THE GREAT COURSES®

Topic
Science & Mathematics

Subtopic
Technology

# How to Program

## Computer Science Concepts and Python Exercises

Professor John Keyser
Texas A&M University

Transcript Book

**PUBLISHED BY:**

**THE GREAT COURSES**
**Corporate Headquarters**
**4840 Westfields Boulevard, Suite 500**
**Chantilly, Virginia 20151-2299**
**Phone: 1-800-832-2412**
**Fax: 703-378-3819**
**www.thegreatcourses.com**

## John Keyser, Ph.D.

Professor and Associate
Department Head for Academics
in the Department of Computer
Science and Engineering
Texas A&M University

Dr. John Keyser is a Professor and the Associate Department Head for Academics in the Department of Computer Science and Engineering at Texas A&M University. He has been at Texas A&M since earning his Ph.D. in Computer Science from the University of North Carolina in 2000. As an undergraduate, he earned three bachelor's degrees—in Computer Science, Engineering Physics, and Applied Math—from Abilene Christian University.

Dr. Keyser's interests in physics, math, and computing led him to a career in computer graphics, in which he has been able to combine all three disciplines. He has published several articles in geometric modeling, particularly looking at ways of quantifying and eliminating uncertainty in geometric calculations. He has been a long-standing member of the solid and physical modeling community, including previously serving on the Solid Modeling Association executive committee. He has also published several articles in physically based simulation for graphics, including developing ways to simulate waves, fire, and large groups of rigid objects. As a member of the Brain Networks Laboratory collaboration at Texas A&M, he has worked on developing a new technique for rapidly scanning vast amounts of biological data, reconstructing the geometric structures in that data, and helping visualize the results in effective ways. In addition, he has published papers on a variety of other graphics topics, including rendering and modeling.

Dr. Keyser's teaching has spanned a range of courses, from introductory undergraduate courses in computing and programming to graduate courses in modeling and simulation. Among these, he created a new Programming Studio course that has become required for all Computer Science and Computer Engineering majors at Texas A&M. He has won several teaching awards at Texas A&M, including the Distinguished Achievement Award in Teaching, which he received once at the university level and twice from the Dwight Look College of Engineering. As an Assistant Professor, he was named a Montague Scholar by the Center for Teaching Excellence, and he has received other awards, including the Tenneco Meritorious Teaching Award and the Theta Tau Most Informative Lecturer Award.

Since writing his first computer program more than 35 years ago, Dr. Keyser has loved computer programming. He has particularly enjoyed programming competitions, both as a student competitor and now as a team coach. Of the many computer science classes he took, the most important class turned out to be the one in which he met his wife. In his free time, he enjoys traveling with her and their two daughters. ■

# Table of Contents

## [ INTRODUCTION ]



## [ LECTURE GUIDES ]

## [ SUPPLEMENTAL MATERIAL ]

# HOW TO PROGRAM

## Computer Science Concepts and Python Exercises

As computers are becoming more ingrained in our everyday lives and affecting every field of study, from science to the humanities, more and more people are wanting to learn how computers work. This course will teach you about the fundamental ways that computers operate by teaching you how to program computers.

Using the language Python, you will learn programming, from the most basic commands to the techniques used to develop larger pieces of software. Starting with the first lecture, you will learn about how computers operate and how to write programs to instruct them.

The course begins with a discussion of the most basic programming commands that correspond to the most basic operations in a computer. In the first two lectures, you will learn about variables, basic operations like arithmetic, and text-based input and output.

Throughout the first half of the course, the course covers all of the most common programming operations.

- With conditionals and Boolean expressions, you will learn how to make the computer respond differently to different situations.
- Loops will teach you how to get the computer to repeat the same task for you again and again.
- One of the common things you will want to do is process information that might be stored somewhere else, so you next learn about how to work with files.

- The data you read in from files is often organized in long lists, so the course discusses how to handle lists in Python. This is an area where Python particularly excels, and you will be introduced to some of the features that Python includes for handling lists.

- The course will introduce one of the most powerful ideas in all of computer science—abstraction—and show how functions help you put abstraction into action. Functions let you separate different concepts into different parts of a computer program, and the way these different parts communicate is through parameter passing, so you will learn about this process in detail.

One of the particular benefits of using Python is the ease with which we can write powerful Python programs by making use of large collections of code that other people have written. The way to do this is through Python modules, and you will learn about modules as you reach the halfway point in the course. You will learn how to write powerful Python programs, sometimes with just a line or two of code, by calling functions from these modules.

Throughout the course, you will learn how to put basic programming tools together to form more complete programs. In Lecture 4, you will learn how testing and iterative development help create and improve programs. In Lecture 8, you will learn about the idea of top-down design by building a basic data analysis program—for weather, in this case. Lecture 11 focuses on the debugging process and how to identify and deal with the various errors that people encounter when programming. Lectures 13 and 14 show you how abstraction is important when developing these larger programs. First, top-down design is used, but this time with functions, to show how to create a game. Then, the concept of bottom-up design is introduced, and you learn how it can be used in graphics and robotics applications.

In the second half of the course, you will discover some more advanced development skills. You will learn about event-driven programming and how to can create graphical user interfaces. In addition, you will learn how to use loops and modules to generate random numbers and use plots and graphs to create simulations, such as a retirement portfolio. Next, you will learn about the core ideas of object-oriented programming, including encapsulation, inheritance, and polymorphism. You will learn how to use object-oriented design to group your data together and construct larger programs.

The last portion of the course turns to some slightly more advanced topics. You will learn about how to organize data through data structures and then how algorithms allow us to describe fundamental operations on data. After learning about recursive algorithms, you will look in more detail about a particularly useful data structure, graphs, and some of the algorithms that can run on it.

After the course concludes with a look at a current trend in computing, parallel programming, you will be able to write small programs yourself, as well as have all the tools needed to proceed to more advanced study or larger program development.

# Installing Python and PyCharm

To write your own programs in Python, there are two main pieces of software that you will need to know how to set up: **Python** itself, and an environment for using Python called **PyCharm**.

First, you need to install Python on your computer. This means that you will download a whole set of files that will let you write and run Python programs on your computer. Once you have installed these files, you will be able to execute ".py" files, run an interactive window, and execute basically any Python command.

Second, you're going to want an integrated development environment (IDE). Python comes with an interactive program named interactive development and learning environment (IDLE) that will let you do some simple programming, but it's not an IDE and won't provide nearly the range of features provided in an IDE. It is recommended that you get a full IDE, such as PyCharm. This makes it much easier to write, run, and try out your code. You'll have an application that comes up, and you can write your code in that application, manage files, run code, see output, debug, and more all in that same application.

> Whenever you're following instructions about going to websites to download and install software, keep in mind that things can change. Some of the details might no longer be exactly the same. Where to go, or what to do, might change over time. If you find that things aren't exactly like the following instructions, just treat this as an opportunity for problem solving!

## Python

To install Python, go to **www.python.org**, the official site for Python. You can find documentation, tutorials, and examples of using Python on the website. But for now, install Python on your computer. Follow the link to the downloads page.

You'll probably see two different options: one for Python version 2 and one for Python version 3. Version 3 doesn't work quite the same way as version 2. Many people who were already invested in version 2—because they had large amounts of software written in Python 2 or were already familiar and comfortable with Python 2—chose to keep maintaining and developing software in version 2. As a result, two types of Python continued to be used: the version 2 branch and the version 3 branch. The similarities between these are much greater than the differences, but there are some differences. Python 3 is the more up-to-date version, and for people who aren't tied to any old code from Python 2, Python 3 is better.

**Follow the link to the Python 3 page**. You don't want to get anything called a "development" version—that's a version still being developed, and not fully tested. Look for whatever stable version is right for your operating system. If you're on a mac, you'll probably want the Mac OS X version. If you're on Windows, get the newest Windows version that your operating system can handle. (For example, if you are running Windows XP or older, you might need to download an older version of Python 3.) There's also a Linux version. Whichever system you're on, download the newest stable release that your operating system can handle. For the most part, this should involve clicking a single link.

The installation process should be straightforward. The downloaded files will probably ask for permission to be installed, and it is strongly recommended that you let them install in their suggested location. Chances are that this will go smoothly, and when it does, you should have Python fully installed on your system.

The first thing you can do is bring up the interactive shell, called IDLE. After your download, you should be able to find a program named IDLE somewhere on your machine. If it's not on the desktop or a start menu, you might need to do a search.

Once you've found IDLE, run it. You should see a window pop up with a name like "Python 3.__ shell." There will be a prompt consisting of several greater-than signs. Basically, any command you type into this window will be interpreted as a Python command and executed.

IDLE is useful, especially for trying out something small. But for most of the development in this course, it is recommended that you get a full IDE. There are many Python IDEs to choose from, but PyCharm is a great IDE. It's simple enough that most people can easily use it for basic Python programming, and it's powerful enough that high-end programmers will still use it. The basic edition is free to download, and it has all the tools you could want for this course, such as syntax highlighting, error checking, auto-completion, and a full debugger.

## PyCharm

To download PyCharm, go to its website: **www.jetbrains.com/pycharm**. From there, click on the "Download" link. That should bring you to a page where you can choose which operating system you have—Windows or Mac OS X or Linux—and then click on the free "Community" version of PyCharm to download it. When you click that link, it should download a program that you can run to install PyCharm on your computer. Again, you should let it install into whichever directory it would like.

With PyCharm installed, you should be able to run it. Somewhere on your machine should be a PyCharm application, and you want to run that. When you do, there should be a window that opens, possibly with a "hint" window that you can close. You want to try to get a program running in that PyCharm window. There are two steps.

First, go to the **File** menu in PyCharm. Click on the link **New Project**. When you do that, it should prompt you for the name and location for the new project, along with the interpreter to use. The interpreter should default to the Python version that you just installed. For the project title, it will probably default to one called "untitled." Pick a new name for the project, and feel free to pick a different directory. Finally, click the **Create** link. You will probably need to choose whether the new project is in a new window or not. If you pick a new window, you can start off a project in a clean window.

A "project" is going to be a location where there will be one or more Python files that you are developing. You can think of it more like a directory that will hold Python files. Once you have the directory, you need to create a file that will actually be your program. So, go back to the **File** tab in the PyCharm window, click on **New**, and select **Python File**. It will ask you for a name for the file, and this is where you pick the name for your Python file.

In the main PyCharm window, you should see a mostly blank screen. It might have a single line of Python code. You can ignore that line of code; in fact, you can delete it entirely if you want, or just leave it in. It won't make any difference to your code. That window, though, is where you will type in your program.

Again, let's start with a "Hello, World" program. In that window, type in the following: print ("Hello, World"). As you're typing, you might notice that PyCharm will start filling in things for you—for example, when you open the parentheses, it will automatically generate a close parenthesis, and same with the quotation mark.

Notice that unlike the IDLE window, when you hit enter after this line of code, you don't see the results of this code. To see the results, you have to explicitly tell the computer to run the program. In the PyCharm window, go up to the menu item where it says **Run**, and then select "Run" from that menu. You will have to pick the name of the program you just wrote. When you do, a new window will appear at the bottom of

the PyCharm environment. This is the window showing the output. You should see the words “Hello, World” output there, along with something saying “Process finished with exit code 0.” That last line just means that the program completed without an error.

You might have noticed that there was a green arrow in front of “Run.” After running for the first time, you can click the green arrow at the upper-right corner of the PyCharm window or the green arrow down near the output window to run your code again. The new output will replace the old output, so you won’t notice anything new if you rerun the same code. But you can also make a modification to your code—maybe add another print statement or change what this print statement says. Then, hit the green arrow to see the results down in the output window.

---

You’ve now created your own Python program and run it in the PyCharm IDE. For most of this course, it is recommended that you do all of your development in the PyCharm IDE. You can write your code, run it to see how it works, go back and modify your code, and run it again. It makes it very easy to make modifications and test them. Even if it feels awkward at the moment, as you create more and more programs, it will become very natural to create new projects and new Python files, and run them.

As you create these files in PyCharm, it is saving a copy of that program as a “.py” file on your computer. If you navigate to the directory where you set up the project, you should see the “.py” file that was created. You are able to execute that file directly, because you have installed Python on your computer. So, if you had a program to print “Hello, World,” if you double-click on that file, a window will pop up that prints “Hello, World.” The window will disappear as soon as it does that, because the program ends, so it might go so quickly that you don’t see it, but there will be a window. As you develop programs in the future, you will be able to run the programs this way, if you so choose.

Take some time to practice creating and running programs in the PyCharm IDE.

01

# What Is Programming? Why Python?

The three main goals of this course are to teach you the basic tools of programming, how those basic tools can be used to assemble larger pieces of software, and how data structures and algorithms can help us write programs that deal with deeper and more challenging problems in computer science. In this lecture, you will learn about programming—which is a form of communication, from the programmer to both the computer and other people. Just as language helps us organize and describe ideas for people, programming languages help us organize and describe ideas for the computer.

## [ PROGRAMMING LANGUAGES ]

- When we think of everything that computers can do, it's easy to think of them as incredibly smart and powerful machines that humans can't hope to comprehend. At its heart, a computer is a machine that can do just a few basic things extremely well.

- But in order for a computer to do any of those things we think of as powerful and intelligent, it needs a computer **program**. Computer programmers are the ones who give the computer the power and the brains to do all the amazing things we think of a computer doing.

- A computer only understands a few **commands**, and those commands need to be given in the language the computer understands: binary, which is simply a series of ones and zeros. There are a few people who learn to decipher the ones and zeros that a given computer sees at the lowest level. These machine instructions tell the computer to do one of those few commands it knows how to do.

- The set of commands can be different depending on the family of processor running the code. Each command will have some binary code corresponding to it, and that, along with the data contained, forms the binary sequence of commands.

- Obviously, though, this is not the way people naturally think or communicate. So, to overcome these difficulties, people have developed higher-level **programming languages**. These programming languages let us express commands to the computer in a way that people can more easily comprehend.

- From very early on, these higher-level programming languages, such as Fortran (1957) and COBOL (1959), were developed to let people write code that was independent of any specific computer. As people had new ideas for ways to organize their thinking and how they'd like to express that in code, more languages were developed. Along the way, we've had BASIC, Pascal, C, C++, Python, Java, and so on.

- Each of these languages was developed for people to be able to understand and write instructions to the computer. When people write programs, it's just as important that people understand the code as it is that the computer understands it.

- For programmers, the code we write is a way of taking our ideas and expressing them in a logical, ordered way. The programming languages we use help us do that—to structure our ideas and instructions for the computer in ways that we can understand. Then, there is a separate program, called a **compiler** or **interpreter**, that will convert the program into a series of ones and zeros that the computer will understand.

- Regardless of what language you write a program in, it still has to be translated into computer instructions—those ones and zeros. In fact, we can usually accomplish the exact same thing in many different programming languages. But the reason for the different programming languages is that they all have different strengths and weaknesses, and there's no single "right" or "best" language.

› For this course, we will use a general-purpose language that's easy for you to put on your computer, relatively easy to learn, and useful enough that people use it regularly on real-world projects: Python. In addition to these characteristics, it also doesn't require you to use any one particular programming style—it's a good language for several different styles.

## [ PYTHON BASICS ]

› The following code is an example of a Python program.

