Contents

1 Why learn computer programming?

Programming fosters creativity, reasoning, and problem solving. The programmer gets the opportunity to create something from nothing, use logic to turn programming constructs into a form that a computer can run, and, when things don't work quite as well as expected, use problem solving to figure out what has gone wrong. Programming is a fun, sometimes challenging (and occasionally frustrating) activity, and the skills learned from it can be useful both in school and at work … even if your career has nothing to do with computers.

And, if nothing else, programming is a great way to spend an afternoon when the weather outside is dreary.

2 Why Python?

Python is an easy-to-learn programming language that has some really useful features for a beginning programmer. The code is quite easy to read when compared to other programming languages, and it has an interactive shell into which you can enter your programs and see them run. In addition to its simple language structure and an interactive shell with which to experiment, Python has some features that greatly augment the learning process and allow you to put together simple animations for creating your own games. One is the turtle module, inspired by Turtle graphics (used by the Logo programming language back in the 1960s) and designed for educational use. Another is the tkinter module, an interface for the Tk GUI toolkit, which provides a simple way to create programs with slightly more advanced graphics and animation.

3 Why learn computer programming?

Programming fosters creativity, reasoning, and problem solving. The programmer gets the opportunity to create something from nothing, use logic to turn programming constructs into a form that a computer can run, and, when things don't work quite as well as expected, use problem solving to figure out what has gone wrong. Programming is a fun, sometimes challenging (and occasionally frustrating) activity, and the

skills learned from it can be useful both in school and at work … even if your career has nothing to do with computers.

And, if nothing else, programming is a great way to spend an afternoon when the weather outside is dreary.

4 Why Python?

Python is an easy-to-learn programming language that has some really useful features for a beginning programmer. The code is quite easy to read when compared to other programming languages, and it has an interactive shell into which you can enter your programs and see them run. In addition to its simple language structure and an interactive shell with which to experiment, Python has some features that greatly augment the learning process and allow you to put together simple animations for creating your own games. One is the turtle module, inspired by Turtle graphics (used by the Logo programming language back in the 1960s) and designed for educational use. Another is the tkinter module, an interface for the Tk GUI toolkit, which provides a simple way to create programs with slightly more advanced graphics and animation.

5 introduction to python

A computer program is a set of instructions that causes a computer to perform some kind of action. It isn't the physical parts of a computer—like the wires, micro- chips, cards, hard drive, and such—but the hidden stuff running on that hardware. A computer program, which I'll usually refer to as just a program, is the set of commands that tell that dumb hardware what to do. Software is a collection of computer programs.

Without computer programs, almost every device you use daily would either stop working or be much less useful than it is now. Computer programs, in one form or another, control not only your personal computer but also video game systems, cell phones, and the GPS units in cars. Software also controls less obvious items like LCD TVs and their remote controllers, as well as some of the newest radios, DVD players, ovens, and some fridges. Even car engines, traffic lights, street lamps, train signals, electronic billboards, and elevators are controlled by programs.

Programs are a bit like thoughts. If you didn't have thoughts, you would probably just sit on the floor, staring vacantly and drooling down the front of your shirt. Your thought "get up off the floor" is an instruction, or command, that tells your body to stand up. In the same way, computer programs tell computers what to do.

If you know how to write computer programs, you can do all sorts of useful things. Sure, you may not be able to write programs to control cars, traffic lights, or your fridge (well, at least not at first), but you could create web pages, write your own games, or even make a program to help with your homework.

6 introduction to python

A computer program is a set of instructions that causes a computer to perform some kind of action. It isn't the physical parts of a computer—like the wires, micro- chips, cards, hard drive, and such—but the hidden stuff running on that hardware. A computer program, which I'll usually refer to as just a program, is the set of commands that tell that dumb hardware what to do. Software is a collection of computer programs.

Without computer programs, almost every device you use daily would either stop working or be much less useful than it is now. Computer programs, in one form or another, control not only your personal computer but also video game systems, cell phones, and the GPS units in cars. Software also controls less obvious items like LCD TVs and their remote controllers, as well as some of the newest radios, DVD players, ovens, and some fridges. Even car engines, traffic lights, street lamps, train signals, electronic billboards, and elevators are controlled by programs.

Programs are a bit like thoughts. If you didn't have thoughts, you would probably just sit on the floor, staring vacantly and drooling down the front of your shirt. Your thought "get up off the floor" is an instruction, or command, that tells your body to stand up. In the same way, computer programs tell computers what to do.

If you know how to write computer programs, you can do all sorts of useful things. Sure, you may not be able to write programs to control cars, traffic lights, or your fridge (well, at least not at first), but you could create web pages, write your own games, or even make a program to help with your homework.

7 Installing python

I skipped all this

8 Installing python

I skipped all this

9 Hello world

Every programming language starts wit *Hello world*. Let's enter some commands at the prompt, beginning with the following:

print("Hello World")

```
print("Hello World")
```

```
Hello World
```

See..... if prints out the words

Hello World

Exciting huh

10 Calculating with Python

Normally, when asked to find the product of two numbers like 8 × 3.57, you would use a calculator or a pencil and paper. Well, how about using the Python shell to perform your calculation? Let's try it.

8 * 3.57

```
8 * 3.57
```

```
28.56
```

Notice that when entering a multiplication calculation in Python, you use the asterisk symbol (*) instead of a multiplication sign (×).

How about if we try an equation that's a bit more useful? Suppose you are digging in your backyard and uncover a bag of 20 gold coins. The next day, you sneak down to the basement and stick the coins inside your grandfather's steam-powered repli- cating invention (luckily, you can just fit the 20 coins inside). You hear a whiz and a pop and, a few hours later, out shoot another 10 gleaming coins.

How many coins would you have in your treasure chest if you did this every day for a year? On paper, the equations might look like this:

$10 \times 365 = 3650$

$20 + 3650 = 3670$

Sure, it's easy enough to do these calculations on a calculator or on paper, but we can do all of these calculations with the Python shell as well. First, we multiply 10 coins by 365 days in a year to get 3650. Next, we add the original 20 coins to get 3670.

```
10 * 365


3650
```

```
20 + 3650
```

```
3670
```

Now, what if a raven spots the shiny gold sitting in your bed- room, and every week flies in and manages to steal three coins? How many coins would you have left at the end of the year? Here's how this calculation looks in the shell:

```
3*52
```

```
156
```

```
3670 - 156
```

```
3514
```

First, we multiply 3 coins by 52 weeks in the year. The result is 156. We subtract that number from our total coins (3670), which tells us that we would have 3514 coins remaining at the end of the year.

10.1 Python Operators

Symbol	Operation
+	Addition
-	Subtraction
|Multiplication |/|Division	

10.2 Python Operators

You can do multiplication, addition, subtraction, and division in the Python shell, among other mathematical operations that we won't go into right now. The basic symbols used by Python to perform mathematical operations are called operators, as listed in Table 2-1.

