python provides several ways to add new data to existing lists.

Appending Elements to the End of a List

The simplest way to add a new element to a list is to *append* the item to the list. When you append an item to a list, the new element is added to the end of the list. Using the same list we had in the previous example, we'll add the new element `'ducati'` to the end of the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❶ motorcycles.append('ducati')
  print(motorcycles)
```

The `append()` method at ❶ adds `'ducati'` to the end of the list without affecting any of the other elements in the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

The `append()` method makes it easy to build lists dynamically. For example, you can start with an empty list and then add items to the list using a series of `append()` statements. Using an empty list, let's add the elements `'honda'`, `'yamaha'`, and `'suzuki'` to the list:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

The resulting list looks exactly the same as the lists in the previous examples:

```
['honda', 'yamaha', 'suzuki']
```

Building lists this way is very common, because you often won't know the data your users want to store in a program until after the program is running. To put your users in control, start by defining an empty list that will hold the users' values. Then append each new value provided to the list you just created.

Inserting Elements into a List

You can add a new element at any position in your list by using the `insert()` method. You do this by specifying the index of the new element and the value of the new item.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ motorcycles.insert(0, 'ducati')
  print(motorcycles)
```

In this example, the code at ❶ inserts the value `'ducati'` at the beginning of the list. The `insert()` method opens a space at position `0` and stores the value `'ducati'` at that location. This operation shifts every other value in the list one position to the right:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Removing Elements from a List

Often, you'll want to remove an item or a set of items from a list. For example, when a player shoots down an alien from the sky, you'll most likely want to remove it from the list of active aliens. Or when a user

decides to cancel their account on a web application you created, you'll want to remove that user from the list of active users. You can remove an item according to its position in the list or according to its value.

Removing an Item Using the del Statement

If you know the position of the item you want to remove from a list, you can use the `del` statement.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❶ del motorcycles[0]
print(motorcycles)
```

The code at ❶ uses `del` to remove the first item, `'honda'`, from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

You can remove an item from any position in a list using the `del` statement if you know its index. For example, here's how to remove the second item, `'yamaha'`, in the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

The second motorcycle is deleted from the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

In both examples, you can no longer access the value that was removed from the list after the `del` statement is used.

Removing an Item Using the pop() Method

Sometimes you'll want to use the value of an item after you remove it from a list. For example, you might want to get the *x* and *y* position of an alien that was just shot down, so you can draw an explosion at that position. In a web application, you might want to remove a user from a list of active members and then add that user to a list of inactive members.

The `pop()` method removes the last item in a list, but it lets you work with that item after removing it. The term *pop* comes from thinking of a list as a stack of items and popping one item off the top of the stack. In this analogy, the top of a stack corresponds to the end of a list.

Let's pop a motorcycle from the list of motorcycles:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

We start by defining and printing the list `motorcycles` at ❶. At ❷ we pop a value from the list and store that value in the variable `popped_motorcycle`. We print the list at ❸ to show that a value has been removed from the list. Then we print the popped value at ❹ to prove that we still have access to the value that was removed.

The output shows that the value `'suzuki'` was removed from the end of the list and is now stored in the variable `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

How might this `pop()` method be useful? Imagine that the motorcycles in the list are stored in chronological order according to when we owned them. If this is the case, we can use the `pop()` method to print a statement about the last motorcycle we bought:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print("The last motorcycle I owned was a " + last_owned.title() + ".")
```

The output is a simple sentence about the most recent motorcycle we owned:

```
The last motorcycle I owned was a Suzuki.
```

Popping Items from any Position in a List

You can actually use `pop()` to remove an item in a list at any position by including the index of the item you want to remove in parentheses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ first_owned = motorcycles.pop(0)
❷ print('The first motorcycle I owned was a ' + first_owned.title() + '.')
```

We start by popping the first motorcycle in the list at ❶, and then we print a message about that motorcycle at ❷. The output is a simple sentence describing the first motorcycle I ever owned:

```
The first motorcycle I owned was a Honda.
```

Remember that each time you use `pop()`, the item you work with is no longer stored in the list.

If you're unsure whether to use the `del` statement or the `pop()` method, here's a simple way to decide: when you want to delete an item from a list and not use that item in any way, use the `del` statement; if you want to use an item as you remove it, use the `pop()` method.

Removing an Item by Value

Sometimes you won't know the position of the value you want to remove from a list. If you only know the value of the item you want to remove, you can use the `remove()` method.

For example, let's say we want to remove the value `'ducati'` from the list of motorcycles.

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

❶ motorcycles.remove('ducati')
print(motorcycles)
```

The code at ❶ tells Python to figure out where `'ducati'` appears in the list and remove that element:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

You can also use the `remove()` method to work with a value that's being removed from a list. Let's remove the value `'ducati'` and print a reason for removing it from the list:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
print(motorcycles)
❹ print("\nA " + too_expensive.title() + " is too expensive for me.")
```

After defining the list at ❶, we store the value `'ducati'` in a variable called `too_expensive` ❷. We then use this variable to tell Python which value

to remove from the list at ❸. At ❹ the value 'ducati' has been removed from the list but is still stored in the variable too_expensive, allowing us to print a statement about why we removed 'ducati' from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

NOTE *The remove() method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to determine if all occurrences of the value have been removed. You'll learn how to do this in Chapter 7.*

TRY IT YOURSELF

The following exercises are a bit more complex than those in Chapter 2, but they give you an opportunity to use lists in all of the ways described.

3-4. Guest List: If you could invite anyone, living or deceased, to dinner, who would you invite? Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

3-5. Changing Guest List: You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.

- Start with your program from Exercise 3-4. Add a print statement at the end of your program stating the name of the guest who can't make it.
- Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
- Print a second set of invitation messages, one for each person who is still in your list.

3-6. More Guests: You just found a bigger dinner table, so now more space is available. Think of three more guests to invite to dinner.

- Start with your program from Exercise 3-4 or Exercise 3-5. Add a print statement to the end of your program informing people that you found a bigger dinner table.
- Use insert() to add one new guest to the beginning of your list.
- Use insert() to add one new guest to the middle of your list.
- Use append() to add one new guest to the end of your list.
- Print a new set of invitation messages, one for each person in your list.

3-7. Shrinking Guest List: You just found out that your new dinner table won't arrive in time for the dinner, and you have space for only two guests.

- Start with your program from Exercise 3-6. Add a new line that prints a message saying that you can invite only two people for dinner.
- Use `pop()` to remove guests from your list one at a time until only two names remain in your list. Each time you pop a name from your list, print a message to that person letting them know you're sorry you can't invite them to dinner.
- Print a message to each of the two people still on your list, letting them know they're still invited.
- Use `del` to remove the last two names from your list, so you have an empty list. Print your list to make sure you actually have an empty list at the end of your program.

Organizing a List

Often, your lists will be created in an unpredictable order, because you can't always control the order in which your users provide their data. Although this is unavoidable in most circumstances, you'll frequently want to present your information in a particular order. Sometimes you'll want to preserve the original order of your list, and other times you'll want to change the original order. Python provides a number of different ways to organize your lists, depending on the situation.

Sorting a List Permanently with the sort() Method

Python's `sort()` method makes it relatively easy to sort a list. Imagine we have a list of cars and want to change the order of the list to store them alphabetically. To keep the task simple, let's assume that all the values in the list are lowercase.

cars.py

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
❶ cars.sort()
print(cars)
```

The `sort()` method, shown at ❶, changes the order of the list permanently. The cars are now in alphabetical order, and we can never revert to the original order:

```
['audi', 'bmw', 'subaru', 'toyota']
```

You can also sort this list in reverse alphabetical order by passing the argument `reverse=True` to the `sort()` method. The following example sorts the list of cars in reverse alphabetical order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

Again, the order of the list is permanently changed:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Sorting a List Temporarily with the sorted() Function

To maintain the original order of a list but present it in a sorted order, you can use the `sorted()` function. The `sorted()` function lets you display your list in a particular order but doesn't affect the actual order of the list.