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

› This is an instruction that we are giving the computer. This line of code prints the words "Hello, World" to the screen. (The word "print" has nothing to do with putting ink on paper.) For many people, a "Hello, World" program is the first one they write.

```
print ("Hello, World")

OUTPUT:
Hello, World
```

› What's going on for the computer to be able to run this program? First, we have our computer program that we've written. In this case, the program is just a single line of code.

› We could enter this program in a few basic ways. We could type it into an interactive window, known as a shell window, which will give us results as we type our code; or we could enter it within what's called a development environment, in which we type several lines of code—in other words, we develop the program—before we let the program work.

› With either of these methods, we can save the program as a ".py" file, indicating that it's a Python program. We could even just create a ".py" file on our own, in any text editor.

- Regardless of how we type in and run the program, a computer will need that code converted into a set of machine instructions—a bunch of ones and zeros that the computer can understand. Then, it's going to run, or **execute**, those machine instructions. When the computer follows those machine instructions, it does exactly what they tell it to. In this case, it prints out the letters "Hello, World" to the screen.

- Computer programs—the things written in a language that people understand—need to be translated into machine instructions that the computer understands. This translation process can happen in different ways.

- One way will encounter each line of the program and immediately convert it to machine instructions that are run. This is called "interpreting."

- Another method will take the whole program at once and try to figure out the best machine instructions to do what that program meant to do. This is called "compiling."

- The basic process of compiling or interpreting involves taking the individual lines of code and converting them to at least one, and often many more, machine instructions. Some lines of code are very close to what the machine instructions will be while some lines of code can end up being translated into huge amounts of machine code.

- The programs that convert Python to machine commands are difficult to classify. They're basically compilers that act like interpreters. But, in actual use, we can usually treat Python like it's an interpreted language. That means we can go through it line by line and understand what it is supposed to be doing.

- If you hear Python programs referred to as "scripts" or Python itself referred to as a "scripting language," this is partly just a way of saying that Python is an interpreted language that offers some powerful high-level commands. It can be used to automate complex tasks in just a few lines of code.

› We can see how Python acts like an interpreted language if we run it in an interactive window. Python installations come with a program called IDLE, which lets you type in Python commands and see them right away.

## [ COMMENTS AND SYNTAX ]

› Let's try another program that's a little more complicated. Let's say that we're creating a game and want to welcome newcomers to the game. We'll want to greet the user and give an invitation to play. The following is a new program that does this.

```
#Greeting
print ("Howdy!")
#Invitation
print ("Shall we play a game?")
```

› Now we have two lines of instructions for the computer, plus a few one-word comments that are meant for humans. The comments are the lines that begin with the pound sign, which people also call a hashtag, number sign, or hatch mark. When an interpreter sees the pound sign, it ignores everything else on that line.

› Comments are text that's meant for people reading the code. They're ignored by the computer, so their only real purpose is to tell someone reading the code how to understand it. Comments can be critical for helping someone understand the purpose of code and the way it's working. In this case, our comments give the purpose for each of the commands that follow—a greeting followed by an invitation.

```
#Greeting
print ("Howdy!")
#Invitation
print ("Shall we play a game?")
```

```
OUTPUT:
Howdy!
Shall we play a game?
```

- The first line of code prints our greeting: “Howdy!” The interpreter will convert that line to machine instructions that print “Howdy!” to the screen. The next line of code prints out a question: “Shall we play a game?” Again, the interpreter will convert that to instructions that get the computer to print out that text to the screen.

- Notice that we followed the statements in order. The order that instructions are given is the order they’ll be executed. Computer programs are made of long sequences of instructions.

- In addition, notice that we skipped over the comments and the blank lines. Just like comments, blank lines are things we put in there to help people understand the code. Blank lines help separate different parts of the code visually so that it’s clear which things belong together. Throughout your code, it’s a good idea to use comments and blank spaces to help communicate what you’re trying to do.

- Because comments are for people, they can be free-form, but for everything else, there’s syntax, which is the particular way that you need to write and structure your code in a language—Python, in our case—so that the computer understands. Syntax is especially important because computers are only going to understand exactly what they’re told, so we need be very precise in how we talk to them.

- The syntax of Python is designed to support more than one programming style, even within a single program. Python also is constructed so that the language syntax itself is relatively easy for people to follow.

## Reading

Gries, *Practical Programming*, chaps. 1–2.

## Exercises

What would be the output from each of the following lines of code?

```
1  print(1+2+3+4+5)
2  print (3**2 + 4**2)
3  print(3*(5+2))
4  #print(100)
5  print(1/2 + 1/2)
6  print (1//2 + 1//2)
7  print (3985780149 % 2)
```

What would be the code you would write for each of the following?

8. To find the number of items in 8 dozen
9. To find and print the number of weeks in 180 days
10. To print out “I love Python!”
11. A comment to indicate that it is your first program

TRANSCRIPT 01

# What Is Programming? Why Python?

When I say computer, you probably think of something extremely powerful. Computers can calculate numbers at sizes and speeds that humans can't begin to match. They can record our writing and send it almost instantaneously to anyone around the world. They can help us find almost any information we want, from how to repair a faucet to who won the Cowboys' game last night. We've seen computers beat Garry Kasparov at chess and Ken Jennings at Jeopardy. So when we think of everything that computers can do, it's easy to think of them as incredibly smart and powerful machines that humans just can't hope to comprehend.

But computers on their own aren't really that smart. In fact, you might even call them really, really dumb. OK, dumb might be too strong of a word, so let me explain a little bit more. At its heart, a computer is just a machine that can do just a few basic things extremely well. It can do those things very, very reliably, and very, very fast. But on its own, a computer can't do much of anything useful at all. In order for a computer to do any of those things that we think of as so powerful and so intelligent, it needs a computer program. As computer programmers, we are the ones who give the computer the power and the brains to do all the amazing things that we think of computers doing. In essence, the programmers bring the computer to life.

Now a computer really only understands a few commands. And those commands need to be given in the language that the computer understands—that's binary, which is simply a series of ones and zeroes. There are a few people who actually learn to decipher the ones and zeroes that a given computer sees at the lowest level. These machine instructions tell the computer to do one of those very few commands that it knows how to do. These are very simple commands—things like performing basic arithmetic, or comparing two numbers, or loading a value from memory or storing it to memory. The set of commands can

be different depending on the family of processor running the code. Each command will translate to a binary sequence. Data can also be converted into binary. And the combination of these two forms the machine code. Obviously, though, this is not the way that people naturally think or communicate, so to overcome these difficulties, people have developed higher-level programming languages. These programming languages let us express commands to the computer in a way that people can more easily comprehend.

And from very early on, these higher-level programming languages, such as FORTRAN in 1957 and COBOL in 1959, were developed to let people write code that was independent of any specific computer. Over time, as people had new ideas for ways to organize their thinking, and how they'd like to express that in code, more languages were developed. So along the way, we've had BASIC, Pascal, C, C++, Python, Java, and so on. Each of these languages was developed for people to be able to understand and write instructions to the computer. Let me say that again because it's important to understand: Programming languages are developed for people, not for computers. When people write programs, it's just as important that people understand the code as it is that the computer understands it.

For programmers, the code we write is a way of taking our ideas and expressing them in a logical, ordered way. The programming languages we use help us do that; they help us structure our ideas and instructions for the computer in ways that we can understand. Then there is a separate program, called a compiler or an interpreter that will convert the program into a series of ones and zeroes that the computer will understand. Regardless of which language you write a program in, it still has to be translated into computer instructions—those ones and zeroes. In fact, we can usually accomplish the exact same thing in many different programming languages. It seems kind of redundant then to have all these different languages, doesn't it? I mean, shouldn't one language be enough? Well, different programming languages will have different strengths.

Some languages are easier to learn. In fact, there have been languages developed just to teach programming concepts. This was one of the main motivations for the development of Pascal, for instance. Some languages are made so that you can have a really close relationship between the programming language and what actually happens on the computer. FORTRAN and C are examples here. Some languages are developed for a person to be able to read them really easily. COBOL was one of the earliest programming languages, and it was developed with this in mind. Some languages are designed to support a particular style of programming. Java is a language that's designed specifically for object-oriented programming. And then there are languages that let you program easily in multiple styles, and C++ is a great example here. There are a lot of programming languages that are specialized for one particular task. SQL is a language developed for databases. The language R is used for statistics. The main point to realize is that different languages have different strengths, and there's no single right or best language. There will be strengths and weaknesses to every language.

For this course, I wanted to choose a general-purpose language that's easy for you to put on your own computer, that's relatively easy to learn, and that's useful enough that people use it regularly on real-world projects, not just in toy examples. The language that we're going to use is Python, which has all of these characteristics and more. It's easy to get on your computer. Compared to many languages, it's easy to learn, and it's harder to make big mistakes. And it's powerful enough that people use it regularly in real-world projects. There are many professional software projects developed in Python. Python also doesn't require you to use any one particular programming style; it's a good language for several different styles. Oh, and it's fun. In fact, if you're wondering why this language is called Python, it's named after the British comedy troupe Monty Python.

So let's take a look at a very first example. The code you see is an example of a Python program. That's right. The whole program is one line. Now, remember this is an instruction that we're giving the computer. What do you think this does? It's not a trick question though I will tell you that the word print has nothing to do with putting ink on paper. What's

going to happen when we run this code? Well, this line of code prints the words “Hello, world” to the screen. For many people, a Hello World program is the first one they write. In fact, it’s a long-standing tradition among programmers, probably dating to Bell Labs in the 1970s, that your first program in any language is Hello World.

Let’s talk about what’s going on here for the computer to be able to run this program. First, we have our computer program that we’ve written. In this case, the program is one line of code. We could enter this program in a couple of basic ways. We could type it into an interactive window, known as a shell window. The shell window will give us results as we type our code. Or we could enter it within what’s called a development environment. In a development environment, we type several lines of code—in other words, we develop the program before we actually let the program work. With either of these methods, we can save the program as a .py file, indicating that it’s a Python program. We could even just create a .py file on our own in a text editor.

Now regardless of how we type in and run the program, a computer will need that code converted into machine instructions—a whole bunch of ones and zeroes that the computer can understand. Then it’s going to run, or execute, those machine instructions. When the computer follows those machine instructions, it does exactly what they tell it to. In this case, it prints out the letters “Hello, world” to the screen.

Now remember, computer programs, the things written in a language that people understand, need to be translated into machine instructions that the computer understands. This translation process can happen in a couple of different ways. One way will encounter each line of the program and immediately convert it to machine instructions that are run; this is called interpreting. Another method will take the whole program at once and try to figure out the best machine instructions to do what the program meant to do; this is called compiling.

Now the analogy here is a lot like someone translating a human language. When you have an interpreter, they’re basically taking what someone says and immediately translating it into another language. In

programming, an interpreter is going to basically take one command at a time, translate it to machine instructions, and let it run. On the other hand, a compiler is going to take all of the code together, look at all of it, and then figure out what the commands for the computer should be. It's more like hiring someone to read or listen to an entire speech and then producing a new speech in a different language that still gives the same meaning. Sometimes there will be a sentence-by-sentence correlation, but not always. Since this approach is compiling together all the code at once in order to do the translation, we call it a compiler.

Now either way, whether a program is interpreted line by line or compiled all at once, the most important thing to know is that we're taking the program and converting it to machine commands. The basic process of compiling or interpreting involves taking the individual lines of code and converting them to at least one, and often many more, machine instructions. Some lines of code are very close to the machine instructions—for instance, the most basic arithmetic, like adding two numbers together, corresponds to just one machine command. It's a very direct, straightforward translation of a higher-level code to machine code; however, there will be additional machine commands to load in data from memory or to load a specific value into the processor. So even just a simple line of code where two numbers get added together can take three or more machine instructions. And at the other end, some lines of code can end up being translated into huge amounts of machine code—for example, calling a function may require only a single line in our program, but it could involve many, many lines of machine code.

Finding ways to perform this translation from a high-level language that lets people express ideas clearly to machine language that implements the code efficiently is an active area of computer science research. People are always looking for ways to find the optimal translation of high-level ideas to machine instructions.

Now the programs that convert Python to machine commands are actually a little difficult to classify. They're basically compilers that act like interpreters, but in actual use, we can usually treat Python as though

it's an interpreted language. That means we can go through it line by line and understand what it's supposed to be doing. If you hear Python programs referred to as scripts or Python itself referred to as a scripting language, this is partly just a way of saying that Python is an interpreted language that offers some powerful high-level commands. It can be used to automate complex tasks in just a few lines of code.

We can see how Python acts like an interpreted language if we run it in an interactive window. Now Python installations come with a program called IDLE, that lets you type in Python commands and see them run right away. So if we type in the Hello World program we saw earlier, and press Enter in the IDLE window, sure enough, it prints "Hello, world" for us. We can also give basic arithmetic commands and see the results right away. If we type in 2+2, we get 4 back. 8*15; that's 120. 2**5—that's what that double asterisk does, it raises to a power—there's our answer, 32. 2**100; no problem.

Let's try another program that's a little more complicated. Say we're creating a game, and we want to welcome newcomers to the game. We want to greet the user and give an invitation to play. Here's a new program that does this. Now we have two lines of instruction for the computer, plus a couple of one-word comments which are meant for humans. The comments are the lines that begin with the pound sign, which people also call a hashtag or number sign or hatch mark. When an interpreter sees the pound sign, it ignores everything else on that line. Comments are text that's meant for people reading the code; they're totally ignored by the computer. So their only real purpose is to tell someone reading the code how to understand it. Comments can be critical for helping someone understand the purpose of code and the way it's working. In this case, our comments give the purpose for each of the commands that follow—a greeting followed by an invitation.

OK, let's look at our two lines of code. The first prints our greeting. Now I'm from Texas A&M, and our usual greeting there is "howdy," so that's what I'm printing. The interpreter will convert that line of machine instructions that print "howdy" to the screen. The next line of code prints out a question: "Shall we play a game?" Bonus points if you

can recognize what 1983 movie that's from. Again, the interpreter will convert that to instructions that get the computer to print out that text to the screen.

First, notice that we follow the statements in order. The order that instructions are given is the order that they'll be executed. Computer programs are made of long sequences of instructions. Also, notice that we skipped right over the comments and the blank lines. Just like comments, blank lines are things that we put in there for people to understand the code better. Blank lines help separate different parts of the code visually, so it's clear which things belong together. Throughout your code, it's a good idea to use comments and blank spaces to help communicate what you're trying to do. You should never hesitate to put in comments throughout your code. Comments will help you understand your own code better, both as you're writing it, and in the future when you come back to it. And comments are often critical to helping other people understand your code.

Since comments are for people, they can be free-form, but for everything else, there's syntax. Syntax is the particular way that you need to write and structure your code in a language, Python in our case so that the computer understands. Syntax is especially important because, like I said before, computers aren't very smart. They're only going to understand exactly what they're told, and so we need be very precise in how we talk to them. Sometimes that one little change in spelling or punctuation will make all the difference.

For example, commands such as print have to be lowercased. If you try to use uppercase, you'll get a NameError telling you that you've tried to use a name that's not defined. Also, if you read books or programs with Python version 2, the previous version, you'll notice that the print command didn't have to include parentheses. By the time of Python version 3, the parentheses are necessary for the syntax. Why? Well, print used to be just a statement, but now print has become a function, and you need parentheses to use the function. I realize that you probably won't follow that right now, but I want you to understand that there are lots of nice reasons for the way code appears. And best of all, you really

can understand why the syntax works the way that it does. By the time you finish this course, the basic structure of how commands are written will seem much more obvious to you.