Table 2-1: Basic Python Operators

Symbol|Operation —-|—- +|Addition -|Subtraction *|Multiplication /|Division

The *forward slash* (/) is used for division because it's similar to the division line that you would use when writing a frac- tion. For example, if you had 100 pirates and 20 large barrels and you wanted to calculate how many pirates you could hide in each barrel, you could divide 100 pirates by 20 barrels ($100 \div 20$) by entering 100 / 20 in the Python shell. Just remember that the forward slash is the one whose top falls to the right.

10.3 The Order of Operations

We use parentheses in a programming language to control the order of operations. An operation is anything that uses an operator. Multiplication and division have a higher order than addition and subtraction, which means that they're performed first. In other words, if you enter an equation in Python, multiplication or divi- sion is performed before addition or subtraction.

For example, in the following equation, the numbers 30 and 20 are multiplied first, and the number 5 is added to their product.

```
5 + 30 * 20
```

```
605
```

This equation is another way of saying, "multiply 30 by 20, and then add 5 to the result." The result is 605. We can change the order of operations by adding parentheses around the first two numbers, like so:

```
(5 + 30) * 20
```

```
700
```

The result of this equation is 700 (not 605) because the parenthe- ses tell Python to do the operation in the parentheses first, and then do the operation outside the paren- theses. This example is saying "add 5 to 30, and then multiply the result by 20."

Parentheses can be nested, which means that there can be parentheses inside parentheses, like this:

```
((5 + 30) * 20) / 10
```

```
70.0
```

In this case, Python evaluates the innermost parentheses first, then the outer ones, and then the final division operator. In other words, this equation is saying, "add 5 to 30, then multiply the result by 20, and divide that result by 10." Here's what happens:

- Adding 5 to 30 gives 35.
- Multiplying 35 by 20 gives 700.
- Dividing 700 by 10 gives the final answer of 70.

If we had not used parentheses, the result would be slightly different:

```
5 + 30 * 20 / 10
```

```
65.0
```

Remember that multiplication and division always go before addition and subtraction, unless parentheses are used to control the order of operations.

10.4 Variables Are Like Labels

The word *variable* in programming describes a place to store information such as numbers, text, lists of numbers and text, and so on. Another way of looking at a variable is that it's like a *label* for something.

For example, to create a variable named *fred*, we use an equal sign (=) and then tell Python what information the variable should be the label for. Here, we create the variable fred and tell Python that it labels the number 100 (note that this doesn't mean that another variable can't have the same value):

```
fred = 100
```

To find out what value a variable labels, enter print in the shell, followed by the variable name in parentheses, like this:

```
print(fred)
```

```
100
```

We can also tell Python to change the variable *fred* so that it labels something else. For example, here's how to change *fred* to the number 200:

```
fred = 200
print(fred)
```

```
200
```

On the first line, we say that fred labels the number 200. In the second line, we ask what fred is labeling, just to confirm the change. Python prints the result on the last line.

We can also use more than one label (more than one variable) for the same item:

```
fred = 200
john = fred
print(john)
```

```
200
```

In this example, we're telling Python that we want the name (or variable) *john* to label the same thing as *fred* by using the equal sign between *john* and *fred*.

Of course, *fred* probably isn't a very useful name for a variable because it most likely doesn't tell us anything about what the variable is used for. Let's call our variable *number_of_coins* instead of *fred*, like this:

```
number_of_coins = 200
print(number_of_coins)
```

```
200
```

This makes it clear that we're talking about 200 coins.

Variable names can be made up of letters, numbers, and the underscore character (_), but they can't start with a number. You can use anything from single letters (such as a) to long sentences for variable names. (A variable can't contain a space, so use an underscore to separate words.) Sometimes, if you're doing something quick, a short variable name is best. The name you choose should depend on how meaningful you need the variable name to be.

Now that you know how to create variables, let's look at how to use them.

10.5 Using Variables

Remember our equation for figuring out how many coins you would have at the end of the year if you could magically create new coins with your grandfather's crazy invention in the basement? We have this equation:

```
20 + 10 * 365
```

```
3670
```

```
3 * 52
```

```
156
```

```
3670 - 156
```

```
3514
```

We can turn this into a single line of code:

```
20 + 10 * 365 - 3 * 52
```

```
3514
```

Now, what if we turn the numbers into variables? Try entering the following:

```
found_coins = 20
magic_coins = 10
stolen_coins = 3
```

These entries create the variables *found_coins*, *magic_coins*, and *stolen_coins*.

Now, we can reenter the equation like this:

```
found_coins + magic_coins * 365 - stolen_coins * 52
```

```
3514
```

You can see that this gives us the same answer. So who cares, right? Ah, but here's the magic of variables. What if you stick a scarecrow in your win- dow, and the raven steals only two coins instead of three? When we use a variable, we can simply change the variable to hold that new number, and it will change everywhere it is used in the equation. We can change the stolen_coins variable to 2 by entering this:

```
stolen_coins = 2
```

```
found_coins + magic_coins * 365 - stolen_coins * 52
```

```
3566
```

11 Hello world

Every programming language starts wit *Hello world*. Let's enter some commands at the prompt, beginning with the following:

print("Hello World")

```
print("Hello World")
```

```
Hello World
```

See..... if prints out the words

Hello World

Exciting huh

12 Calculating with Python

Normally, when asked to find the product of two numbers like 8 × 3.57, you would use a calculator or a pencil and paper. Well, how about using the Python shell to perform your calculation? Let's try it.

8 * 3.57

```
8 * 3.57
```

```
28.56
```

Notice that when entering a multiplication calculation in Python, you use the asterisk symbol (*) instead of a multiplication sign (×).

How about if we try an equation that's a bit more useful? Suppose you are digging in your backyard and uncover a bag of 20 gold coins. The next day, you sneak down to the basement and stick the coins inside your grandfather's steam-powered repli- cating invention (luckily, you can just fit the 20 coins inside). You hear a whiz and a pop and, a few hours later, out shoot another 10 gleaming coins.

How many coins would you have in your treasure chest if you did this every day for a year? On paper, the equations might look like this:

$10 \times 365 = 3650$

$20 + 3650 = 3670$

Sure, it's easy enough to do these calculations on a calculator or on paper, but we can do all of these calculations with the Python shell as well. First, we multiply 10 coins by 365 days in a year to get 3650. Next, we add the original 20 coins to get 3670.

```
10 * 365
```

```
3650
```

```
20 + 3650
```

```
3670
```

Now, what if a raven spots the shiny gold sitting in your bed- room, and every week flies in and manages to steal three coins? How many coins would you have left at the end of the year? Here's how this calculation looks in the shell:

```
3*52
```

```
156
```

```
3670 - 156
```

```
3514
```

First, we multiply 3 coins by 52 weeks in the year. The result is 156. We subtract that number from our total coins (3670), which tells us that we would have 3514 coins remaining at the end of the year.