Let's try this function on the list of cars.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

❶ print("Here is the original list:")
  print(cars)

❷ print("\nHere is the sorted list:")
  print(sorted(cars))

❸ print("\nHere is the original list again:")
  print(cars)
```

We first print the list in its original order at ❶ and then in alphabetical order at ❷. After the list is displayed in the new order, we show that the list is still stored in its original order at ❸.

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']

❹ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']
```

Notice that the list still exists in its original order at ❹ after the `sorted()` function has been used. The `sorted()` function can also accept a `reverse=True` argument if you want to display a list in reverse alphabetical order.

NOTE *Sorting a list alphabetically is a bit more complicated when all the values are not in lowercase. There are several ways to interpret capital letters when you're deciding on a sort order, and specifying the exact order can be more complex than we want to deal with at this time. However, most approaches to sorting will build directly on what you learned in this section.*

Printing a List in Reverse Order

To reverse the original order of a list, you can use the `reverse()` method. If we originally stored the list of cars in chronological order according to when we owned them, we could easily rearrange the list into reverse chronological order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

Notice that `reverse()` doesn't sort backward alphabetically; it simply reverses the order of the list:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

The `reverse()` method changes the order of a list permanently, but you can revert to the original order anytime by applying `reverse()` to the same list a second time.

Finding the Length of a List

You can quickly find the length of a list by using the `len()` function. The list in this example has four items, so its length is `4`:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

You'll find `len()` useful when you need to identify the number of aliens that still need to be shot down in a game, determine the amount of data you have to manage in a visualization, or figure out the number of registered users on a website, among other tasks.

NOTE *Python counts the items in a list starting with one, so you shouldn't run into any off-by-one errors when determining the length of a list.*

TRY IT YOURSELF

3-8. Seeing the World: Think of at least five places in the world you'd like to visit.

- Store the locations in a list. Make sure the list is not in alphabetical order.
- Print your list in its original order. Don't worry about printing the list neatly, just print it as a raw Python list.
- Use `sorted()` to print your list in alphabetical order without modifying the actual list.
- Show that your list is still in its original order by printing it.
- Use `sorted()` to print your list in reverse alphabetical order without changing the order of the original list.
- Show that your list is still in its original order by printing it again.
- Use `reverse()` to change the order of your list. Print the list to show that its order has changed.
- Use `reverse()` to change the order of your list again. Print the list to show it's back to its original order.
- Use `sort()` to change your list so it's stored in alphabetical order. Print the list to show that its order has been changed.
- Use `sort()` to change your list so it's stored in reverse alphabetical order. Print the list to show that its order has changed.

3-9. Dinner Guests: Working with one of the programs from Exercises 3-4 through 3-7 (page 46), use `len()` to print a message indicating the number of people you are inviting to dinner.

3-10. Every Function: Think of something you could store in a list. For example, you could make a list of mountains, rivers, countries, cities, languages, or anything else you'd like. Write a program that creates a list containing these items and then uses each function introduced in this chapter at least once.

Avoiding Index Errors When Working with Lists

One type of error is common to see when you're working with lists for the first time. Let's say you have a list with three items, and you ask for the fourth item:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

This example results in an *index error*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[3])
IndexError: list index out of range
```

Python attempts to give you the item at index 3. But when it searches the list, no item in `motorcycles` has an index of 3. Because of the off-by-one nature of indexing in lists, this error is typical. People think the third item is item number 3, because they start counting at 1. But in Python the third item is number 2, because it starts indexing at 0.

An index error means Python can't figure out the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one. Then run the program again to see if the results are correct.

Keep in mind that whenever you want to access the last item in a list you use the index `-1`. This will always work, even if your list has changed size since the last time you accessed it:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

The index `-1` always returns the last item in a list, in this case the value `'suzuki'`:

```
'suzuki'
```

The only time this approach will cause an error is when you request the last item from an empty list:

```
motorcycles = []
print(motorcycles[-1])
```

No items are in `motorcycles`, so Python returns another index error:

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

NOTE *If an index error occurs and you can't figure out how to resolve it, try printing your list or just printing the length of your list. Your list might look much different than you thought it did, especially if it has been managed dynamically by your program. Seeing the actual list, or the exact number of items in your list, can help you sort out such logical errors.*

TRY IT YOURSELF

3-11. Intentional Error: If you haven't received an index error in one of your programs yet, try to make one happen. Change an index in one of your programs to produce an index error. Make sure you correct the error before closing the program.

Summary

In this chapter you learned what lists are and how to work with the individual items in a list. You learned how to define a list and how to add and remove elements. You learned to sort lists permanently and temporarily for display purposes. You also learned how to find the length of a list and how to avoid index errors when you're working with lists.

In Chapter 4 you'll learn how to work with items in a list more efficiently. By looping through each item in a list using just a few lines of code you'll be able to work efficiently, even when your list contains thousands or millions of items.

3
WORKING WITH LISTS

In Chapter 3 you learned how to make a simple list, and you learned to work with the individual elements in a list. In this chapter you'll learn how to *loop* through an entire list using just a few lines of code regardless of how long the list is. Looping allows you to take the same action, or set of actions, with every item in a list. As a result, you'll be able to work efficiently with lists of any length, including those with thousands or even millions of items.

Looping Through an Entire List

You'll often want to run through all entries in a list, performing the same task with each item. For example, in a game you might want to move every element on the screen by the same amount, or in a list of numbers you might want to perform the same statistical operation on every element. Or perhaps you'll want to display each headline from a list of articles on a website. When you want to do the same action with every item in a list, you can use Python's `for` loop.

Let's say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. A `for` loop avoids both of these issues by letting Python manage these issues internally.

Let's use a `for` loop to print out each name in a list of magicians:

magicians.py

```
❶ magicians = ['alice', 'david', 'carolina']
❷ for magician in magicians:
❸     print(magician)
```

We begin by defining a list at ❶, just as we did in Chapter 3. At ❷, we define a `for` loop. This line tells Python to pull a name from the list `magicians`, and store it in the variable `magician`. At ❸ we tell Python to print the name that was just stored in `magician`. Python then repeats lines ❷ and ❸, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

A Closer Look at Looping

The concept of looping is important because it's one of the most common ways a computer automates repetitive tasks. For example, in a simple loop like we used in *magicians.py*, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list `magicians` and store it in the variable `magician`. This first value is `'alice'`. Python then reads the next line:

```
    print(magician)
```

Python prints the current value of `magician`, which is still `'alice'`. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, `'david'`, and stores that value in `magician`. Python then executes the line:

```
    print(magician)
```

Python prints the current value of `magician` again, which is now `'david'`. Python repeats the entire loop once more with the last value in the list, `'carolina'`. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the `for` loop, so the program simply ends.

When you're using loops for the first time, keep in mind that the set of steps is repeated once for each item in the list, no matter how many items are in the list. If you have a million items in your list, Python repeats these steps a million times—and usually very quickly.

Also keep in mind when writing your own `for` loops that you can choose any name you want for the temporary variable that holds each value in the list. However, it's helpful to choose a meaningful name that represents a single item from the list. For example, here's a good way to start a `for` loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

These naming conventions can help you follow the action being done on each item within a `for` loop. Using singular and plural names can help you identify whether a section of code is working with a single element from the list or the entire list.

Doing More Work Within a for Loop

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶    print(magician.title() + ", that was a great trick!")
```

The only difference in this code is at ❶ where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of magician is `'alice'`, so Python starts the first message with the name `'Alice'`. The second time through the message will begin with `'David'`, and the third time through the message will begin with `'Carolina'`.

The output shows a personalized message for each magician in the list:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line `for magician in magicians` is considered *inside the loop*, and each indented line is executed once for each

value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
❶   print("I can't wait to see your next trick, " + magician.title() + ".\n")
```

Because we have indented both `print` statements, each line will be executed once for every magician in the list. The newline (`"\n"`) in the second `print` statement ❶ inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice you'll often find it useful to do a number of different operations with each item in a list when you use a `for` loop.

Doing Something After a for Loop

What happens once a `for` loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish.

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the `for` loop without indentation:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() + ".\n")

❶ print("Thank you, everyone. That was a great magic show!")
```

The first two `print` statements are repeated once for each magician in the list, as you saw earlier. However, because the line at ❶ is not indented, it's printed only once:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire data set. For example, you might use a `for` loop to initialize a game by running through a list of characters and displaying each character on the screen. You might then write an unindented block after this loop that displays a Play Now button after all the characters have been drawn to the screen.