One of the nice features of Python is that it supports more than one programming style. For instance, procedural programming is a longstanding programming style that involves organizing your computation around a set of procedures, or actions, to be taken to solve a problem. Python supports this very well with a rich set of functions that it provides and allows programmers to create. We'll start out in this course introducing a procedural style of development.

Python also supports, but doesn't require object-oriented programming. Object-oriented programming organizes the computation around a set of objects and the things that can be done to those objects. Python allows the creation of objects and supports some basic object-oriented principles like inheritance. We'll discuss object-oriented design in the last third of the course. Python even supports a style that we won't need in this course, called functional programming. The syntax of Python is designed to support each of these styles, which can also be combined within a single program.

Python also is constructed so that the language syntax itself is relatively easy for people to follow. One unusual but very nice feature of how Python syntax is made easier for people to understand is the use of indentation. Unlike in most languages, indenting is actually part of the code read by the computer. So whenever you see a top line followed by indented lines underneath, this is not only a helpful way for humans to see the structure of the program at a glance; these same indents are also part of how a computer understands the structure of the program.

Let me use one really common analogy for programming. A computer program is a lot like a recipe. To fix the food, you need some ingredients, and then you follow some instructions. Likewise, running a program takes some ingredients—that's the data—and then follows instructions. When we're programming, we're going to be focusing a lot on how to write the recipe; but what we really care about, the thing that's most

important, is the dish we're cooking. Sometimes we get bogged down in the details of recipe writing—exactly how to tell someone to cook something—and we lose sight of the food that we're trying to create. It can be even easier to get bogged down in the details of programming and lose the big picture of the goal that you're working toward. So as we learn about programming here, I'm going to make sure that you understand the basics—the syntax and techniques of programming that will help you actually create bigger software. Just keep in mind that our coding is the tool that we're using to realize these bigger ideas.

So don't get bogged down and thinking that you need to memorize every last detail about syntax. You'll learn and become familiar with all of the most common commands in this course; however, there is no way that you'll remember all of the syntax for all of the various commands that you'll use when programming, or even the commands that you'll see just in this course. You won't learn every command by heart, and that's totally OK. It's fine if you forget exactly how to do something. The important thing is to know what's possible, preferably along with a general sense of how to do it, and then you can always look up the details of a specific command as needed. That's why there are online references that everyone uses.

By the time you finish this course, there are three main goals that I have in mind that I'd like you to come out with. First, I want you to understand the basic tools of programming—the fundamental commands and structures that you can use to make the computer do what you want it to. We'll spend most of the early part of the course on these items, learning things like variables, and conditionals, and loops, and functions.

Then second, I want you to understand how those basic tools can be used to assemble larger pieces of software. This is where we face issues of design—how we can assemble code to solve a particular problem. We'll talk about different approaches to software development, like top-down, bottom-up, and object-oriented design. And we'll learn how to use debugging techniques, and modules, and packages to improve the code that we can develop. Now these topics will be kind of spread throughout the course, but will be concentrated in the middle lectures.

And finally, I want you to understand how data structures and algorithms can help us write programs that deal with deeper and more challenging problems in computer science. Most of these lectures will come toward the end of the course once we already have some of the basic tools ready to use, and we know how to use them to approach problems. Throughout all of this, though, I want you to get practice in programming. We'll have lots of examples and exercises, but the main thing is that I want you to actually learn to write your own programs and feel confident afterward about continuing to program and going on to learn even more.

Now, this brings me to a point about how you can follow this course. Computer programming is as much of an art as anything else. In fact, one of the most famous and important publications in computer science is a multivolume work entitled *The Art of Computer Programming*. Take any art, like painting. Now you can learn a lot just by having someone explain different painting techniques, and demonstrate them, and show examples of how they're used. This can be very mentally stimulating, and it can give you a much greater appreciation of paintings that you see, but you'll understand painting a whole lot more by actually trying to paint.

As we go through this course, I hope you'll find the descriptions and demonstrations and examples interesting, and I hope watching will give you a greater understanding of the programming process and appreciation for how computer programs are created. But you'll get even more out of the course if you actually try programming yourself. Get into the habit of thinking that any time during the course, it's OK to stop and run a program. Typing it in for yourself gets you following the code much more closely, and you can run that code that I'm showing for yourself. And then, it's also great to change the program that I'm showing and see what happens. It's really OK to experiment. That's one of the great advantages to taking a course this way—you're free to stop and try things out whenever you want.

Now throughout the course, you'll sometimes see a little icon pop up. Those are times that I would particularly suggest that you pause, take

some time to think about the question or problem, and then try some programming yourself to find the answer. Programming that may look too obvious, or too hard, will be a lot more engaging and meaningful if you've already invested some effort toward trying a solution yourself.

Let me give you a little preview of what we'll be talking about in the next lecture. We've already seen that Hello World program. Here's a small extension to that. We're going to add a second line to the program, a line that gets input. The input command will ask for the user's name. The name the user types in will be assigned to the variable called name. Then instead of printing "Hello, world," we're going to print "Hello," name. So can you guess what will happen when we run this program? Have a guess? Let's see what happens when I run it. The program asks my name. I type in "John," and the computer tells me "Hello, John." So the program as a whole will ask a user's name, and then print a statement, saying, "Hello," name-that-got-typed-in. This short program helps illustrate several ideas: input, output, and variables. We'll be learning details about all of this and a whole lot more in our very next lecture.

Now stick around right after this lecture, and I'll show you how to set up Python on your own computer so that you can follow along with the lectures by trying code yourself.

Programming is a great combination of problem solving and creativity. I got hooked writing my first simple computer program on a giant Commodore PET computer back in third grade, and I've loved computer programming ever since. It's also a gateway to what's sometimes called computational thinking. Computational thinking refers to the way we think when using computers to solve problems. It describes how we analyze situations and develop ideas. And it refers to a mindset that many computer scientists see as valuable not just for programming but for facing other tasks in life. So as you learn about programming, it may be that the most important thing you're learning isn't the specific commands to the computer but rather a set of powerful ways to solve problems and exercise your creativity.

I'm honored that you've chosen me to help you learn more about this field that is so interesting, challenging, and just plain fun. I'm looking forward to the next lecture, as we learn some of the most fundamental pieces of computer programming: variables, input, and output.

To write our programs in this course, we're going to want to get you set up to write programs yourself. There are two main pieces of software that I'm going to show you how to set up: Python itself and an environment for using Python called PyCharm.

The first thing you need to do is install Python on your computer. What that means is that you're going to download a whole set of files that will let you write and run Python programs on your computer. Once you've installed these files, you'll be able to execute .py files, run an interactive window, and execute basically any Python command.

The second thing you're going to want is an integrated development environment or IDE. Python comes with an interactive program named Interactive Development and Learning Environment, or IDLE , that will let you do some simple programming, but it's not an IDE; it won't provide nearly the range of features provided in an IDE. I recommend getting a full IDE such as PyCharm. This makes it much, much easier to write, run, and try out your code. You'll have an application that comes up, and you can write your code in that application, manage files, run code, see output, debug, and more, all in that same application.

Now a word of caution, worth keeping in mind whenever you're following instructions about going to websites to download and install software: Things can change. Some of the details may no longer be exactly the same. Where to go or what to do may change over time. So if you find things aren't exactly like these instructions, just treat this as another opportunity for problem-solving.

So to get Python installed, open up a web browser and go to www.python.org. This is the official site for Python, and I'll just note that the entire website is helpful. You can find documentation, tutorials, and examples of using Python on this website. For now, though, we want

to install Python on our computer. So you'll want to follow the link to the Downloads page.

Now you'll probably see two different options, one for Python 2 point something and one for Python 3 point something. Version 3 doesn't work quite the same way as version 2. Many people who were already invested in version 2 because they had large amounts of software written in Python 2, or were just already familiar and comfortable with Python 2, they chose to keep maintaining and developing software in version 2. And so as a result, there are the two types of Python, and both continue to be used—version 2 and version 3.

The similarities between these are much greater than the differences, but there are some differences. Python 3 is the more up-to-date version, and so for those of us who aren't tied to any old code from Python 2, we are better off with Python 3. So follow the link to the Python 3 page. If you're on a Mac, you'll probably want the Mac OS X version. If you're on Windows, get the newest Windows version that your operating system can handle. For example, if you are running Windows XP or older, you might need to download an older version of Python 3. There's also a Linux version. Whichever system you're on, go ahead and download the newest stable release that your operating system can handle.

Now, for the most part, this should involve nothing more than clicking a single link. The installation process should be straightforward. The downloaded files will probably ask for permission to be installed, and I'd strongly recommend letting them install in their suggested location. Chances are that this will all go smoothly, and when it does, you should have Python fully installed on your system.

Now the first thing we can do is bring up the interactive shell, called IDLE. This name may be a nod toward comedian Eric Idle of Monty Python. Anyway, after your download, you should be able to find a program named IDLE somewhere on your machine. If it's not on the desktop or in a start menu, you might need to do a search. Once you've found IDLE, go ahead and run it.

You should see a window pop up with a name like Python 3 point something shell. There will be a prompt showing, consisting of several greater-than signs. Basically, any command you type into this window will be interpreted as a Python command and executed. Now our first lecture showed a Hello World program and some examples of calculator-like applications. Try some of those out in IDLE. As I said before, IDLE is useful, especially for trying out something small. So even though we're going to work in PyCharm later, don't forget that IDLE is there for your use.

Now for most of our development in this course, I'm going to recommend that you get a full IDE. There are many Python IDEs to choose from, but the one I'm going to demonstrate is called PyCharm—that's p-y-c-h-a-r-m. PyCharm is a great IDE. It's simple enough that most people can easily use it for basic Python programming, and it's powerful enough that high-end programmers will still use it.

To download PyCharm, you'll need to go to its website. From there, click on the Download link. That should bring you to a page where you can choose which operating system you have—Windows, or Mac OS X, or Linux—and then click on the free community version of PyCharm to download it. When you click that link, it should download a program that you can run to install PyCharm on your computer. Again, go ahead and let it install, and I'd suggest letting it go to whichever directory it would like. With PyCharm installed, you should be able to run it. Somewhere on your machine should be a PyCharm application, and you want to go ahead and run that.

When you do so, there should be a window that opens. We want to try to get a program running here in that PyCharm window. There will be two steps. First, click on the link New Project. When you do that, it should prompt you for the name and location for the new project, along with the interpreter to use. The interpreter should default to the Python version that you just installed. For the project title, it'll probably default to one called Untitled. Change Untitled to a new name for the project. Finally, click the Create button.

Now a project is going to be a location where there will be one or more Python files that we are developing. You can think of it more like a directory that will hold Python files. Once we have that directory, we'll need to actually create a file that will be our program. So we go back to the File tab in the PyCharm window, click on New, and select Python File. It will ask you for a name for the file, and this is where you pick the name for your Python file.

Now there in the main PyCharm window, you should see a mostly-blank screen. That window though is where you're going to type in your program. Again, let's start with a Hello World program. In that window, type in: print ("Hello, world"). As you're typing, you might notice that PyCharm will start filling things in for you. When you open the parentheses, it'll automatically generate a close parenthesis. Same with the quotation mark.

Notice that unlike the IDLE window, when you hit Enter after this line of code, you don't see the results of this code. To see the results, you have to explicitly tell the computer to run the program. In the PyCharm window, go up to the menu item where it says Run, and then select Run from that menu. You'll have to pick the name of the program you just wrote. When you do, a new window will appear at the bottom of the PyCharm environment. This is the window showing the output. You should see the words "Hello, world" output there, along with something saying, "Process finished with exit code 0." That last line just means that the program completed without an error.

Now you might have noticed that that there was a green arrow in front of Run. After running for the first time, you can click the green arrow at the upper right corner of the PyCharm window or the green arrow down near the output window to run your code again. The new output will replace the old output, so you won't notice anything new if you rerun the same code. But you can make a modification to your code, maybe add another print statement or change what the print statement says; then hit the green arrow to see the results down in the output window. And, that's it. You've now created your own Python program and run it in the PyCharm IDE.

## MORE PYTHON PRACTICE

Let's try writing a new program in the PyCharm IDE. We want to print the results of some of our calculator-like math, like we did in the IDLE window. Inside the parentheses of the print statement, we can put something other than words within quotations. In fact, we can print math. So try writing a line of code to print the number 10. We just have print (10). Is that what you've got? If not, try it out; type that line into the program window in PyCharm, and run the program. It should output 10.

OK, now try something a little bit trickier. Instead of just the number 10, try printing out the results of calculation, like 37/3. What do you write? Well, we have just put 37/3 inside the parentheses. Notice that the output comes to 12.3333, although it ends in a 4, not a 3 like we might expect. That's called a round-off error. Computers don't represent fractions exactly, so there is some approximation whenever you represent a decimal number. For most cases, this tiny little error is so small that it won't affect us.

Let's try one more thing. It's an experiment that I want you to run, to see if you can figure out what these two math symbols do. I want you to take two integers, and use the double slash operation between them. For instance, try writing 10//3; the output here is 3. Also, try using a percentage sign as the operator. Write something like 10%3; the output here is 1. Keep trying this with several different numbers and see if you can figure out what the double slash and the percentage sign are doing.

OK, that might have been kind of tricky. The double slash is doing integer division; it divides the first number by the second number, and the result is the integer quotient. Since 3 goes into 10 3 times, the answer is 3. The percentage sign gives the remainder. So the remainder when 10 is divided by 3 is 1. The percentage sign is also called the modulus operator. Mathematicians would say, "Ten modulo 3." And there you have it—basic calculator-like operations written in the PyCharm IDE. Writing a program like this program is a good way to understand a little bit more about how the computer interprets and computes math.

02

# Variables: Operations and Input/Output

The leap from traditional calculators to the power of computer programming begins when we turn to variables, operations with variables, and input/output commands—the main points of this lecture. For programmers, a variable is a "box" in short-term memory, a place with a label where we put a value. Basic operations in a computer can have different outcomes depending on the type of variable, and we often need to convert data to a different type. When we combine variables with either operations or input/output commands, we get statements that let the computer do virtually everything we regard as impressive.

## [ PROCESSING AND MEMORY ]

› Think of computer programming as based, fundamentally, on boxes: setting up boxes, assigning things into boxes, and doing things inside and among boxes. The manipulation of variables in working memory is what allows computers to become the flexible, powerful, general-purpose machines we all depend on.

› The aspect of a computer that people usually think of first is the **processor**. The **central processing unit** (CPU) is where all the operations of our computer come to be processed. But by itself, the processor is just a very fast, but very dumb, calculator. The processor is located on a board, the **motherboard**, where a whole bunch of other stuff is all connected together.

› This is where **memory** comes in. Memory is where the variables in a program live. Without memory, a computer can still process operations at a very basic level, but for the computer to operate at a higher, more complex level, it needs memory.

- Computer memory is composed of several layers that are categorized by how close they are to the processor. The closer to the CPU, the easier and faster it'll be for the CPU to access that memory. As you get farther away from the CPU, it takes longer to access the memory, but you get more of it, and it generally becomes longer lasting. Critical data might be moved from one layer to another as needed.

- Some memory, called the registers, is built right into the CPU, and some other memory, the cache, is in the same integrated circuit as the CPU. This is all really short-term memory, and most programmers don't need to worry about it.