12.1 Python Operators

Symbol	Operation
+	Addition
-	Subtraction
|Multiplication |/|Division	

12.2 Python Operators

You can do multiplication, addition, subtraction, and division in the Python shell, among other mathematical operations that we won't go into right now. The basic symbols used by Python to perform mathematical operations are called operators, as listed in Table 2-1.

Table 2-1: Basic Python Operators

Symbol|Operation —-|—- +|Addition -|Subtraction *|Multiplication /|Division

The *forward slash* (/) is used for division because it's similar to the division line that you would use when writing a frac- tion. For example, if you had 100 pirates and 20 large barrels and you wanted to calculate how many pirates you could hide in each barrel, you could divide 100 pirates by 20 barrels ($100 \div 20$) by entering 100 / 20 in the Python shell. Just remember that the forward slash is the one whose top falls to the right.

12.3 The Order of Operations

We use parentheses in a programming language to control the order of operations. An operation is anything that uses an operator. Multiplication and division have a higher order than addition and subtraction, which means that they're performed first. In other words, if you enter an equation in Python, multiplication or divi- sion is performed before addition or subtraction.

For example, in the following equation, the numbers 30 and 20 are multiplied first, and the number 5 is added to their product.

```
5 + 30 * 20
```

```
605
```

This equation is another way of saying, "multiply 30 by 20, and then add 5 to the result." The result is 605. We can change the order of operations by adding parentheses around the first two numbers, like so:

```
(5 + 30) * 20
```

```
700
```

The result of this equation is 700 (not 605) because the parenthe- ses tell Python to do the operation in the parentheses first, and then do the operation outside the paren- theses. This example is saying "add 5 to 30, and then multiply the result by 20."

Parentheses can be nested, which means that there can be parentheses inside parentheses, like this:

```
((5 + 30) * 20) / 10
```

```
70.0
```

In this case, Python evaluates the innermost parentheses first, then the outer ones, and then the final division operator. In other words, this equation is saying, "add 5 to 30, then multiply the result by 20, and divide that result by 10." Here's what happens:

- Adding 5 to 30 gives 35.
- Multiplying 35 by 20 gives 700.
- Dividing 700 by 10 gives the final answer of 70.

If we had not used parentheses, the result would be slightly different:

```
5 + 30 * 20 / 10
```

```
65.0
```

Remember that multiplication and division always go before addition and subtraction, unless parentheses are used to control the order of operations.

12.4 Variables Are Like Labels

The word *variable* in programming describes a place to store information such as numbers, text, lists of numbers and text, and so on. Another way of looking at a variable is that it's like a *label* for something.

For example, to create a variable named *fred*, we use an equal sign (=) and then tell Python what information the variable should be the label for. Here, we create the variable fred and tell Python that it labels the number 100 (note that this doesn't mean that another variable can't have the same value):

```
fred = 100
```

To find out what value a variable labels, enter print in the shell, followed by the variable name in parentheses, like this:

```
print(fred)
```

```
100
```

We can also tell Python to change the variable *fred* so that it labels something else. For example, here's how to change *fred* to the number 200:

```
fred = 200
print(fred)
```

```
200
```

On the first line, we say that fred labels the number 200. In the second line, we ask what fred is labeling, just to confirm the change. Python prints the result on the last line.

We can also use more than one label (more than one variable) for the same item:

```
fred = 200
john = fred
print(john)
```

```
200
```

In this example, we're telling Python that we want the name (or variable) *john* to label the same thing as *fred* by using the equal sign between *john* and *fred*.

Of course, *fred* probably isn't a very useful name for a variable because it most likely doesn't tell us anything about what the variable is used for. Let's call our variable *number_of_coins* instead of *fred*, like this:

```
number_of_coins = 200
print(number_of_coins)
```

```
200
```

This makes it clear that we're talking about 200 coins.

Variable names can be made up of letters, numbers, and the underscore character (_), but they can't start with a number. You can use anything from single letters (such as a) to long sentences for variable names. (A variable can't contain a space, so use an underscore to separate words.) Sometimes, if you're doing something quick, a short variable name is best. The name you choose should depend on how meaningful you need the variable name to be.

Now that you know how to create variables, let's look at how to use them.

12.5 Using Variables

Remember our equation for figuring out how many coins you would have at the end of the year if you could magically create new coins with your grandfather's crazy invention in the basement? We have this equation:

```
20 + 10 * 365
```

```
3670
```

```
3 * 52
```

```
156
```

```
3670 - 156
```

```
3514
```

We can turn this into a single line of code:

```
20 + 10 * 365 - 3 * 52
```

```
3514
```

Now, what if we turn the numbers into variables? Try entering the following:

```
found_coins = 20
magic_coins = 10
stolen_coins = 3
```

These entries create the variables *found_coins*, *magic_coins*, and *stolen_coins*.

Now, we can reenter the equation like this:

```
found_coins + magic_coins * 365 - stolen_coins * 52
```

```
3514
```

You can see that this gives us the same answer. So who cares, right? Ah, but here's the magic of variables. What if you stick a scarecrow in your win- dow, and the raven steals only two coins instead of three? When we use a variable, we can simply change the variable to hold that new number, and it will change everywhere it is used in the equation. We can change the stolen_coins variable to 2 by entering this:

```
stolen_coins = 2
```

```
found_coins + magic_coins * 365 - stolen_coins * 52
```

```
3566
```

13 strings, lists, tuples, and maps (dicts)

In this chapter, we'll work with some other items in Python programs: strings, lists, tuples, and maps. You'll use strings to display messages in your programs (such as "Get Ready" and "Game Over" messages in a game). You'll also discover how lists, tuples, and maps are used to store collections of things.

14 Strings

When programming, we usually call text a string. Think of a string as a collection of letters, and the term makes sense. All the letters, numbers, and symbols in this book could be a string, and so could your name and address. In fact, the first Python program we created in Chapter 1 used a string: "Hello World."

14.1 Creating Strings

In Python, we create a string by putting quotes around text because program- ming languages need to distinguish between different types of values. (We need to tell the computer whether a value is a number, a string, or something else.) For example, we could take our fred variable from Chapter 2 and use it to label a string:

```
fred = "Why do gorillas have big nostrils? Big fingers!!"
```

Then, to see what's inside fred, we could enter print(fred):

```
print(fred)
```

```
Why do gorillas have big nostrils? Big fingers!!
```

You can also use single quotes to create a string, like this:

```
fred = 'What is pink and fluffy? Pink fluff!!'
print(fred)
```

```
What is pink and fluffy? Pink fluff!!
```

However, if you try to enter more than one line of text for your string using only a single (') or double quote (") or if you start with one type of quote and finish with another, you'll get an error message in the Python shell. For example, enter the following line:

```
fred = "How do dinosaurs pay their bills
```

```
  File "<ipython-input-6-5638b6c8f5c0>", line 1
    fred = "How do dinosaurs pay their bills
                                            ^
SyntaxError: EOL while scanning string literal
```

This is an error message complaining about syntax because you did not follow the rules for ending a string with a single or double quote.