Avoiding Indentation Errors

Python uses indentation to determine when one line of code is connected to the line above it. In the previous examples, the lines that printed messages to individual magicians were part of the `for` loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure. In longer Python programs, you'll notice blocks of code indented at a few different levels. These indentation levels help you gain a general sense of the overall program's organization.

As you begin to write code that relies on proper indentation, you'll need to watch for a few common *indentation errors*. For example, people sometimes indent blocks of code that don't need to be indented or forget to indent blocks that need to be indented. Seeing examples of these errors now will help you avoid them in the future and correct them when they do appear in your own programs.

Let's examine some of the more common indentation errors.

Forgetting to Indent

Always indent the line after the `for` statement in a loop. If you forget, Python will remind you:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶ print(magician)
```

The print statement at ❶ should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with.

```
  File "magicians.py", line 3
    print(magician)
        ^
IndentationError: expected an indented block
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the for statement.

Forgetting to Indent Additional Lines

Sometimes your loop will run without any errors but won't produce the expected result. This can happen when you're trying to do several tasks in a loop and you forget to indent some of its lines.

For example, this is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
❶ print("I can't wait to see your next trick, " + magician.title() + ".\n")
```

The print statement at ❶ is supposed to be indented, but because Python finds at least one indented line after the for statement, it doesn't report an error. As a result, the first print statement is executed once for each name in the list because it is indented. The second print statement is not indented, so it is executed only once after the loop has finished running. Because the final value of magician is 'carolina', she is the only one who receives the "looking forward to the next trick" message:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

This is a *logical error*. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

Indenting Unnecessarily

If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

hello_world.py

```
message = "Hello Python world!"
❶    print(message)
```

We don't need to indent the `print` statement at ❶, because it doesn't *belong* to the line above it; hence, Python reports that error:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a `for` loop.

Indenting Unnecessarily After the Loop

If you accidentally indent code that should run after a loop has finished, that code will be repeated once for each item in the list. Sometimes this prompts Python to report an error, but often you'll receive a simple logical error.

For example, let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() + ".\n")

❶    print("Thank you everyone, that was a great magic show!")
```

Because the line at ❶ is indented, it's printed once for each person in the list, as you can see at ❷:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

❷ Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

❷ Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

❷ Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in "Forgetting to Indent Additional Lines" on page 58. Because Python doesn't know what you're trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, determine whether you just need to unindent the code for that action.

Forgetting the Colon

The colon at the end of a `for` statement tells Python to interpret the next line as the start of a loop.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
    print(magician)
```

If you accidentally forget the colon, as shown at ❶, you'll get a syntax error because Python doesn't know what you're trying to do. Although this is an easy error to fix, it's not always an easy error to find. You'd be surprised by the amount of time programmers spend hunting down single-character errors like this. Such errors are difficult to find because we often just see what we expect to see.

TRY IT YOURSELF

4-1. Pizzas: Think of at least three kinds of your favorite pizza. Store these pizza names in a list, and then use a `for` loop to print the name of each pizza.

- Modify your `for` loop to print a sentence using the name of the pizza instead of printing just the name of the pizza. For each pizza you should have one line of output containing a simple statement like *I like pepperoni pizza.*
- Add a line at the end of your program, outside the `for` loop, that states how much you like pizza. The output should consist of three or more lines about the kinds of pizza you like and then an additional sentence, such as *I really love pizza!*

4-2. Animals: Think of at least three different animals that have a common characteristic. Store the names of these animals in a list, and then use a `for` loop to print out the name of each animal.

- Modify your program to print a statement about each animal, such as *A dog would make a great pet.*
- Add a line at the end of your program stating what these animals have in common. You could print a sentence such as *Any of these animals would make a great pet!*

Making Numerical Lists

Many reasons exist to store a set of numbers. For example, you'll need to keep track of the positions of each character in a game, and you might want to keep track of a player's high scores as well. In data visualizations, you'll almost always work with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values, among other types of numerical sets.

Lists are ideal for storing sets of numbers, and Python provides a number of tools to help you work efficiently with lists of numbers. Once you understand how to use these tools effectively, your code will work well even when your lists contain millions of items.

Using the range() Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

numbers.py

```
for value in range(1,5):
    print(value)
```

Although this code looks like it should print the numbers from 1 to 5, it doesn't print the number 5:

```
1
2
3
4
```

In this example, `range()` prints only the numbers 1 through 4. This is another result of the off-by-one behavior you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been 5 in this case.

To print the numbers from 1 to 5, you would use `range(1,6)`:

```
for value in range(1,6):
    print(value)
```

This time the output starts at 1 and ends at 5:

```
1
2
3
4
5
```

If your output is different than what you expect when you're using `range()`, try adjusting your end value by 1.

Using range() to Make a List of Numbers

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers = list(range(1,6))
print(numbers)
```

And this is the result:

```
[1, 2, 3, 4, 5]
```

We can also use the `range()` function to tell Python to skip numbers in a given range. For example, here's how we would list the even numbers between 1 and 10:

even_numbers.py

```
even_numbers = list(range(2,11,2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (**) represent exponents. Here's how you might put the first 10 square numbers into a list:

squares.py

```
❶ squares = []
❷ for value in range(1,11):
❸     square = value**2
❹     squares.append(square)

❺ print(squares)
```

We start with an empty list called `squares` at ❶. At ❷, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop, the current value is raised to the second power and stored in the

variable `square` at ❸. At ❹, each new value of `square` is appended to the list `squares`. Finally, when the loop has finished running, the list of squares is printed at ❺:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable `square` and append each new value directly to the list:

```
squares = []
for value in range(1,11):
❶     squares.append(value**2)

print(squares)
```

The code at ❶ does the same work as the lines at ❸ and ❹ in *squares.py*. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

You can use either of these two approaches when you're making more complex lists. Sometimes using a temporary variable makes your code easier to read; other times it makes the code unnecessarily long. Focus first on writing code that you understand clearly, which does what you want it to do. Then look for more efficient approaches as you review your code.

Simple Statistics with a List of Numbers

A few Python functions are specific to lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

NOTE *The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.*

List Comprehensions

The approach described earlier for generating the list `squares` consisted of using three or four lines of code. A *list comprehension* allows you to generate this same list in just one line of code. A list comprehension combines the `for` loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but I have included them here because you'll most likely see them as soon as you start looking at other people's code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

squares.py

```
squares = [value**2 for value in range(1,11)]
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as `squares`. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is `for value in range(1,11)`, which feeds the values 1 through 10 into the expression `value**2`. Notice that no colon is used at the end of the `for` statement.

The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It takes practice to write your own list comprehensions, but you'll find them worthwhile once you become comfortable creating ordinary lists. When you're writing three or four lines of code to generate lists and it begins to feel repetitive, consider writing your own list comprehensions.

TRY IT YOURSELF

4-3. Counting to Twenty: Use a `for` loop to print the numbers from 1 to 20, inclusive.

4-4. One Million: Make a list of the numbers from one to one million, and then use a `for` loop to print the numbers. (If the output is taking too long, stop it by pressing CTRL-C or by closing the output window.)

4-5. Summing a Million: Make a list of the numbers from one to one million, and then use `min()` and `max()` to make sure your list actually starts at one and ends at one million. Also, use the `sum()` function to see how quickly Python can add a million numbers.

4-6. Odd Numbers: Use the third argument of the `range()` function to make a list of the odd numbers from 1 to 20. Use a `for` loop to print each number.

4-7. Threes: Make a list of the multiples of 3 from 3 to 30. Use a `for` loop to print the numbers in your list.

4-8. Cubes: A number raised to the third power is called a *cube*. For example, the cube of 2 is written as `2**3` in Python. Make a list of the first 10 cubes (that is, the cube of each integer from 1 through 10), and use a `for` loop to print out the value of each cube.