- The next level of memory is the main one programmers care about—called **main memory**, or sometimes primary memory. This is the main, short-term memory of the computer. It's the working memory where we keep all the things that are running, from the operating system that we're working inside of to the programs that we're currently executing. This memory is still close to the CPU.

- The random access memory (RAM) is also referred to as "volatile" or temporary memory. The way existing technology works, if we lose power to the computer, the data in this memory is lost. When we talk about variables in memory, we're talking about variables held in temporary memory. When we run a program, part of this temporary, working memory is set aside for use by variables in the program.

- Farther away from the CPU, memory is in so-called permanent **storage**—what's also called secondary memory, or just storage. Whether it's a hard-disk drive or a solid-state drive using flash memory, this is where we store longer-term information, such as files. Programmers deal with storage by deciding what data we want to "read in" for attention in temporary, working memory or "write out" to files in storage.

- Beyond secondary memory that is on board the computer, there's also what's called tertiary memory, or remote storage. This is data that's stored more remotely, possibly offline or over a network. Remote storage

can store massive amounts of data—much more than you could have on your computer. The idea of "cloud storage" is that the computer gets to treat this tertiary memory more like secondary memory.

## [ VARIABLES ]

› A good way of thinking about temporary, working memory is as a set of boxes. Each box can contain some piece of information. If we want to use these boxes, we need to be able to refer to them, so every box has a name. It's these boxes—these regions of memory that have a name and can hold a piece of information—that we refer to as **variables**.

› Variables in programming are similar to the variables in algebra, such as *x* and *y*, in that they can take on different **values**. But in programming, variables are much more flexible.

› Each of these variables, the boxes with a unique name, can be used to store information. To do this, we'll have to do a variable **assignment**. If we have a variable *x* and we want that variable to hold the number 3, we can write this with a line of code.

```
x = 3
```

› The left side of an assignment is always going to be the name of a variable—it's the name of the box we're assigning into. In this case, the box is named "*x*."

› The next character is an equal sign, but this equal sign does not mean "equals" in the way you're used to thinking of it. It's actually what we call an assignment operator—that is, it's indicating that we're assigning a value to a variable. The assignment operator always takes the thing on the right side and assigns it to the box on the left side.

- In this case, that thing on the right side is a number, the value 3. The overall result of this line of code is that we take a box of memory, with the name *x*, and put the value 3 into that box. This does not mean that *x* is now forever equal to that thing on the right side, the 3.

- The right side can be something more complicated. It could be numbers with a decimal point, which are called **floating-point numbers**, or floats. Or, we could assign words; for example, *x* is assigned the value "make pizza."

```
x = 3
x = 3.14
x = "Make pizza"
```

- Expressions in quotation marks are called **strings**, because they are formed by characters strung together. The string can be enclosed within either single or double quotation marks—both work the same way.

- Remember, though, that you need quotes to have a string. For example, "pizza" in quotes is a string, but pizza without quotes could be a variable name. If you're trying to refer to the actual letters of the text, you have to enclose the text in quotes.

- As we write programs, we're using variables to keep track of information, and this means that we'll need names for each of our variables. In Python, you can name variables anything you want, subject to a few rules: use only letters, numbers, and underscore; and don't use a number at the beginning. In addition, a few of Python's built-in commands, such as "print," are known as **keywords** and are not to be used as the name of a variable.

- Every variable has a **type**, which is the way that the piece of information inside that box should be understood. **Integers**, floating-point numbers, and strings are all types. In some languages, you need to be very particular about specifying the type of a variable. Python will figure out what the right type is based on what is assigned, but you have to be careful that you understand the type so that you don't make a mistake.

## [ OPERATIONS WITH VARIABLES ]

› One of the most basic operations that computers can do is simple arithmetic. Standard operations—such as addition, subtraction, multiplication, and division—are built into the CPU. Variables in our programs can use these operations to compute new values, just like a calculator would.

› In the following sequence of code, *a*, *b*, and *c* are variables. Note that *a* is used to define *b*, and then *b* is used to define *c*.

```
a = 2+5
b = a-4
c = a*b

MEMORY:
a: 7
b: 3
c: 21
```

› We can assign the variable *a* the value 2+5, which the processor evaluates, adding 2 and 5 together to get 7. So, when we assign a value to variable *b*, we can use the value that's in *a*. If we assign *a*-4 to *b*, then, because *a* has the value 7, *a* minus 4 is just 3, and this is the value stored in *b*. Then, when variable *c* gets assigned the product of *a* and *b*, the values sitting in boxes *a* and *b* are 7 and 3, so their multiplication evaluates to 21.

```
a = 2+5
b = a-4
c = a*b
a = 42

MEMORY:
a: 42
b: 3
c: 21
```

- Notice that if we change the value of one variable, it does not change the value of any other variables, even if those variables were originally defined from it. If we assign the value 42 to *a*, then the value stored in *a*'s box in memory is changed, but the values stored in the boxes for *b* and *c* are not.

- Addition is also defined for strings. When we add two strings together, the result is a new string with one added onto the end of the other. This process, called **concatenation**, is represented by the addition sign. However, there are no equivalent operations defined for subtraction, multiplication, or division.

- Likewise, most arithmetic operations between different types aren't defined. We can't add a string with an integer; it's an undefined operation. An exception is that in Python, we can multiply a string by an integer, to get the string repeated several times. But data of different types normally do not mix.

## [ INPUT/OUTPUT COMMANDS ]

- The third critical aspect of a computer is the input and output, or I/O. Computers can take input from a variety of sources and send output several places.

- Let's assume that our input and output uses a simple text window on the screen. Any input will be from a person typing something into that window via the keyboard, and any output will be text written out to that window.

- Within Python, the print command is a way of printing a line of data to the screen. The print command consists of the word "print" followed by parentheses. Essentially, the material inside the parentheses is going to be printed out. We can also print out multiple items, just listing them in the parentheses but separating them by commas. It doesn't make any difference if you leave a space after each comma or not; the print command will print one space for each comma when it displays the results.

```
a = input("Enter a value: ")
print(a)
b = input()
print("You entered", b)

WINDOW:
Enter a value:
```

- If we want to start a new line, we can print out a string with the code "\n" inside. The \n is called the "newline" character and is interpreted as "end this line and go to the next one."

- To get input from the user, we have a Python command named "input." This can be used to get information that a user types in. Notice that there are parentheses after the word "input"—this is going to be common for most commands. The input command is put as the right side of an assignment statement. It gets whatever value a user types in so that it can be assigned to whatever variable is on the left side.

- To display some text on screen to prompt the user for input, the input command also lets us include a quoted string in the parentheses. This will be printed out right before the user types in input.

- We don't have to give the user a prompt to get input from the user. If our program has an input command that doesn't include a string in the parameters, the program just waits for the user to type something.

- Input and output where we won't have any user prompt is common when we turn to files for input and output. With files, we use commands called "open," "read," and "write," and we don't give files any prompts before reading from them.

- The idea of type becomes especially important with input. In the following lines of code, the first two lines assign variables *a* and *b* by asking a user to input a value for each one. Let's say that the user inputs 1 and 2. The final line is an output **statement**. You might be surprised to learn that the output is not the number 3, but rather 12.

```
a = input("Enter value one:")
b = input ("Enter value two:")
print("The sum is", a+b)

MEMORY:
a: 1
b: 2

WINDOW:
Enter value one:1
Enter value two:2
The sum is 12
```

- Why did this happen? When we read data with the input command, the data we read always comes in as a string—even if the data is numerical.

- We didn't actually read in the numbers 1 and 2. Instead, we read in the character string 1 and the character string 2. So, because of the way the input command works, both variable *a* and variable *b* were strings, not numbers. When we added *a* and *b* together, even though they looked like numbers, we were actually getting a string concatenation, not an addition.

- If we wanted to treat these like numbers, we would have to convert each from a string to a number. To convert a string to an integer or to a float, we write "int" or "float" and then put the string in parentheses right afterward. In the following, we have two strings, *a* and *b*. We can create an integer, *c*, from the value of *a*'s string. And we can create a floating-point number, *d*, from the value in *b*'s string.

```
a = "1"
b = "3.14159"
c = int(a)
d = float(b)
```

## Reading

Matthes, *Python Crash Course*, chap. 2.

## Exercises

For each of the following short programs, what would be the output of the segment of code?

1
```
a = 10
b = 15
a += b
print(a)
```

2
```
a = 10
b = 15
a = b
b = 1
print(a)
```

3
```
a = 10
b = 15
a = a*a+b
print(a)
```

4
```
a = 10
b = 15
a *= a+b
print(a)
```

5
```
a = 10
b = float(a)
print(b)
```

6
```
a = "10"
b = int(a)
print(b)
```

7
```
a = "Welcome"
b = "Home"
print(a,b)
```

8
```
a="Welcome"
b="Home"
print(a+b)
```

9
```
a = "10"
b = "15"
c = a+b
d = int(c)
print(d)
```

What code would you write for each of the following?

**10** Set the price of bread to be 2.00.

**11** Given a price for a loaf of bread, "bread_price," and a price for a block of cheese, "cheese_price," calculate the cost to buy 2 loaves of bread and 3 blocks of cheese.

**12** Get a user's age.

**13** Write a program to form the name of a knight by asking the user for the knight's name and a personality characteristic. The final name should be printed as "Sir <name> the <characteristic>." For example, if the user enters "Robin" and "Brave," you would print "Sir Robin the Brave."

TRANSCRIPT 02

# Variables: Operations and Input/Output

The leap from traditional calculators to the power of computer programming really begins when we turn to variables, operations with variables, and input/output commands.

Let's start with variables. Now, even though the word variable is used in math, don't think that this will be like high school algebra. Instead, when you hear the word variable in computer science, think of a box. In fact, you can think of computer programming as based, fundamentally, on boxes: setting up boxes, assigning things into boxes, and doing things inside and among boxes. The manipulation of variables in working memory is what allows computers to become the flexible, powerful, general-purpose machines that we all depend on.

Now, this might seem surprising since the aspect of a computer that people usually think of first is the processor, like this one. This is a central processing unit, or CPU. That's where all of the operations that our computer has come to be processed. But by itself, the processor is just a very fast, but very dumb, calculator. Take a look: the processor is located on a board, the motherboard, where you've got a whole bunch of stuff all connected together.

Now, this is where memory comes in. Memory is where the variables in a program live. Without memory, a computer can still process operations at a very basic level, just doing some very basic arithmetic. But, for the computer to operate at a higher, more complex level, really to do anything beyond the absolutely most basic operations, it needs memory.

Computer memory is actually composed of several layers categorized by how close they are to the processor. The closer to the CPU, the easier and faster it'll be for the CPU to access that memory. As you get farther away from the CPU, it takes longer to access the memory, but you get more of it, and it generally becomes longer lasting. Critical data may be moved from one layer to another as needed.

Now, there's some memory, called the registers, built right into the CPU, and some other memory, the cache, that's in the same integrated circuit as the CPU. This is all really short-term memory, and most programmers don't really need to worry about it.

The next level of memory is the main one programmers care about, and it's actually called main memory or, sometimes, primary memory. This is the main, short-term memory of the computer. It's the working memory where we keep all of the things that are running, from the operating system that we're working inside of, to the programs that we're currently executing.

In our demo computer, the main, short-term memory is connected to the motherboard over here. The memory is still close to the CPU—it's directly connected on the motherboard with a really fast connection between these two. This memory is the RAM, the random access memory that we see referred to in a computer's specs. It's called random access because every physical location is accessible with equal speed. However, the way existing technology works, if we lose power to the computer, the data that's in this memory is lost, which is why it's also referred to as volatile or temporary memory. When we talk about variables in memory, we're talking about variables held in temporary memory. When we run a program, part of this temporary, working memory is set aside for use by variables in the program.

Now, if you look farther away from the CPU, over here, you get to memory that's in so-called permanent storage, what's also called secondary memory, or just plain storage. And, whether it's a hard drive like this, or a solid-state drive using flash memory, this is where we store longer-term information, like files. As programmers, we deal with storage by deciding what data we want to read in for attention in temporary, working memory, or write out to files back in storage.

Beyond secondary memory that's on board the computer, there's also what's called tertiary memory or remote storage. This is data that's stored more remotely, possibly offline or over a network. Remote storage can store massive amounts of data, far more than you could have on your own little computer. The idea of cloud storage is that

the computer gets to treat this tertiary memory more like secondary memory. As time goes on, there'll probably be less and less distinction between local and remote storage.

OK, let's talk about how all of this relates to the way we write code, starting with main memory. Again, a good way of thinking about this temporary, working memory is as a set of boxes. It might even help to think of an actual physical box. Now, each box can contain some piece of information. If we want to use these boxes, we need to be able to refer to them, so every box is going to have a name. It's these boxes—these regions of memory that have a name and can hold a piece of information—that we refer to as variables. Variables in programming are similar to the variables, like *x* and *y*, that you've used in algebra in that they can take on different values. But in programming, variables are much more flexible.

Each of these variables, the boxes with a unique name, can be used to store information, and to do this, we'll have to do a variable assignment. Now, what if we want to have a variable *x,* and we want that variable to hold the number 3? Well, we can write this with a line of code: $x = 3$. The left-hand side of an assignment is always going to be the name of a variable—it's the name of the box that we're assigning into. In this case, the box is named *x*.

Now, the next character is an equal sign, but it's important to realize that this equal sign does not really mean equals, at least in the way you're used to thinking of it. It's actually what we call an assignment operator—that is, it's indicating that we're assigning a value to a variable. The assignment operator always takes the thing on the right-hand side and assigns it to the box on the left-hand side. In this case, the thing on the right-hand side is a number, the value 3. So, the overall result of this line of code is that we take a box of memory with the name *x*, and we put the value 3 into that box.

Now, to be clear, this does not mean that *x* is now and forevermore equal to that thing on the right-hand side, the 3. This convention of using the equal sign to mean assignment is pretty widespread across computer

languages, going all the way back to FORTRAN. All this means is that the variable *x* has been assigned the value 3. If it helps, you can use the words "is assigned" when you see this equal sign. Now, the right-hand side can actually be something more complicated. It could be numbers with a decimal point, which are called floating-point numbers, or floats for short. Or, we could assign words; for example, *x* is assigned the value "make pizza."

Expressions in quotation marks are called strings since they are formed by characters strung together. The string can be enclosed within either single or double quotation marks—both work the same way. Remember, though, that you need to have quotes to have a string. Pizza in quotes is a string, but pizza without quotes could be a variable name. If you're trying to refer to the actual letters of the text, you have to enclose the text in quotes.

Let's look at variables in a just slightly more complex example. Let's say we follow the previous assignment with a new assignment: *y* gets assigned *x*. Now we have another variable, named *y*, and we're assigning a value to it. The value to assign to *y* is the thing on the right-hand side; in this case, that's the variable *x*. So, when this code executes, we first execute the first line, *x* is assigned 3, and we create a box—a memory location labeled *x*—that contains the value 3. When we execute the next line, we'll take whatever is stored in the box named *x*, and put that value into the box named *y*—so, in this case, both *x* and *y* end up with the value 3 inside.

Now, let's say that we have a third line of code where we assign a value 5 to *x*. Notice that the value of *y* is unchanged. It doesn't matter how *y* was originally defined, all we care about is the current line of code and what it changes. So, assigning 5 to the *x* variable has no effect on *y*. Remember, the equal sign is an assignment operator, and it just tells us to take that thing that's on the right-hand side and assign it to the one on the left. It does not mean that these two things are forever equal to each other. So, when we have an example like you see here, it's critical to understand that changing the assignment to *x* does not make any change at all to *y*, and vice-versa.