Syntax means the arrangement and order of words in a sen- tence or, in this case, the arrangement and order of words and symbols in a program. So SyntaxError means that you did something in an order Python was not expecting, or Python was expecting something that you missed. EOL means end-of-line, so the rest of the error message is telling you that Python hit the end of the line and did not find a double quote to close the string.

To use more than one line of text in your string (called a multiline string), use three single quotes ('"), and then hit enter between lines, like this:

```
fred = '''How do dinosaurs pay their bills?
With tyrannosaurus checks!'''
print(fred)
```

```
How do dinosaurs pay their bills?
With tyrannosaurus checks!
```

14.2 Handling Problems with Strings

Now consider this crazy example of a string, which causes Python to display an error message:

```
silly_string = 'He said, "Aren't can't shouldn't wouldn't."'
```

```
  File "<ipython-input-9-bd5c5f6bf90b>", line 1
    silly_string = 'He said, "Aren't can't shouldn't wouldn't."'
                                   ^
SyntaxError: invalid syntax
```

In the first line, we try to create a string (defined as the vari- able silly_string) enclosed by single quotes, but also containing a mixture of single quotes in the words can't, shouldn't, and wouldn't, as well as double quotes. What a mess!

Remember that Python itself is not as smart as a human being, so all it sees is a string containing He said, "Aren, followed by a bunch of other characters that it doesn't expect. When Python sees a quotation mark (either a single or double quote), it expects a string to start following the first mark and the string to end after the next matching quotation mark (either single or double) on that line. In this case, the start of the string is the single quotation mark before He, and the end of the string, as far as Python is concerned, is the single quote after the n in Aren. The solution to this problem is a multiline string, which we learned about earlier, using three single quotes ('"), which allows us to combine double and single quotes in our string without causing errors. In fact, if we use three single quotes, we can put any combination of single and double quotes inside the string (as long as we don't try to put three single quotes there). This is what the error-free version of our string looks like:

```
silly_string = '''He said, "Aren't can't shouldn't wouldn't."'''
```

```
print(silly_string)
```

```
He said, "Aren't can't shouldn't wouldn't."
```

But wait, there's more. If you really want to use single or dou- ble quotes to surround a string in Python, instead of three single quotes, you can add a backslash () before each quotation mark within the string. This is called escaping. It's a way of saying to Python, "Yes, I know I have quotes inside my string, and I want you to ignore them until you see the end quote."

Escaping strings can make them harder to read, so it's prob- ably better to use multiline strings. Still, you might come across snippets of code that use escaping, so it's good to know why the backslashes are there.

Here are a few examples of how escaping works:

```
single_quote_str = 'He said, "Aren\'t can\'t shouldn\'t wouldn\'t."'
double_quote_str = "He said, \"Aren't can't shouldn't wouldn't.\""
print(single_quote_str)
```

```
He said, "Aren't can't shouldn't wouldn't."
```

```
print(double_quote_str)
```

```
He said, "Aren't can't shouldn't wouldn't."
```

First, at 1, we create a string with single quotes, using the backslash in front of the single quotes inside that string. At 2, we create a string with double quotes, and use the backslash in front of those quotes in the string. In the lines that follow, we print the variables we've just created. Notice that the backslash character doesn't appear in the strings when we print them.

14.3 Embedding Values in Strings

If you want to display a message using the contents of a variable, you can embed values in a string using %s, which is like a marker for a value that you want to add later. (Embedding values, also referred to as string substitution, is programmer-speak for "insert- ing values.") For example, to have Python calculate or store the number of points you scored in a game, and then add it to a sen- tence like "I scored points," use %s in the sentence in place of the value, and then tell Python that value, like this:

```
myscore = 1000
message = 'I scored %s points'
#both approaches work
print(message % (myscore))
print(message % myscore)
```

```
I scored 1000 points
I scored 1000 points
```

Multiple replacements can be made as shown below

```
myscore = 1000
attempts = 5
message = 'I scored %s points. It took me %s attempts'
print(message % (myscore, attempts))
```

```
I scored 1000 points. It took me 5 attempts
```

14.4 Multiplying Strings

What is 10 multiplied by 5? The answer is 50, of course. But what's 10 multiplied by a? Here's Python's answer:

```
print(10 * 'a')
```

```
aaaaaaaaaa
```

Python programmers might use this approach to line up strings with a specific number of spaces when displaying messages in the shell, for example.

```
spaces = ' ' * 25
print('%s 12 Butts Wynd' % spaces)
print('%s Twinklebottom Heath' % spaces)
print('%s West Snoring' % spaces)
print()
print()
print('Dear Sir')
print()
print('I wish to report that tiles are missing from the')
print('outside toilet roof.')
print('I think it was bad wind the other night that blew them away.')
print()
print('Regards')
print('Malcolm Dithering')
```

```
                          12 Butts Wynd
                          Twinklebottom Heath
                          West Snoring


Dear Sir

I wish to report that tiles are missing from the
outside toilet roof.
I think it was bad wind the other night that blew them away.

Regards
Malcolm Dithering
```

In addition to using multiplication for alignment, we can also use it to fill the screen with annoying messages. Try this example for yourself:

```
print(1000 * 'snirt')
```

```
snirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsni
```

15 Lists Are More Powerful than Strings

"Spider legs, toe of frog, eye of newt, bat wing, slug butter, and snake dandruff" is not quite a normal shopping list (unless you happen to be a wizard), but we'll use it as our first example of the differences between strings and lists.

We could store this list of items in the wizard_list variable using a string like this:

```
wizard_list = 'spider legs, toe of frog, eye of newt, bat wing, slug butter, snake dandruff'
print(wizard_list)
```

```
spider legs, toe of frog, eye of newt, bat wing, slug butter, snake dandruff
```

But we could also create a list, a somewhat magical kind of Python object that we can manipulate. Here's what these items would look like written as a list:

```
wizard_list = ['spider legs', 'toe of frog', 'eye of newt','bat wing', 'slug butter', 'snake dandruff']
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'eye of newt', 'bat wing', 'slug butter', 'snake dandruff']
```

Creating a list takes a bit more typing than creating a string, but a list is more useful than a string because it can be manipulated. For example, we could print the third item in the wizard_list (eye of newt) by entering its position in the list (called the index position) inside square brackets ([]), like this:

```
print(wizard_list[2])
```

```
eye of newt
```

?????????

Huh? Isn't it the third item on the list? Yes, but lists start at index position 0, so the first item in a list is 0, the second is 1, and the third is 2. That may not make a lot of sense to humans, but it does to computers.

The code below loops over the list and prints out the index value and the text value for each member - we'll talk about lists later.

```
for index,item in enumerate(wizard_list):
    print ('Index '+str(index)+ ' has a value: '+item)
```

```
Index 0 has a value: spider legs
Index 1 has a value: toe of frog
Index 2 has a value: eye of newt
Index 3 has a value: bat wing
Index 4 has a value: slug butter
Index 5 has a value: snake dandruff
```

We can also change an item in a list much more easily than we could in a string. Perhaps instead of eye of newt we needed a snail tongue. Here's how we would do that with our list:

```
wizard_list[2] = 'snail tongue'
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'snake dandruff']
```

This sets the item in index position 2, previously eye of newt, to snail tongue.