4-9. Cube Comprehension: Use a list comprehension to generate a list of the first 10 cubes.

Working with Part of a List

In Chapter 3 you learned how to access single elements in a list, and in this chapter you've been learning how to work through all the elements in a list. You can also work with a specific group of items in a list, which Python calls a *slice*.

Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices `0` through `3`, which would return elements `0`, `1`, and `2`.

The following example involves a list of players on a team:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
❶ print(players[0:3])
```

The code at ❶ prints a slice of this list, which includes just the first three players. The output retains the structure of the list and includes the first three players in the list:

```
['charles', 'martina', 'michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index `1` and end at index `4`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

This time the slice starts with `'martina'` and ends with `'florence'`:

```
['martina', 'michael', 'florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles', 'martina', 'michael', 'florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index `2` and omit the second index:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael', 'florence', 'eli']
```

This syntax allows you to output all of the elements from any point in your list to the end regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

This prints the names of the last three players and would continue to work as the list of players changes in size.

Looping Through a Slice

You can use a slice in a `for` loop if you want to loop through a subset of the elements in a list. In the next example we loop through the first three players and print their names as part of a simple roster:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Instead of looping through the entire list of players at ❶, Python loops through only the first three names:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Slices are very useful in a number of situations. For instance, when you're creating a game, you could add a player's final score to a list every time that player finishes playing. You could then get a player's top three scores by sorting the list in decreasing order and taking a slice that includes just the first three scores. When you're working with data, you can use slices to process

your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

Copying a List

Often, you'll want to start with an existing list and make an entirely new list based on the first one. Let's explore how copying a list works and examine one situation in which copying a list is useful.

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

For example, imagine we have a list of our favorite foods and want to make a separate list of foods that a friend likes. This friend likes everything in our list so far, so we can create their list by copying ours:

foods.py

```
❶ my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

At ❶ we make a list of the foods we like called `my_foods`. At ❷ we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices and store the copy in `friend_foods`. When we print each list, we see that they both contain the same foods:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favorite foods:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)
```

```
print("\nMy friend's favorite foods are:")
print(friend_foods)
```

At ❶ we copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example. Next, we add a new food to each list: at ❷ we add `'cannoli'` to `my_foods`, and at ❸ we add `'ice cream'` to `friend_foods`. We then print the two lists to see whether each of these foods is in the appropriate list.

```
My favorite foods are:
❹ ['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:
❺ ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

The output at ❹ shows that `'cannoli'` now appears in our list of favorite foods but `'ice cream'` doesn't. At ❺ we can see that `'ice cream'` now appears in our friend's list but `'cannoli'` doesn't. If we had simply set `friend_foods` equal to `my_foods`, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```
my_foods = ['pizza', 'falafel', 'carrot cake']

# This doesn't work:
❶ friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Instead of storing a copy of `my_foods` in `friend_foods` at ❶, we set `friend_foods` equal to `my_foods`. This syntax actually tells Python to connect the new variable `friend_foods` to the list that is already contained in `my_foods`, so now both variables point to the same list. As a result, when we add `'cannoli'` to `my_foods`, it will also appear in `friend_foods`. Likewise `'ice cream'` will appear in both lists, even though it appears to be added only to `friend_foods`.

The output shows that both lists are the same now, which is not what we wanted:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

NOTE *Don't worry about the details in this example for now. Basically, if you're trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice, as we did in the first example.*

TRY IT YOURSELF

4-10. Slices: Using one of the programs you wrote in this chapter, add several lines to the end of the program that do the following:

- Print the message, *The first three items in the list are:*. Then use a slice to print the first three items from that program's list.
- Print the message, *Three items from the middle of the list are:*. Use a slice to print three items from the middle of the list.
- Print the message, *The last three items in the list are:*. Use a slice to print the last three items in the list.

4-11. My Pizzas, Your Pizzas: Start with your program from Exercise 4-1 (page 60). Make a copy of the list of pizzas, and call it `friend_pizzas`. Then, do the following:

- Add a new pizza to the original list.
- Add a different pizza to the list `friend_pizzas`.
- Prove that you have two separate lists. Print the message, *My favorite pizzas are:*, and then use a `for` loop to print the first list. Print the message, *My friend's favorite pizzas are:*, and then use a `for` loop to print the second list. Make sure each new pizza is stored in the appropriate list.

4-12. More Loops: All versions of *foods.py* in this section have avoided using `for` loops when printing to save space. Choose a version of *foods.py*, and write two `for` loops to print each list of foods.

Tuples

Lists work well for storing sets of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*.

Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

dimensions.py

```
❶ dimensions = (200, 50)
❷ print(dimensions[0])
  print(dimensions[1])
```

We define the tuple `dimensions` at ❶, using parentheses instead of square brackets. At ❷ we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
200
50
```

Let's see what happens if we try to change one of the items in the tuple `dimensions`:

```
dimensions = (200, 50)
❶ dimensions[0] = 250
```

The code at ❶ tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
Traceback (most recent call last):
  File "dimensions.py", line 3, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

Looping Through All Values in a Tuple

You can loop over all the values in a tuple using a `for` loop, just as you did with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Writing over a Tuple

Although you can't modify a tuple, you can assign a new value to a variable that holds a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
❶ dimensions = (200, 50)
  print("Original dimensions:")
  for dimension in dimensions:
      print(dimension)