As we write programs, we're using variables to keep track of information, and this means that we'll need names for each of our variables. In Python, you can name variables anything you like, subject to a few rules: one, use only letters, numbers, and underscore; and two, don't use a number at the beginning. So, let's check: are all of following valid variable names? TestProGram? Game1? Castle_Anthrax? question3_a? Yep, all of these are perfectly acceptable variable names. We don't have to follow English spelling rules.

OK, what about these? New_year's_day? first-name? 1Game? Cheese Shop? Nope, all of those are invalid: they contain characters like apostrophe, hyphen, a beginning number, and a space. If we use any of those, we'll get what's called a syntax error. Oh, one other rule. A few of Python's built-in commands, such as "print," are known as keywords, and should not be used as the name of a variable.

Now, as long as we follow these rules, the computer doesn't care what we name variables; we just need to be consistent. However, code is written for people as much as for the computer, and so it's important to pick good names. If you're familiar with the story of Adam in Genesis, you might remember that just about the first thing we read of him doing is giving names to animals.

Names are critical because we use them to help us understand and clearly refer to the world around us. Similarly, names are important in programming because they help us understand and clearly refer to the variables that we're dealing with. It helps set up the framework and the mindset that we'll use when we're approaching the problem. If I see a variable named *x*, that doesn't really tell me much. But, if I see one named "interest_rate," I have a pretty good idea that that variable is giving me an interest rate. So, as we write code, we'll try to use good names whenever possible. Of course, if we're doing math, or we're illustrating a similarly general point, we can use generic names like *a*, *b*, *x*, *y*, and so on, just like we would in math.

Now, all of this brings up the notion of variable types. Every variable has a type, which is the way that the piece of information inside that box

should be understood. In this example, *a* is an integer, –5; *b* is a floating-point number, 3.14, or just a "float"; and *c* and *d* are both strings. In some languages, you need to be very particular about specifying the type of a variable. In Python, it'll figure out what the right type is based on what's assigned, but you do have to be careful that you understand the type so that you don't make a silly mistake: you don't want to perform division on a string.

Now let's turn to operations on variables, which brings in the processor. One of the most basic operations that computers can do is simple arithmetic. Your standard operations like addition, subtraction, multiplication, and division, those are all built right into the CPU itself. Variables in our program can use these operations to compute new values, just like a calculator would. Take a look at this sequence of code: *a*, *b*, and *c* are variables, and notice how *a* is used to define *b*, and then *b* is used to define *c*.

So, we can assign the variable *a* the value 2 + 5, which the processor evaluates, adding 2 and 5 together to get 7. So, when we assign a value to variable *b*, we can use the value that's in *a*. So, if we assign *a* – 4 to *b*, then, since *a* has the value 7, *a* – 4 is just 3, and this is the value that gets stored in *b*. Then, when variable *c* gets assigned the product of *a* and *b*, the values sitting in boxes *a* and *b* are 7 and 3, so their multiplication evaluates to 21, and that's what gets stored in *c*.

To see how one variable can be used by another, imagine a case where you want to calculate the value of an investment after earning interest. Let's say that we've got four different variables: interest_rate, defined here as 3%; principal of 1000; interest_earned, given by the formula interest rate times principal; and total_amount, which is principal plus interest earned. In this example, the interest rate and the principal are given, then those two variables are used to calculate the amount of interest earned, and then that third variable is added to the second variable, principal, to get a fourth variable, the total amount.

So, let's figure out this next one together. If we add one more statement on, setting principal to 2000, that will obviously change the value stored

in principal. Now, what happens to interest_earned and total_amount in this case? The answer is: nothing. Remember, these variables are just places in memory. When we update the variable principal, it has no effect on those other variables, even though, in reality, principal and total amount would be linked. For the computer, all the assignment line does is change the value in the box belonging to principal. If we wanted to change those other variables, we'd need to actually repeat those lines of code, or write other lines of code, in order to assign new values to them.

Addition is also defined for strings. When we add two strings together, the result is a new string with one added onto the end of the other. This process is called concatenation, and it's represented by the addition sign. A variable called first_name could be added to a variable called last_name to produce a third variable called full_name. With my own first name and my own last name as values for the first two variables, I could form my full name by combining strings representing my first and last names, which will format nicely if I add a fixed string in the middle consisting of a single space.

However, there are no equivalent operations defined for subtraction, multiplication, or division of strings. "Hello minus World" would be meaningless, as would "Hello times World" or "Hello divided by World." Likewise, most arithmetic operations between different types aren't defined. We can't add a string with an integer—it's an undefined operation. One plus pizza does not give us one pizza. Now, an exception that you may come across is that in Python we can multiply a string by an integer to get the string repeated several times. But don't let this rare exception distract you from the larger point that data of different types normally don't mix.

OK, let's practice. Say you have a variable called number_of_weeks that has some value, say 26. And say you wanted another variable that holds the equivalent number of days. What line of code would you write to compute the number of days? Well, we can define a new variable, number_of_days, and use the first variable to assign the new variable a value that's number_of_weeks times 7.

OK, what if you had a variable containing a city name, like Los Angeles, and another containing a state name, like California, and another containing a string that gives a postal code? What if you wanted one string that was the way you would write this information in a U.S.-style mailing address, where you have city, comma, state, space, zip code? How would you do this? Well, we could create a new variable named address, and we could assign it a string formed by concatenating the city name with a comma, and a space, then the state name, then a space, then the zip code string. Note that the zip code has to be a string before we can add it onto the other strings in the address.

Let me explain one more thing that might not be apparent at first. We'll often want to change the value stored in a variable. After all, it's a variable; it can vary. The box is one size fits all. When we have an assignment, the processing of the right-hand side is done first to get a value, and then it's assigned to the left-hand side. But if you think of the assignment as an algebra statement, it might appear to violate basic rules of math.

For instance, we can have a command like $x = x + 1$. Let's see how this code works. We first encounter the line: $x$ is assigned 3, and so this sets up our variable $x$, and it puts the value 3 into its box. When we come to the next line of code, we'll first look at the right-hand side: $x + 1$. Since the current value of $x$ is 3, the computer evaluates this right-hand side to determine that it should be 4. It's this value of 4 that then gets assigned to the variable on the left-hand side of the assignment, which in this case is $x$. Notice that the order is important: the computer first evaluates the right-hand side to determine what will be assigned, and then it assigns it to the left-hand side.

Remember earlier in the lecture when we had $y = x$, and we later changed the assignment to $x$ and found that it didn't change the value of $y$? Well, the same thing holds here, even though $x$ is on both sides. We could have had any expression on the right-hand side that involved $x$, and it would have used the original value of $x$ for the computation. Only at the very end would the new value be placed into $x$ on the left.

The particular case of increasing the value of a variable by some amount is actually a very common operation in programming; we often find ourselves doing things like counting. And just like a child counts by increasing the number of fingers, adding one more finger to however many were already there, in our programs we're going to want to add one more to whatever is already there. We might be counting how many times a user enters something in, or going through every item in a large group.

For this reason, there's actually a special command, which you see in the example here for *y*. The plus-equal symbol gives us another type of assignment operator that we can call "increases by," but, in this case, it computes the right-hand side, and instead of just placing that value into the variable on the left, it adds it to the left-hand side. This command, *y* increases by 1, means the same thing as if we had written *y* is assigned *y* plus 1. There's a similar command, "decreases by," written with a minus and equal symbol, that lets you subtract the value on the right-hand side from that on the left. You see an example here with *z*. There are also "multiply-by" and "divide-by" commands as well, and you see examples in *m* and *n*.

Let's try an example. Let's say we had a bank account with a starting balance of $1000. Then, let's say we had a withdrawal of some amount, say $20. How would we compute the new balance? Well, we would use one of two commands: we could assign balance a value of balance minus withdrawal_amount; or, we could say balance decreases by the value of withdrawal_amount. Either way would perform the same subtraction and give the same answer.

Now, the third critical aspect of a computer is the input and output, or I/O. Computers can take input from a variety of sources and send output to several places. Now, as for the rest of the stuff that you see inside of a computer, for the most part, it's handling I/O. There are ports used to connect the keyboard or mouse. There are connections for output to video cards, or to files, either on hard drives or on removable media. You can also connect to physical devices like printers or robot motors. This is where data comes in from storage, or over a network, or from

sensors attached to the computer, and as you can tell just from looking, a lot of the computer is devoted to input and output.

Let's assume that our input and output uses a simple text window on the screen. Any input will be from a person typing something into that window via the keyboard, and any output will be text written out to that window. Within Python, the print command is a way of printing a line of data to the screen. The print command consists of the word print followed by parentheses. Essentially, the material inside the parentheses is going to be printed out. We can also print out multiple items, just listing them in the parentheses, but separating them by commas. Now, it doesn't make any difference if you leave a space after each comma or not—the print command will print one space for each comma when it displays the results.

One other thing to note: if we want to start a new line, we can print out a string with the code "\n" inside. The \n is called the newline character, and it's interpreted as "end this line and go on to the next one." So, let's look at some examples to illustrate.

We start by creating three variables: *a* containing a string, *b* containing a floating-point number, and *c* containing an integer. The first print command just prints the value of the variable *a*, which in this case is the string "John." In the second print statement, we print out variables *a*, *b*, and *c*. The values of each are printed, separated by a space, so we get John space 3.14159 space 100. Our next print statements print out Hello, followed by a newline, followed by World. Indeed, Hello and World are printed on different lines. Now, because we separated the terms by commas in the print statement, there's actually an extra space printed before World. If we concatenate those three string terms with a plus sign, we could get rid of this extra space. Finally, notice that we cannot concatenate numbers with a plus sign. Whenever they're defined as numbers, the plus sign means addition, not concatenation.

To get input from the user, we have a Python command helpfully named "input." This can be used to get information that a user types in. Notice that there are parentheses after the word input; this is going to be

common for most commands. The input command is put as the right-hand side of an assignment statement. It gets whatever a user types in so that it can be assigned to whatever variable is on the left-hand side. To display some text on the screen to prompt the user for input, the input command also lets us include a quoted string in the parentheses. This'll be printed out right before the user types in input.

In the first line you see here, the input command is going to print the text, "What is your favorite color?" and then wait for the user to type in a value. That value is going to be assigned to the variable Question3. So, suppose a user types in the value blue. This assigns the value of blue to Question3, and when we print out Question3, we indeed see blue prints out.

Now, we don't have to give the user a prompt to get input from the user. If our program has an input command that doesn't include a string in the parameters, the program just waits for the user to type something. In this case, I'm just waiting on user input that I'll assign to the variable *b*. So, if I type in my name, John, then the string "John" gets assigned to the variable *b*, and the print statement will verify that the value being stored is the same one that was entered.

Input and output, where we won't have any user prompt, is common when we turn to files for input and output. With files, we use commands called "open," "read," and "write," and we don't give files any prompts before reading from them. In any case, input and print alone already give us enough to write many programs.

The idea of type becomes especially important with input. In these lines of code, the first two lines assign variables *a* and *b* by asking a user to input a value for each one. Let's say that the user inputs 1 and 2. The final line is an output statement. What do you think is output? Well, you might be really surprised to learn that the output is not the number 3, but rather 12. Why did this happen?

Well, when we read data with the input command, the data we read always comes in as a string. Let me say that again: whenever we read

data using the input command, even if the data is numerical, the input data always comes in as a string. So, what happened? We didn't actually read in the numbers 1 and 2. Instead, we read in the character string 1 and the character string 2. So, because of the way that the input command works, both variable *a* and variable *b* were actually strings, not numbers. And so, when we added *a* and *b* together, even though they looked like numbers, we were actually getting a string concatenation, not an addition. If we wanted to treat these like numbers, we would have to convert each one from a string to a number. It turns out that that's really easy to do, but we have to remember: data is always input as a string.

To convert a string to an integer or to a float, we write "int" or "float," and then we put the string in parentheses right afterward. Here we have two strings, *a* and *b*. We can create an integer, *c*, from the value of *a*'s string, and we can create a floating-point number, *d*, from the value in *b*'s string.

So, what would happen if we tried to do something weird, either trying to call an integer like 1 a float or trying to call a decimal an int? Well, the first line, where we convert the string 1 to a float still works fine: 1 becomes the number 1.0, which is exactly what we'd hoped for. Obviously, the integer 1 is the same as the floating-point number 1.0. However, there's a problem trying to convert a string such as 3.14159 directly to an integer. Python has no direct way to make a string with decimal places into an integer.

We first need to convert the string 3.14159 to a floating-point number, and then we can convert the float to an integer. To go from float to integer, we can either truncate, or we can use rounding. To truncate, we can—just like with a string—write "int()" and then the float to convert inside the parentheses. This will truncate the floating-point number to an integer—that is, it'll drop off any of the fractional part. Alternatively, we can write "round()" and then the float to convert inside of those parentheses. In this case, we'll round up or down to get the integer closest to that floating-point number.

Let's go back and fix the example where we concatenated 1 plus 2 instead of adding them. If we convert the input to an int before assigning it to *a* and *b*, then *a* and *b* are integers, not strings. Notice

that we enclosed the entire input statement, including its parentheses, in parentheses. Just like in math, you handle the stuff inside the parentheses first. So, for each of our two variables, we read the input from the user first, and we get the result. Then, because the input command always brings in data as a string, we convert that result to an int, and then it's that integer that gets assigned to a variable. So, when we use the print statement to add the two converted variables together, now we do indeed get the sum 3, rather than the concatenation 12. You need to be careful to convert to the correct type when you read input.

Let's try a couple more examples. Say we wanted to get the user's name. What code would ask the user for their name and get the answer? Well, we'd use the input command, passing in text such as "What is your name?" We'd assign this to a variable, like "name." Now, say we wanted to greet the user personally, starting with "Hi." How would we do that?

For output, we'd use the print statement. We could either print out two strings—one being "Hi" and the other being the name—or we could print out a single string formed by concatenating "Hi" with the name. Notice that in the first case, there's a space inserted before printing the name, while in the second case, we have to explicitly include that space in the text. Practice input and output commands for yourself until you're very comfortable with them.

For programmers, a variable is a box in short-term memory, a place with a label where we can put a value. When we assign a value to a variable, it does not mean that the variable and value are equal, only that we've temporarily put a value into that labeled box. Basic operations in a computer can have different outcomes depending on the type of variable. An expression written with the plus symbol will add two numbers, but it'll concatenate two strings. Python's input command always brings in data as a string, even if it's numerical data, so we often need to convert data to a different type. When we combine variables with either operations or input/output commands, we get statements that let the computer do virtually everything we regard as impressive.

To preview some syntax for our next lecture: “if” you’ve understood this lecture, “and” you’re ready to continue, “then” in the next lecture, we’ll learn a new way to make a more powerful statement, known as the conditional, “else” review this lecture again.

## MORE PYTHON PRACTICE

OK, let’s do a math example. Say we wanted to compute the area of a circle in a program. First, we’d want to read in the radius of the circle. What would that part of the code look like? We’d start with an input statement, and we’d pass in some text telling the user to enter a radius. Notice that since we want the radius to be a number, we’ll have to convert the input from a string to a float. OK, now say that we want to compute area. What would that involve?