Another option is to show a subset of the items in the list. We do this by using a colon (:) inside square brackets. For example, enter the following to see the third to fifth items in a list (a brilliant set of ingredients for a lovely sandwich):

```
print(wizard_list[2:5])
```

```
['snail tongue', 'bat wing', 'slug butter']
```

Writing [2:5] is the same as saying, **"show the items from index position 2 up to (but not including) index position 5"** —or in other words, items 2, 3, and 4.

Lists can be used to store all sorts of items, like numbers:

```
some_numbers = [1, 2, 5, 10, 20]
```

They can also hold strings:

```
some_strings = ['Which', 'Witch', 'Is', 'Which']
```

They might have mixtures of numbers and strings:

```
numbers_and_strings = ['Why', 'was', 6, 'afraid', 'of', 7,'because', 7, 8, 9]
print(numbers_and_strings)
```

```
['Why', 'was', 6, 'afraid', 'of', 7, 'because', 7, 8, 9]
```

And lists might even store other lists:

```
numbers = [1, 2, 3, 4]
strings = ['I', 'kicked', 'my', 'toe', 'and', 'it', 'is', 'sore']
mylist = [numbers, strings]
print(mylist)
```

```
[[1, 2, 3, 4], ['I', 'kicked', 'my', 'toe', 'and', 'it', 'is', 'sore']]
```

This list-within-list example creates three variables: numbers with four numbers, strings with eight strings, and mylist using numbers and strings. The third list (mylist) has only two elements because it's a list of variable names, not the contents of the variables.

15.1 Adding Items to a List

To add items to a list, we use the append function. A function is a chunk of code that tells Python to do something. In this case, append adds an item to the end of a list.

For example, to add a bear burp (I'm sure there is such a thing) to the wizard's shopping list, do this:

```
wizard_list.append('bear burp')
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'snake dandruff', 'bear burp']
```

You can keep adding more magical items to the wizard's list in the same way, like so:

```
wizard_list.append('mandrake')
wizard_list.append('hemlock')
wizard_list.append('swamp gas')
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'snake dandruff', 'bear burp', 'ma
```

15.2 Removing Items from a List

To remove items from a list, use the del command (short for delete). For example, to remove the sixth item in the wizard's list, snake dandruff, do this:

Remember that positions start at zero, so wizard_list[5] actually refers to the sixth item in the list.

```
del wizard_list[5]

print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'bear burp', 'mandrake', 'hemlock'
```

But what happens if you don't know the index location of a list item? Lets remind ourselves of the list items and their index numbers:

```
for index,item in enumerate(wizard_list):
    print ('Index '+str(index)+ ' has a value: '+item)
```

```
Index 0 has a value: spider legs
Index 1 has a value: toe of frog
Index 2 has a value: snail tongue
Index 3 has a value: bat wing
Index 4 has a value: slug butter
Index 5 has a value: bear burp
Index 6 has a value: mandrake
Index 7 has a value: hemlock
Index 8 has a value: swamp gas
```

How do we find the index number of 'hemlock'? the index function returns the index number of an item:

```
wizard_list.index('hemlock')
```

```
7
```

So if we wanted to delete hemlock from the list we can do the following:

```
del wizard_list[wizard_list.index('hemlock')]

print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'bear burp', 'mandrake', 'swamp ga
```

15.3 List Arithmetic

We can join lists by adding them, just like adding numbers, using a plus (+) sign. For example, suppose we have two lists: list1, con- taining the numbers 1 through 4, and list2, containing some words. We can add them using print and the + sign, like so:

```
list1 = [1, 2, 3, 4]
list2 = ['I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']
print(list1 + list2)
```

```
[1, 2, 3, 4, 'I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']
```

We can also add the two lists and set the result equal to another variable.

```
list1 = [1, 2, 3, 4]
list2 = ['I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']
list3 = list1 + list2
print(list3)
```

```
[1, 2, 3, 4, 'I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']
```

And we can multiply a list by a number. For example, to multiply list1 by 5, we write list1 * 5:

```
list1 = [1, 2]
print(list1 * 5)
```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

On the other hand, division (/) and subtraction (-) give only errors.

But why? Well, joining lists with + and repeating lists with * are straightforward enough operations. These concepts also make sense in the real world. For example, if I were to hand you two paper shopping lists and say, "Add these two lists," you might write out all the items on another sheet of paper in order, end to end. The same might be true if I said, "Multiply this list by 3." You could imagine writing a list of all of the list's items three times on another sheet of paper.

But how would you divide a list? For example, consider how you would divide a list of six numbers (1 through 6) in two. Would we divide the list in the middle, split it after the first item, or just pick some random place and divide it there? There's no simple answer, and when you ask Python to divide a list, it doesn't know what to do, either. That's why it responds with an error.

16 Tuples

A tuple is like a list that uses parentheses, as in this example:

```
fibs = (0, 1, 1, 2, 3)
print(fibs[3])
```

```
2
```

Here we define the variable fibs as the numbers 0, 1, 1, 2, and 3. Then, as with a list, we print the item in index position 3 in the tuple using print(fibs[3]).

The main difference between a tuple and a list is that a tuple cannot change once you've created it. For example, if we try to replace the first value in the tuple fibs with the number 4 (just as we replaced values in our wizard_list), we get an error message:

```
fibs[0] = 4
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-41-534ec6fecf23> in <module>()
----> 1 fibs[0] = 4


TypeError: 'tuple' object does not support item assignment
```

Why would you use a tuple instead of a list? Basically because sometimes it is useful to use something that you know can never change. If you create a tuple with two elements inside, it will always have those two elements inside.

17 Python Maps Won't Help You Find Your Way

In Python, a map (also referred to as a dict, short for dictionary) is a collection of things, like lists and tuples. The difference between maps and lists or tuples is that each item in a map has a key and a corresponding value.

For example, say we have a list of people and their favorite sports. We could put this information into a Python list, with the person's name followed by their sport, like so:

```
favorite_sports = ['Ralph Williams, Football',
'Michael Tippett, Basketball',
'Edward Elgar, Baseball',
'Rebecca Clarke, Netball',
'Ethel Smyth, Badminton',
'Frank Bridge, Rugby']
print(favorite_sports)
```

```
['Ralph Williams, Football', 'Michael Tippett, Basketball', 'Edward Elgar, Baseball', 'Rebecca Clarke, Netb
```

If I asked you what Rebecca Clarke's favorite sport is, you could skim through that list and find the answer is netball. But what if the list included 100 (or many more) people?