❷ dimensions = (400, 100)
❸ print("\nModified dimensions:")
  for dimension in dimensions:
      print(dimension)
```

The block at ❶ defines the original tuple and prints the initial dimensions. At ❷, we store a new tuple in the variable `dimensions`. We then print the new dimensions at ❸. Python doesn't raise any errors this time, because overwriting a variable is valid:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

TRY IT YOURSELF

4-13. Buffet: A buffet-style restaurant offers only five basic foods. Think of five simple foods, and store them in a tuple.

- Use a `for` loop to print each food the restaurant offers.
- Try to modify one of the items, and make sure that Python rejects the change.
- The restaurant changes its menu, replacing two of the items with different foods. Add a block of code that rewrites the tuple, and then use a `for` loop to print each of the items on the revised menu.

Styling Your Code

Now that you're writing longer programs, ideas about how to style your code are worthwhile to know. Take the time to make your code as easy as possible to read. Writing easy-to-read code helps you keep track of what your programs are doing and helps others understand your code as well.

Python programmers have agreed on a number of styling conventions to ensure that everyone's code is structured in roughly the same way. Once you've learned to write clean Python code, you should be able to understand the overall structure of anyone else's Python code, as long as they follow the same guidelines. If you're hoping to become a professional programmer at some point, you should begin following these guidelines as soon as possible to develop good habits.

The Style Guide

When someone wants to make a change to the Python language, they write a *Python Enhancement Proposal (PEP)*. One of the oldest PEPs is *PEP 8*, which instructs Python programmers on how to style their code. PEP 8 is fairly lengthy, but much of it relates to more complex coding structures than what you've seen so far.

The Python style guide was written with the understanding that code is read more often than it is written. You'll write your code once and then start reading it as you begin debugging. When you add features to a program, you'll spend more time reading your code. When you share your code with other programmers, they'll read your code as well.

Given the choice between writing code that's easier to write or code that's easier to read, Python programmers will almost always encourage you to write code that's easier to read. The following guidelines will help you write clear code from the start.

Indentation

PEP 8 recommends that you use four spaces per indentation level. Using four spaces improves readability while leaving room for multiple levels of indentation on each line.

In a word processing document, people often use tabs rather than spaces to indent. This works well for word processing documents, but the Python interpreter gets confused when tabs are mixed with spaces. Every text editor provides a setting that lets you use the TAB key but then converts each tab to a set number of spaces. You should definitely use your TAB key, but also make sure your editor is set to insert spaces rather than tabs into your document.

Mixing tabs and spaces in your file can cause problems that are very difficult to diagnose. If you think you have a mix of tabs and spaces, you can convert all tabs in a file to spaces in most editors.

Line Length

Many Python programmers recommend that each line should be less than 80 characters. Historically, this guideline developed because most computers could fit only 79 characters on a single line in a terminal window. Currently, people can fit much longer lines on their screens, but other reasons exist to adhere to the 79-character standard line length. Professional programmers often have several files open on the same screen, and using the standard line length allows them to see entire lines in two or three files that are open side by side onscreen. PEP 8 also recommends that you limit all of your comments to 72 characters per line, because some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line.

The PEP 8 guidelines for line length are not set in stone, and some teams prefer a 99-character limit. Don't worry too much about line length in your code as you're learning, but be aware that people who are working collaboratively almost always follow the PEP 8 guidelines. Most editors allow you to set up a visual cue, usually a vertical line on your screen, that shows you where these limits are.

NOTE *Appendix B shows you how to configure your text editor so it always inserts four spaces each time you press the TAB key and shows a vertical guideline to help you follow the 79-character limit.*

Blank Lines

To group parts of your program visually, use blank lines. You should use blank lines to organize your files, but don't do so excessively. By following the examples provided in this book, you should strike the right balance. For example, if you have five lines of code that build a list, and then another three lines that do something with that list, it's appropriate to place a blank line between the two sections. However, you should not place three or four blank lines between the two sections.

Blank lines won't affect how your code runs, but they will affect the readability of your code. The Python interpreter uses horizontal indentation to interpret the meaning of your code, but it disregards vertical spacing.

Other Style Guidelines

PEP 8 has many additional styling recommendations, but most of the guidelines refer to more complex programs than what you're writing at this point. As you learn more complex Python structures, I'll share the relevant parts of the PEP 8 guidelines.

TRY IT YOURSELF

4-14. PEP 8: Look through the original PEP 8 style guide at *https://python.org/dev/peps/pep-0008/*. You won't use much of it now, but it might be interesting to skim through it.

4-15. Code Review: Choose three of the programs you've written in this chapter and modify each one to comply with PEP 8:

- Use four spaces for each indentation level. Set your text editor to insert four spaces every time you press TAB, if you haven't already done so (see Appendix B for instructions on how to do this).
- Use less than 80 characters on each line, and set your editor to show a vertical guideline at the 80th character position.
- Don't use blank lines excessively in your program files.

Summary

In this chapter you learned how to work efficiently with the elements in a list. You learned how to work through a list using a `for` loop, how Python uses indentation to structure a program, and how to avoid some common indentation errors. You learned to make simple numerical lists, as well as a few operations you can perform on numerical lists. You learned how to slice a list to work with a subset of items and how to copy lists properly using a slice. You also learned about tuples, which provide a degree of protection to a set of values that shouldn't change, and how to style your increasingly complex code to make it easy to read.

In Chapter 5, you'll learn to respond appropriately to different conditions by using `if` statements. You'll learn to string together relatively complex sets of conditional tests to respond appropriately to exactly the kind of situation or information you're looking for. You'll also learn to use `if` statements while looping through a list to take specific actions with selected elements from a list.

4
IF STATEMENTS

Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's `if` statement allows you to examine the current state of a program and respond appropriately to that state.

In this chapter you'll learn to write conditional tests, which allow you to check any condition of interest. You'll learn to write simple `if` statements, and you'll learn how to create a more complex series of `if` statements to identify when the exact conditions you want are present. You'll then apply this concept to lists, so you'll be able to write a `for` loop that handles most items in a list one way but handles certain items with specific values in a different way.

A Simple Example

The following short example shows how `if` tests let you respond to special situations correctly. Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value `'bmw'` should be printed in all uppercase. The following code loops through a list of car names and looks for the value `'bmw'`. Whenever the value is `'bmw'`, it's printed in uppercase instead of title case:

cars.py

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

❶

The loop in this example first checks if the current value of `car` is `'bmw'` ❶. If it is, the value is printed in uppercase. If the value of `car` is anything other than `'bmw'`, it's printed in title case:

```
Audi
BMW
Subaru
Toyota
```

This example combines a number of the concepts you'll learn about in this chapter. Let's begin by looking at the kinds of tests you can use to examine the conditions in your program.

Conditional Tests

At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a *conditional test*. Python uses the values `True` and `False` to decide whether the code in an `if` statement should be executed. If a conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Checking for Equality

Most conditional tests compare the current value of a variable to a specific value of interest. The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
❶ >>> car = 'bmw'
❷ >>> car == 'bmw'
True
```

The line at ❶ sets the value of `car` to `'bmw'` using a single equal sign, as you've seen many times already. The line at ❷ checks whether the value of `car` is `'bmw'` using a double equal sign (`==`). This *equality operator* returns `True` if the values on the left and right side of the operator match, and `False` if they don't match. The values in this example match, so Python returns `True`.

When the value of `car` is anything other than `'bmw'`, this test returns `False`:

```
❶ >>> car = 'audi'
❷ >>> car == 'bmw'
False
```

A single equal sign is really a statement; you might read the code at ❶ as "Set the value of car equal to `'audi'`." On the other hand, a double equal sign, like the one at ❷, asks a question: "Is the value of car equal to `'bmw'`?" Most programming languages use equal signs in this way.

Ignoring Case When Checking for Equality

Testing for equality is case sensitive in Python. For example, two values with different capitalization are not considered equal:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

If case matters, this behavior is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

This test would return `True` no matter how the value `'Audi'` is formatted because the test is now case insensitive. The `lower()` function doesn't change the value that was originally stored in `car`, so you can do this kind of comparison without affecting the original variable:

```
❶ >>> car = 'Audi'
❷ >>> car.lower() == 'audi'
True
❸ >>> car
'Audi'
```

At ❶ we store the capitalized string `'Audi'` in the variable `car`. At ❷ we convert the value of `car` to lowercase and compare the lowercase value

to the string 'audi'. The two strings match, so Python returns True. At ❸ we can see that the value stored in car has not been affected by the conditional test.

Websites enforce certain rules for the data that users enter in a manner similar to this. For example, a site might use a conditional test like this to ensure that every user has a truly unique username, not just a variation on the capitalization of another person's username. When someone submits a new username, that new username is converted to lowercase and compared to the lowercase versions of all existing usernames. During this check, a username like 'John' will be rejected if any variation of 'john' is already in use.

Checking for Inequality

When you want to determine whether two values are not equal, you can combine an exclamation point and an equal sign (!=). The exclamation point represents *not*, as it does in many programming languages.

Let's use another if statement to examine how to use the inequality operator. We'll store a requested pizza topping in a variable and then print a message if the person did not order anchovies:

toppings.py

```
requested_topping = 'mushrooms'

❶ if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

The line at ❶ compares the value of requested_topping to the value 'anchovies'. If these two values do not match, Python returns True and executes the code following the if statement. If the two values match, Python returns False and does not run the code following the if statement.

Because the value of requested_topping is not 'anchovies', the print statement is executed:

```
Hold the anchovies!
```

Most of the conditional expressions you write will test for equality, but sometimes you'll find it more efficient to test for inequality.

Numerical Comparisons

Testing numerical values is pretty straightforward. For example, the following code checks whether a person is 18 years old:

```
>>> age = 18
>>> age == 18
True
```

You can also test to see if two numbers are not equal. For example, the following code prints a message if the given answer is not correct:

magic_number.py

```
answer = 17

❶ if answer != 42:
    print("That is not the correct answer. Please try again!")
```

The conditional test at ❶ passes, because the value of `answer` (`17`) is not equal to `42`. Because the test passes, the indented code block is executed:

```
That is not the correct answer. Please try again!
```

You can include various mathematical comparisons in your conditional statements as well, such as less than, less than or equal to, greater than, and greater than or equal to:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Each mathematical comparison can be used as part of an `if` statement, which can help you detect the exact conditions of interest.

Checking Multiple Conditions

You may want to check multiple conditions at the same time. For example, sometimes you might need two conditions to be `True` to take an action. Other times you might be satisfied with just one condition being `True`. The keywords `and` and `or` can help you in these situations.

Using and to Check Multiple Conditions

To check whether two conditions are both `True` simultaneously, use the keyword `and` to combine the two conditional tests; if each test passes, the overall expression evaluates to `True`. If either test fails or if both tests fail, the expression evaluates to `False`.

For example, you can check whether two people are both over 21 using the following test:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 and age_1 >= 21
  False
```

```
❸ >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

At ❶ we define two ages, `age_0` and `age_1`. At ❷ we check whether both ages are 21 or older. The test on the left passes, but the test on the right fails, so the overall conditional expression evaluates to `False`. At ❸ we change `age_1` to 22. The value of `age_1` is now greater than 21, so both individual tests pass, causing the overall conditional expression to evaluate as `True`.

To improve readability, you can use parentheses around the individual tests, but they are not required. If you use parentheses, your test would look like this:

```
(age_0 >= 21) and (age_1 >= 21)
```

Using or to Check Multiple Conditions

The keyword `or` allows you to check multiple conditions as well, but it passes when either or both of the individual tests pass. An `or` expression fails only when both individual tests fail.

Let's consider two ages again, but this time we'll look for only one person to be over 21:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
  True
❸ >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

We start with two age variables again at ❶. Because the test for `age_0` at ❷ passes, the overall expression evaluates to `True`. We then lower `age_0` to 18. In the test at ❸, both tests now fail and the overall expression evaluates to `False`.

Checking Whether a Value Is in a List

Sometimes it's important to check whether a list contains a certain value before taking an action. For example, you might want to check whether a new username already exists in a list of current usernames before completing someone's registration on a website. In a mapping project, you might want to check whether a submitted location already exists in a list of known locations.

To find out whether a particular value is already in a list, use the keyword `in`. Let's consider some code you might write for a pizzeria. We'll make a list of toppings a customer has requested for a pizza and then check whether certain toppings are in the list.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
❶ >>> 'mushrooms' in requested_toppings
True
❷ >>> 'pepperoni' in requested_toppings
False
```

At ❶ and ❷, the keyword `in` tells Python to check for the existence of `'mushrooms'` and `'pepperoni'` in the list `requested_toppings`. This technique is quite powerful because you can create a list of essential values, and then easily check whether the value you're testing matches one of the values in the list.

Checking Whether a Value Is Not in a List

Other times, it's important to know if a value does not appear in a list. You can use the keyword `not` in this situation. For example, consider a list of users who are banned from commenting in a forum. You can check whether a user has been banned before allowing that person to submit a comment:

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'

❶ if user not in banned_users:
    print(user.title() + ", you can post a response if you wish.")
```

The line at ❶ reads quite clearly. If the value of `user` is not in the list `banned_users`, Python returns `True` and executes the indented line.

The user `'marie'` is not in the list `banned_users`, so she sees a message inviting her to post a response:

```
Marie, you can post a response if you wish.
```

Boolean Expressions

As you learn more about programming, you'll hear the term *Boolean expression* at some point. A Boolean expression is just another name for a conditional test. A *Boolean value* is either `True` or `False`, just like the value of a conditional expression after it has been evaluated.

Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website:

```
game_active = True
can_edit = False
```

Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

TRY IT YOURSELF

5-1. Conditional Tests: Write a series of conditional tests. Print a statement describing each test and your prediction for the results of each test. Your code should look something like this:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Look closely at your results, and make sure you understand why each line evaluates to `True` or `False`.
- Create at least 10 tests. Have at least 5 tests evaluate to `True` and another 5 tests evaluate to `False`.

5-2. More Conditional Tests: You don't have to limit the number of tests you create to 10. If you want to try more comparisons, write more tests and add them to *conditional_tests.py*. Have at least one `True` and one `False` result for each of the following:

- Tests for equality and inequality with strings
- Tests using the `lower()` function
- Numerical tests involving equality and inequality, greater than and less than, greater than or equal to, and less than or equal to
- Tests using the `and` keyword and the `or` keyword
- Test whether an item is in a list
- Test whether an item is not in a list

if Statements

When you understand conditional tests, you can start writing `if` statements. Several different kinds of `if` statements exist, and your choice of which to use depends on the number of conditions you need to test. You saw several examples of `if` statements in the discussion about conditional tests, but now let's dig deeper into the topic.

Simple if Statements

The simplest kind of `if` statement has one test and one action:

```
if conditional_test:
    do something
```

You can put any conditional test in the first line and just about any action in the indented block following the test. If the conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Let's say we have a variable representing a person's age, and we want to know if that person is old enough to vote. The following code tests whether the person can vote:

voting.py

```
age = 19
❶ if age >= 18:
❷     print("You are old enough to vote!")
```

At ❶ Python checks to see whether the value in `age` is greater than or equal to 18. It is, so Python executes the indented `print` statement at ❷:

```
You are old enough to vote!
```

Indentation plays the same role in `if` statements as it did in `for` loops. All indented lines after an `if` statement will be executed if the test passes, and the entire block of indented lines will be ignored if the test does not pass.

You can have as many lines of code as you want in the block following the `if` statement. Let's add another line of output if the person is old enough to vote, asking if the individual has registered to vote yet:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

The conditional test passes, and both `print` statements are indented, so both lines are printed:

```
You are old enough to vote!
Have you registered to vote yet?
```

If the value of `age` is less than 18, this program would produce no output.

if-else Statements

Often, you'll want to take one action when a conditional test passes and a different action in all other cases. Python's `if-else` syntax makes this possible. An `if-else` block is similar to a simple `if` statement, but the `else` statement allows you to define an action or set of actions that are executed when the conditional test fails.

We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

If the conditional test at ❶ passes, the first block of indented `print` statements is executed. If the test evaluates to `False`, the `else` block at ❷ is executed. Because `age` is less than 18 this time, the conditional test fails and the code in the `else` block is executed:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The `if-else` structure works well in situations in which you want Python to always execute one of two possible actions. In a simple `if-else` chain like this, one of the two actions will always be executed.

The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if-elif-else` syntax. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order until one passes. When a test passes, the code following that test is executed and Python skips the rest of the tests.

Many real-world situations involve more than two possible conditions. For example, consider an amusement park that charges different rates for different age groups:

- Admission for anyone under age 4 is free.
- Admission for anyone between the ages of 4 and 18 is $5.
- Admission for anyone age 18 or older is $10.

How can we use an `if` statement to determine a person's admission rate? The following code tests for the age group of a person and then prints an admission price message:

amusement_park.py

```
age = 12

❶ if age < 4:
    print("Your admission cost is $0.")
```

```
❷ elif age < 18:
      print("Your admission cost is $5.")
❸ else:
      print("Your admission cost is $10.")
```

The `if` test at ❶ tests whether a person is under 4 years old. If the test passes, an appropriate message is printed and Python skips the rest of the tests. The `elif` line at ❷ is really another `if` test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is less than 18, an appropriate message is printed and Python skips the `else` block. If both the `if` and `elif` tests fail, Python runs the code in the `else` block at ❸.

In this example the test at ❶ evaluates to `False`, so its code block is not executed. However, the second test evaluates to `True` (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

```
Your admission cost is $5.
```

Any age greater than 17 would cause the first two tests to fail. In these situations, the `else` block would be executed and the admission price would be $10.

Rather than printing the admission price within the `if-elif-else` block, it would be more concise to set just the price inside the `if-elif-else` chain and then have a simple `print` statement that runs after the chain has been evaluated:

```
age = 12

if age < 4:
❶     price = 0
elif age < 18:
❷     price = 5
else:
❸     price = 10

❹ print("Your admission cost is $" + str(price) + ".")
```

The lines at ❶, ❷, and ❸ set the value of `price` according to the person's age, as in the previous example. After the price is set by the `if-elif-else` chain, a separate unindented `print` statement ❹ uses this value to display a message reporting the person's admission price.

This code produces the same output as the previous example, but the purpose of the `if-elif-else` chain is narrower. Instead of determining a price and displaying a message, it simply determines the admission price. In addition to being more efficient, this revised code is easier to modify than the original approach. To change the text of the output message, you would need to change only one `print` statement rather than three separate `print` statements.

Using Multiple elif Blocks

You can use as many `elif` blocks in your code as you like. For example, if the amusement park were to implement a discount for seniors, you could add one more conditional test to the code to determine whether someone qualified for the senior discount. Let's say that anyone 65 or older pays half the regular admission, or $5:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
❶ elif age < 65:
    price = 10
❷ else:
    price = 5

print("Your admission cost is $" + str(price) + ".")
```

Most of this code is unchanged. The second `elif` block at ❶ now checks to make sure a person is less than age 65 before assigning them the full admission rate of $10. Notice that the value assigned in the `else` block at ❷ needs to be changed to $5, because the only ages that make it to this block are people 65 or older.