Well, there are a few ways we could do this, but one would be to create a variable, Pi, that we set to the value of pi, to a few digits. Then, we can compute area by taking Pi times the radius times the radius. We could have just put the value of Pi in that equation instead of defining it separately.

Now, finally, say we wanted to print out the result. What would we do then? Well, we’d again have the print statement. We could print out the text and the answer separated by a comma. Or a different option would be to concatenate the area with the text. Notice that to concatenate, we need two strings. Since area is a float, we have to first convert the float to a string. Fortunately, it’s easy to do that: we just use an “str” command, similar to the way that we used “int” or “float” to convert to an integer or floating point number. That’s it.

We could expand this program to tell us all kinds of other information if we wanted to, like the circumference of a circle, or the area of a different shape. Keep playing around with it and I’ll see you in the next lecture.

03

# Conditionals and Boolean Expressions

Conditional statements let us write programs where we choose our path. As you will learn in this lecture, our choices are based on comparisons and decisions made by if-then-else statements, which we sometimes write in the more compact form of "elif" statements. All these conditional statements are what allow us to get to a huge variety of different outcomes. We can describe those choices by making comparisons between values, and we can make more complicated comparisons and conditionals by using Boolean operators. Conditional statements can form the basis for much of the complex behavior that a computer program can exhibit.

## [ CONDITIONAL IF-THEN STATEMENTS ]

› A **conditional** is a computer command that lets us make a decision about which option to choose. We'll have a clear basis for making that decision—a way to know which choice is the right one. And, depending on that choice, different things will happen. Another term we use for this is **branching**. Our computer code is going to have different branches, and as we walk through our code, we're going to encounter points at which we have to decide which branch to follow.

› Let's start with a simple example that is easily expressed with conditionals. Suppose that a computer-controlled thermostat needs to decide whether or not to turn on the heat. It will have some criteria, usually whether the temperature is above or below some minimum. If it's below, it'll turn on the heater, and if not, it won't.

› Here's what some code to implement that would look like.

```
If the temperature is below 60 degrees...
    Then turn on the heater!
```

```
If the temperature is between 60 and 70 degrees
    and if it is between 8 a.m. and 10 p.m.
        Then turn on the heater!
If the temperature is above 80 degrees...
    Then turn on the air conditioner!
```

- Let's start with this short piece of code using a conditional statement: the "if" statement.

```
if True:
    print("Turning on the heater.")
```

- We start the statement with the word "if," and this tells us that we'll be having a condition. Right after the "if," we have the condition itself. The condition is something that's either true or false. For now, we are going to use the words "True" and "False" for our conditions. But the condition is usually written as a more complex expression that evaluates to be true or false.

- In this example, we'll have a condition that's just plain true, so we just use the word "True." Note that the "T" is capitalized. If it were lowercased, it would just be a variable with the name "true." In Python, it's common for a constant value to be capitalized. Because "True" is a constant value—meaning "true," of course—it's capitalized.

- After the word "True," we have a colon, indicating that now we're going to see what happens if the condition were true.

- After the line beginning with "if," we will have the result of what should happen if the condition is true. This is indented from the if statement by four spaces. A tab is also possible, although that's not the Python standard method. Good editors, such as PyCharm, should automatically convert your tabs to four spaces, but if you

**TIP: WRITING CODE**

There can be many ways of writing code that do the same thing. Some of that code is simpler and easier to understand, but it all has the same effect. The key to writing good code is to understand the range of options available and choose the one that's clear and simple.

are writing your own code, you should be consistent about always using four spaces.

› Indents are part of Python's syntax, a visual way for humans to see the structure of the program in a way that is also, simultaneously, how the computer reads the code; that is, when you indent a line of code in Python, you are telling the computer where a new block of code begins.

› In this case, we have a single print statement that should be executed if the condition were true. For the example, we'll print out a line of text saying "Turning on the heater."

```
OUTPUT:
Turning on the heater.
```

› What should we expect to happen when this actually executes on the computer? We see an output saying "Turning on the heater." The computer comes to the if statement, evaluates the condition—which is true, in this example—and because it is true, it follows the indented code.

› But what if the condition is false? In this case, we won't execute the code that's indented. So, there is no output.

```
if False:
    print("Turning on the heater.")

OUTPUT:
```

› Let's say that we want to print more than one line if the condition is true. We can indent a second line of code—in this case, one that will output "It was too cold." When we execute this code, we have a true condition, so we execute both lines of code, and we get two lines of output: "Turning on the heater" and "It was too cold."

```
if True:
    print("Turning on the heater.")
    print("It was too cold.")
```

```
OUTPUT:
Turning on the heater.
It was too cold.
```

- If the condition were false, we would again have no output.

```
if False:
    print("Turning on the heater.")
    print("It was too cold.")

OUTPUT:
```

- Let's say that we didn't have the second print statement indented—that there's no tab, and it is just lined up with the "if." In this case, where the condition is true, we still get both lines output.

```
if True:
    print("Turning on the heater.")
print("It was too cold.")

OUTPUT:
Turning on the heater.
It was too cold.
```

- But if the condition were false, the print statement "It was too cold" is not indented, so it's not part of the if statement.

```
if False:
    print("Turning on the heater.")
print("It was too cold.")
```

- The computer comes along, sees the if statement, checks the condition—which is false—and skips the indented lines. Then, it goes on to the next line of code—which is not indented—and executes it. So, the statement printing "It was too cold" is executed.

```
if False:
    print("Turning on the heater.")
print("It was too cold.")

OUTPUT:
It was too cold.
```

## CONDITIONAL IF-THEN-ELSE STATEMENTS

- Suppose that the thermostat's decision is much more complicated—for example, maybe it has a different minimum depending on the time of day. And maybe it also controls an air conditioner, such that it turns on the air conditioner if you get above a maximum.

- For this, we want an "if-then-else" statement, which has two sets of code that can be executed: one set if the condition is true and another set if it's false.

```
if True:
    print("Turning on the heater.")
else:
    print("Temperature is fine.")

OUTPUT:
Turning on the heater.
```

- The syntax looks just like the if-then statement, but now we have another line, "else," right afterward. Notice the colon, too. We again have indented code right after that, saying what to do if the condition is false.

- If the condition is true, only the first set of indented code is followed. So, in this case, we have a true condition, so we see the output "Turning on the heater."

› On the other hand, if the condition is false, we would follow the second set of indented code—the part after the "else." So, in this case, we see the output "Temperature is fine."

```
if False:
    print("Turning on the heater.")
else:
    print("Temperature is fine.")

OUTPUT:
Temperature is fine.
```

› When we have an else statement immediately followed by an if statement, we can use an elif—which is short for "else if"—to combine them together without having to indent further. Anytime you have an else-if combination, it's probably much clearer to combine these statements with the elif.

## [ NESTING ]

› When we have indented code, it is just like the other code we have. The computer executes the commands that are indented just like it executes the first commands. That means that within the indented code, we can have another if-then-else statement. We call this **nesting** because one statement is entirely within the bounds of the other. You can think of the first if statement as being the nest and the other if statement as being inside the nest.

› This can continue as long as we want—we can have an if statement inside of an if statement inside of an if statement, etc. We call the statement that encompasses the other one the "outer" statement, and we call the one that's nested inside the "inner" statement or the "nested" statement.

› What if we insert another if statement within the first indented section? Looking at the code, we see that the inner if statement has its own indented code. In this case, where both conditions are true, as we execute the program, we would first output "It is cold" and then output "The heater is already on." The rest of the code would be skipped.

```
if True:
    print("It is cold.")
    if True:
        print("The heater is already on.")
    else:
        print("Turning the Heater on.")
else:
    print("It is warm enough.")

OUTPUT:
It is cold.
The heater is already on.
```

› Let's assume that the first condition was false. In this case, we'd skip the entire indented first section, including the nested if statement. We would go straight to the "else" portion of the outer statement. It wouldn't matter whether the inner if statement's condition was true or false, because that line of code is never reached.

```
if False:
    print("It is cold.")
    if True:
        print("The heater is already on.")
    else:
        print("Turning the Heater on.")
else:
    print("It is warm enough.")

OUTPUT:
It is warm enough.
```

› Up until now, we've written our conditions as either "True" or "False." But that assumes what we want the computer to find. If we know at the time we're writing the code whether it's true or false, we don't even need a condition.

- So, instead of writing “True” or “False” for the condition, we need a way for our condition to be something that can have the value *either* True *or* False. This value that can be either true or false is called a **Boolean**; a Boolean variable is a type of variable that can be either true or false.

- Suppose that we have the Boolean variable “temp_is_low.” Then, our if statement, instead of saying “if True” will say “if temp_is_low.”

- In this example, the output is still “Turning on the heater” whenever “temp_is_low” is true.

```
temp_is_low = True
if temp_is_low:
    print("Turning on the heater.")
else:
    print("Temperature is fine.")

OUTPUT:
Turning on the heater.
```

- On the other hand, if the value is false for “temp_is_low,” then our output would be “Temperature is fine.”

```
temp_is_low = False
if temp_is_low:
    print("Turning on the heater.")
else:
    print("Temperature is fine.")

OUTPUT:
Temperature is fine.
```

- Any false conditional, however nonsensical, will have the same effect: Output would be “Temperature is fine.”

## [ COMPARISONS ]

› Conditionals are really only valuable when we don't know ahead of time whether the condition will be true or false. This means that we need to be able to take an expression and determine whether that expression is true or false. The most common way we do this is with comparisons, which let us compare two values. We can choose how we want to compare them.

› Let's define three variables—*a*, *b*, and *c*—and assign *a* the value 1 and both *b* and *c* the value 2. Then, let's do some comparisons.

```
a = 1
b = 2
c = 2
a > b                    #False
a < b                    #True
a >= b                   #False
a <= b                   #True
```

› If we check whether *a* is greater than *b*, we'll find that this is false because 1 is not greater than 2, while if we compare whether *a* is less than *b*, it's true. The same thing happens if we use greater than or equal to and less than or equal to as our comparisons: 1 is not greater than or equal to 2, but 1 is less than or equal to 2. Notice the syntax: The greater-than sign always comes first, and then the equal sign. The less-than sign comes first, and then the equal sign.

› What happens if we compare *b* and *c*? They are equal; they both have the value 2. So, both "b > c" and "b < c" are false expressions. But when we also check for "b >= c" or "b <= c," they're true.

```
a = 1
b = 2
c = 2
b > c                    #False
b < c                    #False
b >= c                   #True
b <= c                   #True
```

- Let's look at two more comparisons: equal and not equal. Each of these has a special notation you need to use that's probably different than nonprogrammers would expect.

- To compare for equality, we use a double equal sign. If you use only a single equal sign, that is assignment. If you want to compare whether or not *a* and *b* are equal, that's just an expression that can be true or false, but assigning *b* to *a* is a command.

- In Python, an expression and a command are different things: A command performs an action, such as assigning a value to a variable, while an expression is just something that gets evaluated to find a value. If you get an expression mixed up with a command, or vice versa, the Python interpreter will usually catch this for you.

- To compare for inequality—that is, to check whether or not two things are not equal—we use an exclamation point followed by an equal sign. This comparison will be true when the two things being compared do not have the same value.

```
a = 1
b = 2
c = 2
a == b                   #False
a != b                   #True
b == c                   #True
b != c                   #False
a = b                    #This assigns b to a
```

› We can compare *a* and *b*, which have the values 1 and 2, for equality and get a "False," or we can compare whether they are not equal to each other and get true. If we compare *b* and *c*, which both have the value 2, then the equal comparison is true, and the not-equal comparison is false.

## Readings

Gries, *Practical Programming*, chap. 5

Matthes, *Python Crash Course*, chap. 5.

Zelle, *Python Programming*, chap. 7.

## Exercises

Assume that you have the following lines of code.

```
a = 1
b = 2
c = 2
d = "One"
e = "Two"
f = "Three"
g = "one"
```

Would these Boolean expressions be true or false?

```
1  a > b
2  a == b
3  a != b
4  b == c
5  d < e
6  e < f
7  d < g
8  g < e
```

9 `not (a == b)`

10 `b < c or b > c`

11 `(a+1) == b and not b < c`

12 `((a <= b) and (b <= c)) or ((a >= b) and (b >= c))`

What would be the output for each of the following segments of code?

13
```
total_cost = 100.00
days = 3
cost_per_day = total_cost / days
if cost_per_day > 40:
    print("Too expensive")
elif cost_per_day > 30:
    print("Reasonable cost")
elif cost_per_day > 20:
    print("Excellent cost")
else:
    print("Incredible bargain")
```

14
```
age = 67
income = 10000
if (age > 70):
    if (income < 15000):
        print("Eligible for benefits")
    else:
        if (income < 20000):
            print("Eligible for reduced benefits")
        else:
            print("Not eligible for benefits")
else:
    if (income < 20000):
        print("Eligible for reduced benefits")
    else:
        print("Not eligible for benefits")
```

Write code for each of the following.

**15** Rewrite the code in exercise 14 in a simpler way by using a more complex Boolean expression and an elif statement.

**16** Compare a variable "user_guess" to a variable "hidden_answer," and tell the user whether the guess is too low, too high, or exactly right.

**17** Generally, every fourth year is a leap year, but there are exceptions. If the year is divisible by 100, then it is not a leap year, unless the year is also divisible by 400, in which case it is still a leap year. So, 2000 (divisible by 400) is a leap year, 2100 (divisible by 100 but not 400) is not, 2004 (divisible by 4 but not 100) is a leap year, and 2001 (not divisible by 4) is not. Write code that examines a variable and year and prints out "Leap year" or "Not a leap year" for that value. Try writing the code in the following three different ways.

a) As a series of nested if statements
b) As a set of if-elif-else statements
c) As a single if statement with a complex Boolean expression

# Conditionals and Boolean Expressions

When I was a kid, one of my favorite types of books to read were called *Choose Your Own Adventure* stories. The books would make you the main character of the story, and after reading a few pages, they would ask you a question about what you wanted to do next. Depending on which choice you made, you'd go to some particular page and keep reading from there. Then you'd encounter another question, and so on. You could read these books over and over, making different choices, and get a totally different story each time.

Until now, our computer programs have been like reading a book straight through from beginning to end. We have instructions in order, one after the other. You perform command 1, then command 2, then 3, and so on. Now we're going to see how to make choose-your-own-adventure computer programs. We'll get to make choices about what to do, and this will make our programs and the things we can do with them a whole lot more interesting.

We're going to introduce a new type of computer command—the conditional. A conditional lets us make a decision about which option to choose. We'll have a clear basis for making that decision—a way to know which choice is the right one. And depending on that choice, different things will happen. Another term we use for this is branching. Our computer code is going to have different branches, and as we walk through our code, we're going to encounter points at which we have to decide which branch to follow.

Let's start with a simple example that's easily expressed with conditionals. Say a computer-controlled thermostat needs to decide whether or not to turn on the heat. It will have some criteria, usually whether the temperature is above or below some minimum. If it's below, it'll turn on the heater, and if not, it won't. Here's what some code to implement that would look like. We'll see how this statement works in this lecture.

Let's start with this short piece of code using a conditional statement—the if statement. We start the statement with the word if, and this tells us that we'll be having a condition. Right after the if, we have the condition itself. The condition is something that's either true or false. For right now, we are going to use the words true and false for our conditions, but in a few minutes, we'll talk about how to make them much more general. It's important that you understand this. I'm going to just be saying true and false right now, but we'll see in a minute that the condition is usually written as a more complex expression that evaluates to be true or false. In this example, the one you see here, we'll have a condition that's just plain true, so we just use the word True. Notice that the T is capitalized. If the t in true were lowercase, it would just be a variable with the name true. In Python, it's common for a constant value to be capitalized. Since True is a constant value—meaning true, of course—it's capitalized. OK, after the word True, we have a colon, indicating that now we're going to see what happens if the condition were true.