Now, if we store this same information in a map, with the person's name as the key and their favorite sport as the value, the Python code would look like this:

```
favorite_sports = {'Ralph Williams' : 'Football',
'Michael Tippett' : 'Basketball',
'Edward Elgar' : 'Baseball',
'Rebecca Clarke' : 'Netball',
'Ethel Smyth' : 'Badminton',
'Frank Bridge' : 'Rugby'}
print(favorite_sports)
```

```
{'Ralph Williams': 'Football', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clarke
```

We use colons to separate each key from its value, and each key and value is surrounded by single quotes. Notice, too, that the items in a map are enclosed in braces ({}), not parentheses or square brackets.

The result is a map (each key maps to a particular value), as shown in Table 3-1.

Table 3-1: Keys Pointing to Values in a Map of Favorite Sports

Key	Value
Ralph Williams	Football
Michael Tippett	Basketball
Edward Elgar	Baseball
Rebecca Clarke	Netball
Ethel Smyth	Badminton
Frank Bridge	Rugby

Now, to get Rebecca Clarke's favorite sport, we access our map favorite_sports using her name as the key, like so:

```
print(favorite_sports['Rebecca Clarke'])
```

```
Netball
```

To delete a value in a map, use its key. For example, here's how to remove Ethel Smyth:

```
del favorite_sports['Ethel Smyth']
print(favorite_sports)
```

```
{'Ralph Williams': 'Football', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clarke
```

To replace a value in a map, we also use its key:

```
print(favorite_sports)
favorite_sports['Ralph Williams'] = 'Ice Hockey'
print(favorite_sports)
```

```
{'Ralph Williams': 'Football', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clarke
{'Ralph Williams': 'Ice Hockey', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clar
```

```
x = {'a': 1, 'b': 2}
y = {'b': 3, 'c': 4}
z = {**x, **y}
print(z)
```

```
{'Ralph Williams': 'Ice Hockey', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clar
```

18 Programming Puzzles

The following are a few experiments you can try yourself. The answers can be found at http://python-for-kids.com/.

18.1 1: Favorites

Make a list of your favorite hobbies and give the list the variable name games. Now make a list of your favorite foods and name the variable foods. Join the two lists and name the result favorites. Finally, print the variable favorites.

18.2 2: Counting Combatants

If there are 3 buildings with 25 ninjas hiding on each roof and 2 tunnels with 40 samurai hiding inside each tunnel, how many ninjas and samurai are about to do battle? (You can do this with one equation in the Python shell.)

18.3 3: Greetings!

Create two variables: one that points to your first name and one that points to your last name. Now create a string and use placeholders to print your name with a message using those two variables, such as “Hi there, Brando Ickett!”

19 strings, lists, tuples, and maps (dicts)

In this chapter, we’ll work with some other items in Python programs: strings, lists, tuples, and maps. You’ll use strings to display messages in your programs (such as “Get Ready” and “Game Over” messages in a game). You’ll also discover how lists, tuples, and maps are used to store collections of things.

20 Strings

When programming, we usually call text a string. Think of a string as a collection of letters, and the term makes sense. All the letters, numbers, and symbols in this book could be a string, and so could your name and address. In fact, the first Python program we created in Chapter 1 used a string: “Hello World.”

20.1 Creating Strings

In Python, we create a string by putting quotes around text because program- ming languages need to distinguish between different types of values. (We need to tell the computer whether a value is a number, a string, or something else.) For example, we could take our fred variable from Chapter 2 and use it to label a string:

```
fred = "Why do gorillas have big nostrils? Big fingers!!"
```

Then, to see what's inside fred, we could enter print(fred):

```
print(fred)
```

```
Why do gorillas have big nostrils? Big fingers!!
```

You can also use single quotes to create a string, like this:

```
fred = 'What is pink and fluffy? Pink fluff!!'
print(fred)
```

```
What is pink and fluffy? Pink fluff!!
```

However, if you try to enter more than one line of text for your string using only a single (') or double quote (") or if you start with one type of quote and finish with another, you'll get an error message in the Python shell. For example, enter the following line:

```
fred = "How do dinosaurs pay their bills
```

```
  File "<ipython-input-6-5638b6c8f5c0>", line 1
    fred = "How do dinosaurs pay their bills
                                            ^
SyntaxError: EOL while scanning string literal
```

This is an error message complaining about syntax because you did not follow the rules for ending a string with a single or double quote.

Syntax means the arrangement and order of words in a sen- tence or, in this case, the arrangement and order of words and symbols in a program. So SyntaxError means that you did something in an order Python was not expecting, or Python was expecting something that you missed. EOL means end-of-line, so the rest of the error message is telling you that Python hit the end of the line and did not find a double quote to close the string.

To use more than one line of text in your string (called a multiline string), use three single quotes ('"), and then hit enter between lines, like this:

```
fred = '''How do dinosaurs pay their bills?
With tyrannosaurus checks!'''
print(fred)
```

```
How do dinosaurs pay their bills?
With tyrannosaurus checks!
```

20.2 Handling Problems with Strings

Now consider this crazy example of a string, which causes Python to display an error message:

```
silly_string = 'He said, "Aren't can't shouldn't wouldn't."'
```

```
  File "<ipython-input-9-bd5c5f6bf90b>", line 1
    silly_string = 'He said, "Aren't can't shouldn't wouldn't."'
                                   ^
SyntaxError: invalid syntax
```

In the first line, we try to create a string (defined as the vari- able silly_string) enclosed by single quotes, but also containing a mixture of single quotes in the words can't, shouldn't, and wouldn't, as well as double quotes. What a mess!

Remember that Python itself is not as smart as a human being, so all it sees is a string containing He said, "Aren, followed by a bunch of other characters that it doesn't expect. When Python sees a quotation mark (either a single or double quote), it expects a string to start following the first mark and the string to end after the next matching quotation mark (either single or double) on that line. In this case, the start of the string is the single quotation mark before He, and the end of the string, as far as Python is concerned, is the single quote after the n in Aren. The solution to this problem is a multiline string, which we learned about earlier, using three single quotes (‘”), which allows us to combine double and single quotes in our string without causing errors. In fact, if we use three single quotes, we can put any combination of single and double quotes inside the string (as long as we don’t try to put three single quotes there). This is what the error-free version of our string looks like:

```
silly_string = '''He said, "Aren't can't shouldn't wouldn't."'''
```

```
print(silly_string)
```

```
He said, "Aren't can't shouldn't wouldn't."
```

But wait, there’s more. If you really want to use single or dou- ble quotes to surround a string in Python, instead of three single quotes, you can add a backslash () before each quotation mark within the string. This is called escaping. It’s a way of saying to Python, “Yes, I know I have quotes inside my string, and I want you to ignore them until you see the end quote.”

Escaping strings can make them harder to read, so it’s prob- ably better to use multiline strings. Still, you might come across snippets of code that use escaping, so it’s good to know why the backslashes are there.