Omitting the else Block

Python does not require an `else` block at the end of an `if-elif` chain. Sometimes an `else` block is useful; sometimes it is clearer to use an additional `elif` statement that catches the specific condition of interest:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
elif age < 65:
    price = 10
❶ elif age >= 65:
    price = 5

print("Your admission cost is $" + str(price) + ".")
```

The extra `elif` block at ❶ assigns a price of $5 when the person is 65 or older, which is a bit clearer than the general `else` block. With this change, every block of code must pass a specific test in order to be executed.

The else block is a catchall statement. It matches any condition that wasn't matched by a specific if or elif test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you are testing for, consider using a final elif block and omit the else block. As a result, you'll gain extra confidence that your code will run only under the correct conditions.

Testing Multiple Conditions

The if-elif-else chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests. This behavior is beneficial, because it's efficient and allows you to test for one specific condition.

However, sometimes it's important to check all of the conditions of interest. In this case, you should use a series of simple if statements with no elif or else blocks. This technique makes sense when more than one condition could be True, and you want to act on every condition that is True.

Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

toppings.py

```
❶ requested_toppings = ['mushrooms', 'extra cheese']

❷ if 'mushrooms' in requested_toppings:
      print("Adding mushrooms.")
❸ if 'pepperoni' in requested_toppings:
      print("Adding pepperoni.")
❹ if 'extra cheese' in requested_toppings:
      print("Adding extra cheese.")

  print("\nFinished making your pizza!")
```

We start at ❶ with a list containing the requested toppings. The if statement at ❷ checks to see whether the person requested mushrooms on their pizza. If so, a message is printed confirming that topping. The test for pepperoni at ❸ is another simple if statement, not an elif or else statement, so this test is run regardless of whether the previous test passed or not. The code at ❹ checks whether extra cheese was requested regardless of the results from the first two tests. These three independent tests are executed every time this program is run.

Because every condition in this example is evaluated, both mushrooms and extra cheese are added to the pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

This code would not work properly if we used an `if-elif-else` block, because the code would stop running after only one test passes. Here's what that would look like:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

The test for `'mushrooms'` is the first test to pass, so mushrooms are added to the pizza. However, the values `'extra cheese'` and `'pepperoni'` are never checked, because Python doesn't run any tests beyond the first test that passes in an `if-elif-else` chain. The customer's first topping will be added, but all of their other toppings will be missed:

```
Adding mushrooms.

Finished making your pizza!
```

In summary, if you want only one block of code to run, use an `if-elif-else` chain. If more than one block of code needs to run, use a series of independent `if` statements.

TRY IT YOURSELF

5-3. Alien Colors #1: Imagine an alien was just shot down in a game. Create a variable called `alien_color` and assign it a value of `'green'`, `'yellow'`, or `'red'`.

- Write an `if` statement to test whether the alien's color is green. If it is, print a message that the player just earned 5 points.
- Write one version of this program that passes the `if` test and another that fails. (The version that fails will have no output.)

5-4. Alien Colors #2: Choose a color for an alien as you did in Exercise 5-3, and write an `if-else` chain.

- If the alien's color is green, print a statement that the player just earned 5 points for shooting the alien.
- If the alien's color isn't green, print a statement that the player just earned 10 points.
- Write one version of this program that runs the `if` block and another that runs the `else` block.

5-5. Alien Colors #3: Turn your `if-else` chain from Exercise 5-4 into an `if-elif-else` chain.

- If the alien is green, print a message that the player earned 5 points.
- If the alien is yellow, print a message that the player earned 10 points.
- If the alien is red, print a message that the player earned 15 points.
- Write three versions of this program, making sure each message is printed for the appropriate color alien.

5-6. Stages of Life: Write an `if-elif-else` chain that determines a person's stage of life. Set a value for the variable `age`, and then:

- If the person is less than 2 years old, print a message that the person is a baby.
- If the person is at least 2 years old but less than 4, print a message that the person is a toddler.
- If the person is at least 4 years old but less than 13, print a message that the person is a kid.
- If the person is at least 13 years old but less than 20, print a message that the person is a teenager.
- If the person is at least 20 years old but less than 65, print a message that the person is an adult.
- If the person is age 65 or older, print a message that the person is an elder.

5-7. Favorite Fruit: Make a list of your favorite fruits, and then write a series of independent `if` statements that check for certain fruits in your list.

- Make a list of your three favorite fruits and call it `favorite_fruits`.
- Write five `if` statements. Each should check whether a certain kind of fruit is in your list. If the fruit is in your list, the `if` block should print a statement, such as *You really like bananas!*

Using if Statements with Lists

You can do some interesting work when you combine lists and `if` statements. You can watch for special values that need to be treated differently than other values in the list. You can manage changing conditions efficiently, such as the availability of certain items in a restaurant throughout a shift. You can also begin to prove that your code works as you expect it to in all possible situations.

Checking for Special Items

This chapter began with a simple example that showed how to handle a special value like `'bmw'`, which needed to be printed in a different format than other values in the list. Now that you have a basic understanding of conditional tests and `if` statements, let's take a closer look at how you can watch for special values in a list and handle those values appropriately.

Let's continue with the pizzeria example. The pizzeria displays a message whenever a topping is added to your pizza, as it's being made. The code for this action can be written very efficiently by making a list of toppings the customer has requested and using a loop to announce each topping as it's added to the pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```

The output is straightforward because this code is just a simple `for` loop:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

But what if the pizzeria runs out of green peppers? An `if` statement inside the `for` loop can handle this situation appropriately:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
❶     if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
❷     else:
        print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```

This time we check each requested item before adding it to the pizza. The code at ❶ checks to see if the person requested green peppers. If so, we display a message informing them why they can't have green peppers. The `else` block at ❷ ensures that all other toppings will be added to the pizza.

The output shows that each requested topping is handled appropriately.

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Checking That a List Is Not Empty

We've made a simple assumption about every list we've worked with so far; we've assumed that each list has at least one item in it. Soon we'll let users provide the information that's stored in a list, so we won't be able to assume that a list has any items in it each time a loop is run. In this situation, it's useful to check whether a list is empty before running a `for` loop.

As an example, let's check whether the list of requested toppings is empty before building the pizza. If the list is empty, we'll prompt the user and make sure they want a plain pizza. If the list is not empty, we'll build the pizza just as we did in the previous examples:

```
❶ requested_toppings = []

❷ if requested_toppings:
      for requested_topping in requested_toppings:
          print("Adding " + requested_topping + ".")
      print("\nFinished making your pizza!")
❸ else:
      print("Are you sure you want a plain pizza?")
```

This time we start out with an empty list of requested toppings at ❶. Instead of jumping right into a `for` loop, we do a quick check at ❷. When the name of a list is used in an `if` statement, Python returns `True` if the list contains at least one item; an empty list evaluates to `False`. If `requested_toppings` passes the conditional test, we run the same `for` loop we used in the previous example. If the conditional test fails, we print a message asking the customer if they really want a plain pizza with no toppings ❸.

The list is empty in this case, so the output asks if the user really wants a plain pizza:

```
Are you sure you want a plain pizza?
```

If the list is not empty, the output will show each requested topping being added to the pizza.

Using Multiple Lists

People will ask for just about anything, especially when it comes to pizza toppings. What if a customer actually wants french fries on their pizza? You can use lists and `if` statements to make sure your input makes sense before you act on it.

Let's watch out for unusual topping requests before we build a pizza. The following example defines two lists. The first is a list of available toppings at the pizzeria, and the second is the list of toppings that the user has requested. This time, each item in `requested_toppings` is checked against the list of available toppings before it's added to the pizza:

```
❶ available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings:
❹     if requested_topping in available_toppings:
          print("Adding " + requested_topping + ".")
❺     else:
          print("Sorry, we don't have " + requested_topping + ".")

print("\nFinished making your pizza!")
```

At ❶ we define a list of available toppings at this pizzeria. Note that this could be a tuple if the pizzeria has a stable selection of toppings. At ❷, we make a list of toppings that a customer has requested. Note the unusual request, `'french fries'`. At ❸ we loop through the list of requested toppings. Inside the loop, we first check to see if each requested topping is actually in the list of available toppings ❹. If it is, we add that topping to the pizza. If the requested topping is not in the list of available toppings, the `else` block will run ❺. The `else` block prints a message telling the user which toppings are unavailable.

This code syntax produces clean, informative output:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

In just a few lines of code, we've managed a real-world situation pretty effectively!

TRY IT YOURSELF

5-8. Hello Admin: Make a list of five or more usernames, including the name `'admin'`. Imagine you are writing code that will print a greeting to each user after they log in to a website. Loop through the list, and print a greeting to each user:

- If the username is `'admin'`, print a special greeting, such as *Hello admin, would you like to see a status report?*
- Otherwise, print a generic greeting, such as *Hello Eric, thank you for logging in again.*

5-9. No Users: Add an `if` test to *hello_admin.py* to make sure the list of users is not empty.

- If the list is empty, print the message *We need to find some users!*
- Remove all of the usernames from your list, and make sure the correct message is printed.

5-10. Checking Usernames: Do the following to create a program that simulates how websites ensure that everyone has a unique username.

- Make a list of five or more usernames called `current_users`.
- Make another list of five usernames called `new_users`. Make sure one or two of the new usernames are also in the `current_users` list.
- Loop through the `new_users` list to see if each new username has already been used. If it has, print a message that the person will need to enter a new username. If a username has not been used, print a message saying that the username is available.
- Make sure your comparison is case insensitive. If `'John'` has been used, `'JOHN'` should not be accepted.

5-11. Ordinal Numbers: Ordinal numbers indicate their position in a list, such as *1st* or *2nd*. Most ordinal numbers end in *th*, except 1, 2, and 3.

- Store the numbers 1 through 9 in a list.
- Loop through the list.
- Use an `if-elif-else` chain inside the loop to print the proper ordinal ending for each number. Your output should read `"1st 2nd 3rd 4th 5th 6th 7th 8th 9th"`, and each result should be on a separate line.

Styling Your if Statements

In every example in this chapter, you've seen good styling habits. The only recommendation PEP 8 provides for styling conditional tests is to use a single space around comparison operators, such as ==, >=, <=. For example:

```
if age < 4:
```

is better than:

```
if age<4:
```

Such spacing does not affect the way Python interprets your code; it just makes your code easier for you and others to read.

TRY IT YOURSELF

5-12. Styling `if` statements: Review the programs you wrote in this chapter, and make sure you styled your conditional tests appropriately.

5-13. Your Ideas: At this point, you're a more capable programmer than you were when you started this book. Now that you have a better sense of how real-world situations are modeled in programs, you might be thinking of some problems you could solve with your own programs. Record any new ideas you have about problems you might want to solve as your programming skills continue to improve. Consider games you might want to write, data sets you might want to explore, and web applications you'd like to create.

Summary

In this chapter you learned how to write conditional tests, which always evaluate to `True` or `False`. You learned to write simple `if` statements, `if-else` chains, and `if-elif-else` chains. You began using these structures to identify particular conditions you needed to test and to know when those conditions have been met in your programs. You learned to handle certain items in a list differently than all other items while continuing to utilize the efficiency of a `for` loop. You also revisited Python's style recommendations to ensure that your increasingly complex programs are still relatively easy to read and understand.

In Chapter 6 you'll learn about Python's dictionaries. A dictionary is similar to a list, but it allows you to connect pieces of information. You'll learn to build dictionaries, loop through them, and use them in combination with lists and `if` statements. Learning about dictionaries will enable you to model an even wider variety of real-world situations.

5
DICTIONARIES

In this chapter you'll learn how to use Python's dictionaries, which allow you to connect pieces of related information. You'll learn how to access the information once it's in a dictionary and how to modify that information. Because dictionaries can store an almost limitless amount of information, I'll show you how to loop through the data in a dictionary. Additionally, you'll learn to nest dictionaries inside lists, lists inside dictionaries, and even dictionaries inside other dictionaries.

Understanding dictionaries allows you to model a variety of real-world objects more accurately. You'll be able to create a dictionary representing a person and then store as much information as you want about that person. You can store their name, age, location, profession, and any other aspect of a person you can describe. You'll be able to store any two kinds of information that can be matched up, such as a list of words and their meanings, a list of people's names and their favorite numbers, a list of mountains and their elevations, and so forth.

A Simple Dictionary

Consider a game featuring aliens that can have different colors and point values. This simple dictionary stores information about a particular alien:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's color and point value. The two `print` statements access and display that information, as shown here:

```
green
5
```

As with most new programming concepts, using dictionaries takes practice. Once you've worked with dictionaries for a bit you'll soon see how effectively they can model real-world situations.

Working with Dictionaries

A *dictionary* in Python is a collection of *key-value pairs*. Each *key* is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'color': 'green', 'points': 5}
```

A *key-value pair* is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'color': 'green'}
```

This dictionary stores one piece of information about `alien_0`, namely the alien's color. The string `'color'` is a key in this dictionary, and its associated value is `'green'`.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

This returns the value associated with the key `'color'` from the dictionary `alien_0`:

```
green
```

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
alien_0 = {'color': 'green', 'points': 5}
```

Now you can access either the color or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
alien_0 = {'color': 'green', 'points': 5}

❶ new_points = alien_0['points']
❷ print("You just earned " + str(new_points) + " points!")
```

Once the dictionary has been defined, the code at ❶ pulls the value associated with the key `'points'` from the dictionary. This value is then stored in the variable `new_points`. The line at ❷ converts this integer value to a string and prints a statement about how many points the player just earned:

```
You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

Let's add two new pieces of information to the `alien_0` dictionary: the alien's x- and y-coordinates, which will help us display the alien in a particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left

edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At ❶ we add a new key-value pair to the dictionary: key `'x_position'` and value `0`. We do the same for key `'y_position'` at ❷. When we print the modified dictionary, we see the two additional key-value pairs:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify color and point value, and two more specify the alien's position. Notice that the order of the key-value pairs does not match the order in which we added them. Python doesn't care about the order in which you store each key-value pair; it cares only about the connection between each key and its value.

Starting with an Empty Dictionary

It's sometimes convenient, or even necessary, to start with an empty dictionary and then add each new item to it. To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add color and point values to it. The result is the dictionary we've been using in previous examples:

```
{'color': 'green', 'points': 5}
```

Typically, you'll use empty dictionaries when storing user-supplied data in a dictionary or when you write code that generates a large number of key-value pairs automatically.

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

```
alien_0 = {'color': 'green'}
print("The alien is " + alien_0['color'] + ".")

alien_0['color'] = 'yellow'
print("The alien is now " + alien_0['color'] + ".")
```

We first define a dictionary for `alien_0` that contains only the alien's color; then we change the value associated with the key `'color'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

For a more interesting example, let's track the position of an alien that can move at different speeds. We'll store a value representing the alien's current speed and then use it to determine how far to the right the alien should move:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print("Original x-position: " + str(alien_0['x_position']))

# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
❶ if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # This must be a fast alien.
    x_increment = 3

# The new position is the old position plus the increment.
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print("New x-position: " + str(alien_0['x_position']))
```

We start by defining an alien with an initial *x* position and *y* position, and a speed of `'medium'`. We've omitted the color and point values for the sake of simplicity, but this example would work the same way if you included those key-value pairs as well. We also print the original value of `x_position` to see how far the alien moves to the right.

At ❶, an `if-elif-else` chain determines how far the alien should move to the right and stores this value in the variable `x_increment`. If the alien's speed is `'slow'`, it moves one unit to the right; if the speed is `'medium'`, it moves two