After the line beginning with if, we have the result of what should happen if the condition is true. This is indented in from the if statement by four spaces. A tab is also possible, though that's not the Python standard method. Good editors, like PyCharm, will automatically convert your tabs to four spaces, but if you're writing your own code—say, in a text editor—you should always be consistent about using four spaces. Now, in this case, we have a single print statement that should be executed if the condition were true. For the example, we'll print out a line of text saying, "Turning on the heater."

OK, so now that we've seen this line of code, what should we expect to happen when this actually executes on the computer? Well, we see an output saying, "Turning on the heater." What's happened here is that the computer comes to the statement, it evaluates the condition, which is true in this example, and since it's true, it follows the indented code. But what if the condition is false? Well, in this case, we won't execute the code that's indented, so there's nothing at all output.

Let me encourage you to try coding something like this yourself right now. We're going to have a lot more examples coming up, and you'll

probably understand them better if you try writing the code and testing it yourself as you follow along. So go ahead and see whether you can write some code that will print out, “Dangerous cholesterol levels,” if some condition is true—say if we eat ham and jam a lot. Here’s what that might look like. See, pretty simple. We have an if statement, a condition, a colon, and then we indent the print statement. We have to indent. We have to indent because indentation is how we know, and how the computer knows, what to do if the conditional is true—that is, when you indent a line of code in Python, you are literally telling the computer where a new block of code begins.

All right. Let’s say that we want to print more than one line if the condition is true. We can indent a second line of code, in this case, one that’ll output, “It was too cold.” When we execute this code, we have a true condition, and so we execute both lines of code, and we get two lines of output: “Turning on the heater,” and, “It was too cold.” Now if the condition were false, we’d again have nothing output.

Let’s say that we didn’t have the second print statement indented—that there’s no tab, and it just lined up with the if. In this case, where the condition is true, we still get both lines output, but what if the condition were false? What do you think will happen here? OK, in this case, the print statement, “It was too cold,” is not indented, so it’s not part of the if statement. The computer comes along, sees the if statement, checks the condition, which is false, and skips the indented lines; then it goes on to the next line of code, which is not indented, and it executes it. So the statement printing, “It was too cold,” is executed.

All right. This code that we’ve been looking at is an example of an if statement or an if-then statement. But sometimes we want even more from our conditionals. Suppose our thermostat’s decision is a lot more complicated—for example, maybe it has a different minimum depending on the time of day. And maybe it also controls an air conditioner, so that it turns on the air conditioner if you get above a maximum. Well for this we want an if-then-else statement. If-then-else statements have two sets of code that can be executed: one if the condition is true, and the other set if it’s false. It’s a simple idea, so let’s look at an example.

The syntax looks just like the if-then statement, but now we have another line, else, right afterward. Notice the colon also. We again have indented code right after that, saying what to do if the condition is false. So let's see what happens if this is executed. If the condition is true, only the first set of indented code is followed. So in this case, we have a true condition, and so we see the output: "Turning on the heater." On the other hand, if the condition is false, we would follow the second set of indented code, the part after the else. So in this case, we see the output: "Temperature is fine."

OK, let's try another one. How would you create an if-then-else statement that either prints out, "Dangerous cholesterol levels," if a condition is true, or, "Cholesterol level seems OK," if it's not? Here's what that would look like. Again, it's not too complicated; we've just added on the else line and an indented print statement after it.

Now let's make if-then-else slightly more complicated. We're going to introduce the idea of nesting. When we have indented code, it's just like the other code we have. The computer executes the commands that are indented just like it executes the first commands. That means that within the indented code, we can have another if-then-else statement. We call this nesting because one statement is entirely within the bounds of the other. You can think of the first if statement as being the nest, and the other if statement being inside the nest, and this can continue as long as we want. We can have an if statement inside of an if statement inside of an if statement, etc. We call the statement that encompasses the other one the outer statement, and we call the one that's nested inside the inner statement or the nested statement.

What if we insert another if statement within the first indented section? Looking at the code, we see that the inner if statement has its own indented code. In this case, where both conditions are true, as we execute the program, we would first output, "It is cold," and then output, "The heater is already on." The rest of the code would be skipped. Now let's assume that the first condition was false. In this case, we'd skip the entire indented first section, including the nested if statement. We would go straight to the else portion of the outer statement. It wouldn't

matter a bit whether the inner statement's if condition was true or false, because that line of code never gets reached.

OK, up until now, we've written our conditions as either true or false. But that assumes what we want the computer to find. If we know at the time we're writing the code whether it's true or false, we don't even need a condition. So instead of writing true or false for the condition, we need a way for our condition to be something that can either have the value true or false. This value that can be either true or false is called a Boolean. A Boolean variable is a type of variable that can either be true or false.

Suppose we have a Boolean variable temp_is_low. Then our if statement, instead of saying, "if True," will say, "if temp_is_low." In the example you see, the output is still, "Turning on the heater," whenever temp_is_low is true. On the other hand, if the value is false for temp_is_low, then our output would be: "Temperature is fine." Any false conditional, however nonsensical, will have the same effect; output would be: "Temperature is fine."

Now conditionals are really only valuable when we don't know ahead of time whether the condition will be true or false. This means that we need to be able to take an expression and determine whether that expression is true or false. The most common way we do this is with comparisons. A comparison lets us compare two values. We can pick how we want to compare them.

The easiest way to understand this will be to look at some examples. Let's define three variables, *a*, *b*, and *c*, and assign *a* the value 1 and both *b* and *c* the value 2. Then let's do some comparisons. If we check whether $a > b$, we'll find that this is false since 1 is not greater than 2, while if we compare whether $a < b$, it's true. The same thing happens if we use greater-than or equal, or less-than or equal as our comparisons—1 is not greater than or equal to 2, but 1 is less than or equal to 2. Notice the syntax: the greater-than sign always comes first, then the equal sign. The less-than sign comes first, then the equal sign.

Let's see what happens if we compare *b* and *c*. *b* and *c* are equal; they both have the same value, 2. So, *b* > *c* and *b* < *c* are false expressions. But when we also check for *b* >= *c* or *b* <= *c*, they're true.

Now let's look at two more comparisons: equal and not equal. Each of these has a special notation you need to use that's probably a little different than nonprogrammers would expect. To compare for equality, we use a double equal sign. It's important to use a double equal for a comparison. If you use only a single equal sign, that's an assignment. If you're wanting to compare whether or not *a* and *b* are equal, that's just an expression that can be true or false, but assigning *b* to *a* is a command.

In Python an expression and a command are different things. A command actually performs an action, such as assigning a value to a variable, while an expression is just something that gets evaluated to find a value. If you get an expression mixed up with a command or vice versa, the Python interpreter will usually catch this for you. In other languages, by the way, the line between expression and command is more blurry, and finding those mistakes can be a big headache.

OK, to compare for inequality—that is, to check whether two things are not equal—we use an exclamation point followed by an equal sign. This comparison will be true when the two things being compared do not have the same value. As you can see, we can compare *a* and *b*, which have the values 1 and 2, for equality, and we get a false; or we can compare them for inequality, and, in that case, we get true. If we compare *b* and *c*, which both have the value 2, then the equal comparison is true, and the not-equal comparison is false. And remember, the single equal sign is not a comparison, it's an assignment. So writing *a* = *b* will actually give *a* the value 2 in this case.

Now we're at a point where we can actually write some more interesting code. You've probably seen restaurants that advertise that kids 12 and under eat free. Let's say that we want to put this concept into code. We'll assume that we've already gotten someone's age, and we want to tell them whether or not their meal will cost anything. As you see in the code, we will use a comparison to determine whether or not the

person's age is <= 12. Now the first line probably looks a little confusing. Just to make things clearer, it helps to put in some parentheses. So the code checks whether the person is 12 or under. If they are, then *x* will be true, and we'll tell them that the meal is free. If they're not, then *x* will be false, and we'll tell them they have to pay.

Here's a tip. Now doing the comparison, assigning it to a variable, and then checking whether the variable is true or false is more work than we really need to do. Most of the time, we don't need to keep the value of the comparison around any longer than just checking it in the if statement. So instead of assigning the comparison to the variable, we'll just put the comparison straight into the if statement. You can see an example here, which works exactly like the code we just saw. Instead of assigning something to *x* and then saying, "if *x*," we now just say: if age <= 12. It's also a lot easier to understand this code since we have the comparison right in the if statement so that it's easy to read. Previously, to understand the if statement, we'd first need to figure out what *x* was supposed to be.

Let's look back at one of our earlier examples, the one where we illustrated nesting. Initially, we just had true and false as the conditions. Imagine that we had the current temperature stored in a variable called currentTemp, and we had a Boolean variable, heaterIsOn, that's true if the heater is currently on and false otherwise. How might we write our condition? Well, for the first one, we might do a comparison, whether currentTemp is below some threshold, say 68°, or not. For the second, we can use the heaterIsOn variable to be our condition.

Let's look at one more example, just a little bit more complicated. Look at the code here for a minute, and see if you can figure out what it's doing. You can see that this code gives information about servicing a car. It first computes how long it's been since the last oil change and tire rotation. If it's been at least 3000 miles since an oil change, it recommends an oil change and possibly also recommends rotating tires. If it's not time for an oil change, it just gives a message saying how long until the next oil change.

Conditionals like these form the heart of many programs, including ones you encounter all the time. Now you've probably visited websites where you enter information, and the Web page checks something for you; maybe it makes sure that none of the required fields are blank, or maybe it checks your credit card number to see if it's valid, or maybe it checks an age or a birth date to see if you are eligible for something. The code underneath all those Web pages is using conditionals just like the ones you've seen here.

Now, what if we wanted to form more complicated conditions? For instance, maybe we want to check whether the number *a* is between 1 and 10. That actually requires two checks: one to see whether it's >= 1 and another to see whether it's <= 10. So what do we do? Look at this code. You'll see it does a check for *a* being between 1 and 10, but it's rather messy. We have to nest two if statements together. If both are true, then we print out that it's in range, but if either one is false, we print out that it's not in range. Since we've nested the if statements, we need to print out "not in range" in both else clauses. This doesn't seem very good, especially since we have a pretty simple condition we're checking for. So it turns out that we can combine conditions using some simple Boolean logic.

The three Boolean operators we'll care about most are and, or, and not. We use the and operator when we want two things to both have to be true in order for an overall expression to be true. For instance, if I say, "My name is John, and the moon is made of green cheese," that would be a false statement overall. Although the first part is true, my name is indeed John, the second part is not, and so the overall statement is false. On the other hand, the or operator is used when you want the statement to be true if either of the two things or both, is true. So if I said, "My name is John, or the moon is made of green cheese," that's a true statement since the first part of it was true. The third operator is not. You can put not in front of any Boolean to switch a false to a true or vice versa. If I was speaking, this not would come in the middle—"My name is not John"—and that's turned the statement into a false one. In programming, though, we just have values, and so the not would come first, so we'd say, "Not my name is John." In code, we just use these

words, all in lowercase, to combine Boolean values together to get new Booleans. We can chain lots of these together, but when we do so, it often helps to use parentheses to distinguish which parts are which.

Let's look again at some simple examples with variables *a*, *b*, and *c*. The first statement combines a false statement that *a* and *b* are equal, with a true one, that *a* and *c* are not equal, using an and. Since an and requires both parts to be true, this overall statement is false. The next statement is an or statement, and it combines two true statements. So it's true, and in fact, it would have still been true even if one part or the other was false like we see in the third example. The fourth line shows the use of not. We've added parentheses to make it clear.

Now take a look at this example, which is quite a bit more complicated. Look at it for a while, and see if you can figure out whether it's true or false. Well, if we look at the values for each of these statements and then combine them one by one, we can determine the answer. The first clause, $a < b$, is true since 1 is less than 2. The next one, $b < c$, is false, since *b* and *c* are equal to each other. Then the last comparison, $b = c$, is true. We can work through to get the answer just like we would for an algebra problem, but the choices here are much simpler. The not false becomes true, and the not true becomes false. The first part is true and true, which is true. And finally, we have true or false, which evaluates to true. So that first statement, complicated as it was, was actually a true statement.

Usually, if you do find yourself with such a complicated comparison, you want to stop and think about whether it's really necessary. Most of the time when I have a really complicated condition like that, I find that I've been thinking about it in a more complex way than is needed, and I can simplify it down to a much more manageable condition. Think about it: Whenever you have a condition, you must indent on the next line. So if you don't combine your conditions with a Boolean, then they have to be nested.

Let's see how this could be used in the example we saw earlier. Remember the check to see if variable *a* was in the range from 1 to 10? We had a set of nested if statements. If you think for a minute about what we've just seen, how could we simplify this so that we have just a single if statement? You

can see here that we can combine the conditions into a single condition using an and: (a>=1) and (a<=10). This makes our if statement much simpler in terms of code, and it's also much simpler in terms of understanding what's happening. We can see the entire condition on one line, rather than having to look at all the lines of nested ifs.

Now before we move on to some other topics, I want to show you one more thing about this statement. First, remember the code we've just been looking at, where we're checking whether the variable is between 1 and 10? Notice that this alternative code does the exact same thing. If *a* < 1 or > 10, then we're out of range, otherwise we're in range. So does this code. Putting a not in there makes it more complicated, though. And even this one. Wow. I mean, this is a really complicated way of writing it. In this case, we check for numbers > 1 and < 10 first, and then we check if it's equal to 1, then if it's equal to 10. Ugh.

Well, my point in showing you four different ways to do the same thing is to make one key point: There can be lots of ways of writing code that does the same thing. Some of that code is simpler and easier to understand, but it all has the same effect. Writing code is not something where there's just one correct solution. The key to writing good code is to understand the range of options available and choose one that's clear and simple. When you write code, don't get bogged down in the idea that there's only one right answer. There are usually many right answers, and the one that's best is often just a matter of judgment.

There's one more topic related to if-then statements that I want to talk with you about. Let's use an example that you might have run into if you buy a movie ticket, or visit an amusement park, or go eat at a buffet restaurant. Often businesses will set different prices, depending on age. Kids less than 3 might be free. There might be one price for kids 12 and under, and another for seniors 65 and older, and another price for everyone else. Let's see how we might set this up in code. If you look at the code here, you'll see we have a few if statements to handle these cases. First, we have the case to check for the youngest kids, then we check for the kids 12 and under, then we check to see if we have a senior; if not, we use the default price. Again, we could have written

this in several different ways, but all of them would've been nested if statements. That's kind of a pain, especially if there are lots of cases. You can end up indenting your code halfway across the screen.

Fortunately, there's an additional command that we can use to handle these cases—the elif statement, which is short for else if. When we have an else statement immediately followed by an if, we can use the elif to combine them together without having to indent further. So let's look at that last example, and how elif can simplify it. Here again is the set of nested if statements that we were just looking at. Now take a look at this code, where we've used elif to handle the exact same cases. The code works the same way; it sets the same prices. All those elif statements are ways of identifying the various cases. And this way of writing it is a whole lot simpler and easier to understand. Any time that you have an else-if combination, it's probably a lot clearer to combine these statements with the elif.