Here are a few examples of how escaping works:

```
single_quote_str = 'He said, "Aren\'t can\'t shouldn\'t wouldn\'t."'
double_quote_str = "He said, \"Aren't can't shouldn't wouldn't.\""
print(single_quote_str)
```

```
He said, "Aren't can't shouldn't wouldn't."
```

```
print(double_quote_str)
```

```
He said, "Aren't can't shouldn't wouldn't."
```

First, at 1, we create a string with single quotes, using the backslash in front of the single quotes inside that string. At 2, we create a string with double quotes, and use the backslash in front of those quotes in the string. In the lines that follow, we print the variables we’ve just created. Notice that the backslash character doesn’t appear in the strings when we print them.

20.3 Embedding Values in Strings

If you want to display a message using the contents of a variable, you can embed values in a string using %s, which is like a marker for a value that you want to add later. (Embedding values, also referred to as string substitution, is programmer-speak for “insert- ing values.”) For example, to have Python calculate or store the number of points you scored in a game, and then add it to a sen- tence like “I scored points,” use %s in the sentence in place of the value, and then tell Python that value, like this:

```
myscore = 1000
message = 'I scored %s points'
#both approaches work
print(message % (myscore))
print(message % myscore)
```

```
I scored 1000 points
I scored 1000 points
```

Multiple replacements can be made as shown below

```
myscore = 1000
attempts = 5
message = 'I scored %s points. It took me %s attempts'
print(message % (myscore, attempts))
```

```
I scored 1000 points. It took me 5 attempts
```

20.4 Multiplying Strings

What is 10 multiplied by 5? The answer is 50, of course. But what's 10 multiplied by a? Here's Python's answer:

```
print(10 * 'a')
```

```
aaaaaaaaaa
```

Python programmers might use this approach to line up strings with a specific number of spaces when displaying messages in the shell, for example.

```
spaces = ' ' * 25
print('%s 12 Butts Wynd' % spaces)
print('%s Twinklebottom Heath' % spaces)
print('%s West Snoring' % spaces)
print()
print()
print('Dear Sir')
print()
print('I wish to report that tiles are missing from the')
print('outside toilet roof.')
print('I think it was bad wind the other night that blew them away.')
print()
print('Regards')
print('Malcolm Dithering')
```

```
                          12 Butts Wynd
                          Twinklebottom Heath
                          West Snoring


Dear Sir

I wish to report that tiles are missing from the
outside toilet roof.
I think it was bad wind the other night that blew them away.

Regards
Malcolm Dithering
```

In addition to using multiplication for alignment, we can also use it to fill the screen with annoying messages. Try this example for yourself:

```
print(1000 * 'snirt')
```

```
snirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsnirtsni
```

21 Lists Are More Powerful than Strings

"Spider legs, toe of frog, eye of newt, bat wing, slug butter, and snake dandruff" is not quite a normal shopping list (unless you happen to be a wizard), but we'll use it as our first example of the differences between strings and lists.

We could store this list of items in the wizard_list variable using a string like this:

```
wizard_list = 'spider legs, toe of frog, eye of newt, bat wing, slug butter, snake dandruff'
print(wizard_list)
```

```
spider legs, toe of frog, eye of newt, bat wing, slug butter, snake dandruff
```

But we could also create a list, a somewhat magical kind of Python object that we can manipulate. Here's what these items would look like written as a list:

```
wizard_list = ['spider legs', 'toe of frog', 'eye of newt','bat wing', 'slug butter', 'snake dandruff']
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'eye of newt', 'bat wing', 'slug butter', 'snake dandruff']
```

Creating a list takes a bit more typing than creating a string, but a list is more useful than a string because it can be manipulated. For example, we could print the third item in the wizard_list (eye of newt) by entering its position in the list (called the index position) inside square brackets ([]), like this:

```
print(wizard_list[2])
```

```
eye of newt
```

?????????

Huh? Isn't it the third item on the list? Yes, but lists start at index position 0, so the first item in a list is 0, the second is 1, and the third is 2. That may not make a lot of sense to humans, but it does to computers.

The code below loops over the list and prints out the index value and the text value for each member - we'll talk about lists later.

```
for index,item in enumerate(wizard_list):
    print ('Index '+str(index)+ ' has a value: '+item)
```

```
Index 0 has a value: spider legs
Index 1 has a value: toe of frog
Index 2 has a value: eye of newt
Index 3 has a value: bat wing
Index 4 has a value: slug butter
Index 5 has a value: snake dandruff
```

We can also change an item in a list much more easily than we could in a string. Perhaps instead of eye of newt we needed a snail tongue. Here's how we would do that with our list:

```
wizard_list[2] = 'snail tongue'
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'snake dandruff']
```

This sets the item in index position 2, previously eye of newt, to snail tongue.

Another option is to show a subset of the items in the list. We do this by using a colon (:) inside square brackets. For example, enter the following to see the third to fifth items in a list (a brilliant set of ingredients for a lovely sandwich):

```
print(wizard_list[2:5])
```

```
['snail tongue', 'bat wing', 'slug butter']
```

Writing [2:5] is the same as saying, **"show the items from index position 2 up to (but not including) index position 5"** —or in other words, items 2, 3, and 4.

Lists can be used to store all sorts of items, like numbers:

```
some_numbers = [1, 2, 5, 10, 20]
```

They can also hold strings:

```
some_strings = ['Which', 'Witch', 'Is', 'Which']
```

They might have mixtures of numbers and strings:

```
numbers_and_strings = ['Why', 'was', 6, 'afraid', 'of', 7,'because', 7, 8, 9]
print(numbers_and_strings)
```

```
['Why', 'was', 6, 'afraid', 'of', 7, 'because', 7, 8, 9]
```

And lists might even store other lists:

```
numbers = [1, 2, 3, 4]
strings = ['I', 'kicked', 'my', 'toe', 'and', 'it', 'is', 'sore']
mylist = [numbers, strings]
print(mylist)
```

```
[[1, 2, 3, 4], ['I', 'kicked', 'my', 'toe', 'and', 'it', 'is', 'sore']]
```

This list-within-list example creates three variables: numbers with four numbers, strings with eight strings, and mylist using numbers and strings. The third list (mylist) has only two elements because it's a list of variable names, not the contents of the variables.

21.1 Adding Items to a List

To add items to a list, we use the append function. A function is a chunk of code that tells Python to do something. In this case, append adds an item to the end of a list.

For example, to add a bear burp (I'm sure there is such a thing) to the wizard's shopping list, do this:

```
wizard_list.append('bear burp')
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'snake dandruff', 'bear burp']
```

You can keep adding more magical items to the wizard's list in the same way, like so:

```
wizard_list.append('mandrake')
wizard_list.append('hemlock')
wizard_list.append('swamp gas')
print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'snake dandruff', 'bear burp', 'ma
```

21.2 Removing Items from a List

To remove items from a list, use the del command (short for delete). For example, to remove the sixth item in the wizard's list, snake dandruff, do this:

Remember that positions start at zero, so wizard_list[5] actually refers to the sixth item in the list.

```
del wizard_list[5]

print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'bear burp', 'mandrake', 'hemlock'
```

But what happens if you don't know the index location of a list item? Lets remind ourselves of the list items and their index numbers:

```
for index,item in enumerate(wizard_list):
    print ('Index '+str(index)+ ' has a value: '+item)
```

```
Index 0 has a value: spider legs
Index 1 has a value: toe of frog
Index 2 has a value: snail tongue
Index 3 has a value: bat wing
Index 4 has a value: slug butter
Index 5 has a value: bear burp
Index 6 has a value: mandrake
Index 7 has a value: hemlock
Index 8 has a value: swamp gas
```

How do we find the index number of 'hemlock'? the index function returns the index number of an item:

```
wizard_list.index('hemlock')
```

```
7
```

So if we wanted to delete hemlock from the list we can do the following:

```
del wizard_list[wizard_list.index('hemlock')]

print(wizard_list)
```

```
['spider legs', 'toe of frog', 'snail tongue', 'bat wing', 'slug butter', 'bear burp', 'mandrake', 'swamp ga
```

21.3 List Arithmetic

We can join lists by adding them, just like adding numbers, using a plus (+) sign. For example, suppose we have two lists: list1, con- taining the numbers 1 through 4, and list2, containing some words. We can add them using print and the + sign, like so:

```
list1 = [1, 2, 3, 4]
list2 = ['I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']
print(list1 + list2)
```

```
[1, 2, 3, 4, 'I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']
```

We can also add the two lists and set the result equal to another variable.

```
list1 = [1, 2, 3, 4]
list2 = ['I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']
list3 = list1 + list2
print(list3)
```

```
[1, 2, 3, 4, 'I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']
```

And we can multiply a list by a number. For example, to multiply list1 by 5, we write list1 * 5:

```
list1 = [1, 2]
print(list1 * 5)
```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

On the other hand, division (/) and subtraction (-) give only errors.

But why? Well, joining lists with + and repeating lists with * are straightforward enough operations. These concepts also make sense in the real world. For example, if I were to hand you two paper shopping lists and say, "Add these two lists," you might write out all the items on another sheet of paper in order, end to end. The same might be true if I said, "Multiply this list by 3." You could imagine writing a list of all of the list's items three times on another sheet of paper.

But how would you divide a list? For example, consider how you would divide a list of six numbers (1 through 6) in two. Would we divide the list in the middle, split it after the first item, or just pick some random place and divide it there? There's no simple answer, and when you ask Python to divide a list, it doesn't know what to do, either. That's why it responds with an error.

22 Tuples

A tuple is like a list that uses parentheses, as in this example:

```
fibs = (0, 1, 1, 2, 3)
print(fibs[3])
```

```
2
```

Here we define the variable fibs as the numbers 0, 1, 1, 2, and 3. Then, as with a list, we print the item in index position 3 in the tuple using print(fibs[3]).

The main difference between a tuple and a list is that a tuple cannot change once you've created it. For example, if we try to replace the first value in the tuple fibs with the number 4 (just as we replaced values in our wizard_list), we get an error message:

```
fibs[0] = 4
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-41-534ec6fecf23> in <module>()
----> 1 fibs[0] = 4


TypeError: 'tuple' object does not support item assignment
```

Why would you use a tuple instead of a list? Basically because sometimes it is useful to use something that you know can never change. If you create a tuple with two elements inside, it will always have those two elements inside.

23 Python Maps Won't Help You Find Your Way

In Python, a map (also referred to as a dict, short for dictionary) is a collection of things, like lists and tuples. The difference between maps and lists or tuples is that each item in a map has a key and a corresponding value.

For example, say we have a list of people and their favorite sports. We could put this information into a Python list, with the person's name followed by their sport, like so:

```
favorite_sports = ['Ralph Williams, Football',
'Michael Tippett, Basketball',
'Edward Elgar, Baseball',
'Rebecca Clarke, Netball',
'Ethel Smyth, Badminton',
'Frank Bridge, Rugby']
print(favorite_sports)
```

```
['Ralph Williams, Football', 'Michael Tippett, Basketball', 'Edward Elgar, Baseball', 'Rebecca Clarke, Netb
```

If I asked you what Rebecca Clarke's favorite sport is, you could skim through that list and find the answer is netball. But what if the list included 100 (or many more) people?

Now, if we store this same information in a map, with the person's name as the key and their favorite sport as the value, the Python code would look like this:

```
favorite_sports = {'Ralph Williams' : 'Football',
'Michael Tippett' : 'Basketball',
'Edward Elgar' : 'Baseball',
'Rebecca Clarke' : 'Netball',
'Ethel Smyth' : 'Badminton',
'Frank Bridge' : 'Rugby'}
print(favorite_sports)
```

```
{'Ralph Williams': 'Football', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clarke
```

We use colons to separate each key from its value, and each key and value is surrounded by single quotes. Notice, too, that the items in a map are enclosed in braces ({}), not parentheses or square brackets.

The result is a map (each key maps to a particular value), as shown in Table 3-1.

Table 3-1: Keys Pointing to Values in a Map of Favorite Sports

Key	Value
Ralph Williams	Football
Michael Tippett	Basketball
Edward Elgar	Baseball
Rebecca Clarke	Netball
Ethel Smyth	Badminton
Frank Bridge	Rugby

Now, to get Rebecca Clarke's favorite sport, we access our map favorite_sports using her name as the key, like so:

```
print(favorite_sports['Rebecca Clarke'])
```

```
Netball
```

To delete a value in a map, use its key. For example, here's how to remove Ethel Smyth:

```
del favorite_sports['Ethel Smyth']
print(favorite_sports)
```

```
{'Ralph Williams': 'Football', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clarke
```

To replace a value in a map, we also use its key:

```
print(favorite_sports)
favorite_sports['Ralph Williams'] = 'Ice Hockey'
print(favorite_sports)
```

```
{'Ralph Williams': 'Football', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clarke
{'Ralph Williams': 'Ice Hockey', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clar
```

```
x = {'a': 1, 'b': 2}
y = {'b': 3, 'c': 4}
z = {**x, **y}
print(z)
```

```
{'Ralph Williams': 'Ice Hockey', 'Michael Tippett': 'Basketball', 'Edward Elgar': 'Baseball', 'Rebecca Clar
```

24 Programming Puzzles

The following are a few experiments you can try yourself. The answers can be found at http://python-for-kids.com/.

24.1 1: Favorites

Make a list of your favorite hobbies and give the list the variable name games. Now make a list of your favorite foods and name the variable foods. Join the two lists and name the result favorites. Finally, print the variable favorites.

24.2 2: Counting Combatants

If there are 3 buildings with 25 ninjas hiding on each roof and 2 tunnels with 40 samurai hiding inside each tunnel, how many ninjas and samurai are about to do battle? (You can do this with one equation in the Python shell.)

24.3 3: Greetings!

Create two variables: one that points to your first name and one that points to your last name. Now create a string and use placeholders to print your name with a message using those two variables, such as “Hi there, Brando Ickett!”