Let's see one more example of how we could simplify with a Boolean. In the casino game craps, a player rolls a pair of dice. If the roll is a 2, 3, or 12, the player loses, and if the result is a 7 or 11, the player wins. If any other number is rolled, that number is called the point, and the game continues. Here is an example of a set of if else-if statements expressing this. How might we simplify this using Booleans? There is more than one way, but here's one option. We can combine the losing options into one check, the winning options into another, and the point options into a third. This is obviously more compact, and it's also easier to understand.

Conditional statements let us write programs where we choose our own adventure. Our choices are based on comparisons and decisions made by if-then-else statements, which we sometimes write in the more compact form of elif statements. All these conditional statements are what allow us to get a huge variety of different outcomes. We can describe those choices by making comparisons between values, and we can make more complicated comparisons and conditionals by using Boolean operators such as and, or, and not. Although there can be many different ways to achieve the same goal, we should try to use the simplest way of writing that we can. These conditional statements can form the basis for

much of the complex behavior that a computer program can exhibit. Any time you see a computer reacting differently depending on your action, there's almost always a conditional somewhere underneath. In the next lecture, we'll be looking at how to take some of these basic ideas we've discussed so far, and put them together to make a larger program, one that'll help us evaluate savings. See you then.

## MORE PYTHON PRACTICE

Here's an exercise to practice the conditional. Imagine that you're writing a routine to help a doctor identify potentially high cholesterol levels. You can assume that there are three variables defined: HDL, LDL, and TRI, which stand for the three components of cholesterol, high-density lipoproteins, low-density lipoproteins, and triglycerides. You can get total cholesterol by summing up LDL, HDL, and one-fifth of triglycerides.

Now the charts that you see show levels of each that are considered good, borderline, and bad. Just to be clear, this is an approximation, not medical advice. You want to output one of three messages. If a person is good in every category, you want to tell the doctor, "Cholesterol looks great." If any one category is bad, you want to tell the doctor, "Warning: Cholesterol looks bad." And otherwise, you want to tell the doctor, "Borderline cholesterol problems." See if you can put together some code that will make this check. Pause, and once you've implemented this yourself, you can continue to see how I did it.

OK, what you see here is one possible way to implement that idea. It's not the only way to do it, so if you came up with something different, it's probably still OK. In our first line, we compute total cholesterol. We then have three checks. The first condition checks whether all four values are in the good range. If so, it outputs the "Cholesterol looks great" line. The elif condition checks whether any one of the values is in the bad range. If so, it prints out the warning. And if neither of these is the case, we get to the else clause, and we print out the borderline message. This is just one example of how a conditional can be used to help sort out and choose from one of several outcomes.

04

# Basic Program Development and Testing

In this lecture, you will learn about some of the big-picture aspects of the programming process. You will learn what's really involved in creating programs, especially as programs become larger and more complex. This is software analysis and development—also called software engineering—which is a critical aspect of computer science. This lecture will focus on one program and the process of developing it. Along the way, you will learn three general principles that are important in practical programming: Plan ahead, keep testing, and develop iteratively.

## [ CREATING A PROGRAM ]

- We're going to create a program to help you save money toward a goal—maybe a new appliance, a vacation, a car, or a house. Then, let's assume that you have a certain amount of money you can set aside each week or month. You want a program that will help you understand how many times you will need to set aside that amount to save up enough money to meet the goal. Mathematically, this is really simple—basically just a division.

- It can be really tempting to just jump in and start writing code, but the first thing we should do when we start programming is to stop and think about our program.

- In this program, we need a division to calculate the number of payments. In order for the program to do that, we first need to get information from the user about how much he or she wants to save and how much he or she is setting aside each period of time. We also need to present the results to the user.

› Generally, you can stop planning at the point when it's obvious how to turn your plans to code. Then, it can be a good idea to start by writing comments to ensure that we know what needs to be done in each part of the program that we'll write. In this case, we can write a comment for each of the three main steps.

```
#get information from user
#calculate number of payments
#present information to user
```

› Next, we'll fill in the actual code for each of these steps. We'll start at the beginning, requesting and reading in some data from the user. We need two pieces of data from each user: the amount to save for and the amount regularly set aside each period of time.

› For this simple version of the program, we won't ask for how long each period of time will be. Instead, we'll just calculate a total number of payments. So, we'll need to ask the user for two pieces of information and then store each of those in a variable. Let's use the variable "balance" to store the total balance and the variable "payment" to store the amount set aside each period.

```
#Get information from user
balance = input("Enter how much you want to save: ")
payment = input("Enter how much you will save each period: ")
#Calculate number of payments that will be needed
#Present information to user
```

› The next thing we should do is test the code we wrote to make sure it works. Regular **testing** is extremely important in software development, and it's only by testing along the way that you're likely to catch errors that might otherwise slip through. Test each logical section of your code; you should never write large amounts of code without stopping to check to make sure it works.

› We want to see if we really did write code that will read in the values we wanted and store them correctly. One of the simplest ways we can test code is to run it and print out the values of variables. So, we'll type in a print statement and run our code.

```
#Get information from user
balance = input("Enter how much you want to save: ")
payment = input("Enter how much you will save each period: ")
print("Balance is", balance, "and payment is", payment)
#Calculate number of payments that will be needed
#Present information to user
```

› When we run the code and enter a few values for balance and payment, such as 100 and 10, we do get the values printed back out. So, it looks like the code we have written is working.

› Next, let's write—and test—our next section of code, the calculation. To determine how many payments we'll need to save up for our goal, we'll just divide the balance by the payment. Let's store that in a variable "num_remaining_payments."

```
#Get information from user
balance = input("Enter how much you want to save: ")
payment = input("Enter how much you will save each period: ")
#Calculate number of payments that will be needed
num_remaining_payments = balance/payment
#Present information to user
```

› To test this, we'll want to print out that variable to make sure we got the right answer. And that happens to be the last of our three tasks: presenting the information to the user. So, we have a program, and we need to test it.

```
#Get information from user
balance = input("Enter how much you want to save: ")
payment = input("Enter how much you will save each period: ")
#Calculate number of payments that will be needed
```

```
num_remaining_payments = balance/payment
#Present information to user
print("You must make", num_remaining_payments, "more payments")
```

- Let's run the code and say that we want to save 100 dollars, and can save 10 dollars, per week. The computer tells us that there is an error in line 9: "unsupported operand type(s) for /: 'str' and 'str.'" This means that we tried to do a division—that's the slash operand—between two strings—that's what the "str" means. The computer thinks that we're trying to divide a string by another string, which can't be done.

- Our division is between the variables balance and payment, so that must mean the computer thinks that both balance and payment are strings. To fix this, we will add float around each of the input lines to make sure we're getting floats rather than strings. Our input is coming in as strings, so we need to convert it to integers or floats if we want numbers.

```
#Get information from user
balance = float(input("Enter how much you want to save: "))
payment = float(input("Enter how much you will save each period:
  "))
print("Balance is", balance, "and payment is", payment)
#Calculate number of payments that will be needed
num_remaining_payments = balance/payment
#Present information to user
print("You must make", num_remaining_payments, "more payments.")
```

- We type in 100 and 10, and we get the right answer. But just because this passed one test doesn't mean it's actually doing what we want it to do. Testing means testing many cases. For example, if we enter 325 and 60, we get an answer that looks right. But when we type in 100 and 0, we get another error—this time a division by 0.

- The problem arises when we're trying to divide by zero. Basically, we want to make sure that the user enters a payment amount that's not zero. This program is assuming that we set the same amount aside each time, so we have to have a positive amount or it won't work.

- We perform a check with a conditional statement—an if-then statement. So, let's put something in the code right after we read in the payment to make sure it's not zero. We'll have an if-then statement, and the condition seems obvious—we want to check whether the payment is equal to zero. But what should we do if it is zero?

- There are a few options. Maybe we want to ask them to put in a new value. Or, maybe we want to say, "That's no good—you can't reach your goal if you set no money aside" and close the program. Or, maybe we want to say, "Zero in is never going to let you save money. What about if we use some small value, such as 1?" We need to think about which functionality we want if someone enters bad information, especially bad information that could crash our program.

- So, we'll write an if statement. For the condition, we check whether payment is equal to zero. Notice that we need the double equal sign.

```
#Get information from user
balance = float(input("Enter how much you want to save: "))
payment = float(input("Enter how much you will save each period:
  "))
if (payment == 0):
    payment = float(input("You can't get something for nothing!
      Enter some nonzero value:"))
#Calculate number of payments that will be needed
num_remaining_payments = balance/payment
#Present information to user
print("You must make", num_remaining_payments, "more payments.")
```

- Then, if the condition "payment is equal to 0" is true, we'll print out a message to the user, saying that you need to save something, so please enter a nonzero value. Notice that in this case, we're getting another input so that we can enter something new.

- So, let's run this to test it. If we enter in 100 and 0, now we get the message, and we get a chance to enter some other value, such as 10. That's a lot better than what we had before.

- After some testing, it seems like regular values are working, and we can handle the cases where we enter 0 because there is a warning. What if we enter a negative number, such as a payment of -10? We get a negative answer.

- The math is correct, but it doesn't make much sense. We don't need to be talking about saving up negative amounts, and saving up negative amounts will never get us to our value. So, we probably should put in another check to make sure we handle the negative cases.

- For the balance, if the user enters a negative number, there could be several reasons. Maybe it's an error on the user's part, or maybe it means that there is already enough money saved—the amount the user still needs is less than zero.

- Assuming it's the second case, we want to treat the balance as zero so that we compute that no more payments are needed. So, in this case, we'll write a statement telling the user what we're doing and just set the balance to zero.

```
#Get information from user
balance = float(input("Enter how much you want to save: "))
if (balance<0):
    print("Looks like you already saved enough!")
    balance = 0
payment = float(input("Enter how much you will save each period:
  "))
if (payment == 0):
    payment = float(input("You can't get something for nothing!
      Enter some nonzero value:"))
#Calculate number of payments that will be needed
num_remaining_payments = balance/payment
#Present information to user
print("You must make", num_remaining_payments, "more payments.")
```

› What about for a negative payment? We already have a check, so let's just modify the check we had previously to also handle negative cases. We need to adjust the condition, so we're excluding payments less than or equal to zero. And we need to adjust the message to say that the user needs a positive number.

```
#Get information from user
balance = float(input("Enter how much you want to save: "))
if (balance<0):
    print("Looks like you already saved enough!")
    balance = 0
payment = float(input("Enter how much you will save each period:
  "))
if (payment <= 0):
    payment = float(input("Savings must be positive. Please enter
      a positive value:"))
#Calculate number of payments that will be needed
num_remaining_payments = balance/payment
#Present information to user
print("You must make", num_remaining_payments, "more payments.")
```

› If we enter a negative payment, we get the message to enter a positive value, and it seems to work.

› If we enter a negative savings goal, we get the message, but then we also get asked for a payment again. That doesn't make sense—we don't need to save anything, so why make a payment? So, let's turn that first statement into an if-then-else statement and do the whole bit about asking for payment information only if needed. That means we need to put in an else and also to indent all the payment part. And because we are still computing balance divided by payment, we need to set some payment value if we did have a negative balance, so we'll set it to 1.

```
#Get information from user
balance = float(input("Enter how much you want to save: "))
if (balance<0):
    print("Looks like you already saved enough!")
```

```
    balance = 0
    payment = 1
else:
    payment = float(input("Enter how much you will save each
      period: "))
    if (payment <= 0):
        payment = float(input("Savings must be positive. Please
          enter a positive value:"))
#Calculate number of payments that will be needed
num_remaining_payments = balance/payment
#Present information to user
print("You must make", num_remaining_payments, "more payments.")
```

› Now, if we run this and enter a negative balance, it goes right to the end. And if we run it again with reasonable values, we still get the right messages when we have a positive balance. Everything seems to be working.

## [ ITERATIVE DEVELOPMENT ]

› You might look at this program and see additional areas that could be improved. For example, the program could interact more with the user. Maybe we want to ask for the user's name and use that in the output, or ask what the user is saving for and use that. Maybe we want to compute the number of payments differently, such that we want to round up to the next-highest integer. We could make all kinds of further modifications to the program to improve it.

› When we make improvements, we are often adding additional features onto already-working code. The process is called "iterative" because it consists of a series of iterations, each one adding a little bit more to the previous round. But the key is that we don't move on to the next iteration until the previous one is working.

- In our program example, we didn't move on to trying to handle the trickier cases with negative numbers and zero until we first made sure the basic cases were working.

- You always want to make sure that the code you have is stable, working code before you try to add on something else. Repeated testing will help ensure that you always have a stable base where you can return, without needing to reexamine everything.

## Readings

Gries, *Practical Programming*, chap. 15.

Zelle, *Python Programming*, chap. 2.

## Exercises

Try making additional iterations to improve the code developed in the lecture. The following are a few possible improvements you might want to make.

1. Assume that the user has already saved up some amount of money. Ask the user for an amount already saved. (Note that the balance will be the cost minus the amount already saved.)

2. Ask the user for the period (week, month, etc.) for how often they will regularly save money. Use this to make the input and output for the user more meaningful.

TRANSCRIPT 04

# Basic Program Development and Testing

In programming, it's often true that we can't see the forest for the trees. We may look at individual lines of code in a program and think, "Hey, that looks easy," but fail to really see how the entire program is working. Or we might dive right in to writing our own program and get so caught up in the details that we can't keep that bigger picture in mind. Either way, there's a big difference between looking at code line by line and seeing how those individual statements work versus seeing how all the details come together to make a complete piece of software.

So in this lecture, I want to pull back a little bit and take a look at some of the bigger-picture aspects of the programming process. I'm going to explain what's really involved in creating programs, especially as our programs get larger and more complex. This takes us into software analysis and development—what's also called software engineering. All of this is a critical aspect of computer science as a whole.

We're going to spend the entire time in this lecture focused on just one program and the process of actually putting it together. I want us to focus directly on what it's actually like to develop a program. Along the way, we'll talk about three general principles that are important in practical programming: one, plan ahead; two, keep on testing; and three, develop iteratively, in what I like to call a pyramid style.

To see all of this in action, we're going to create a program to help you save money toward a goal—maybe a new appliance, maybe a vacation, or a car, or a house, or a second house. Then let's assume that you have a certain amount of money that you can set aside each week or month. You'll want a program that can help you understand exactly how many times you'll need to set that money aside in order to save up enough money to reach your goal. Well mathematically, this is really simple—basically just a division. Now I have deliberately picked a very simple

example so that we can focus on fundamental challenges and pitfalls involved whenever we turn ideas into working code.

Let's go ahead and start programming. It can be really tempting—and I mean really, really tempting—to just jump in and start writing code. But like many things in life, you'll often be better served by thinking about what you're going to do before doing it. So the first thing that we should do when we start programming is not actually write code. The first thing to do is to stop and think about our program. If you were wanting to build a house, the first thing you'd do would not be to start mixing concrete; you'd first want to sit down and make sure you had the design plans worked out and could afford what you intended to build. When it comes to coding, an ounce of preventive planning is worth a pound of cure.

So in this program, what is it that we really want to do? Well, we realized earlier that we need a division to calculate the number of payments. But in order for the program to do that, we first need to get information from the user about how much they want to save and about how much they're setting aside each period of time. And what else? Well, we need to be able to present the results to the user. And that's probably enough to start with for now. Again, this is a simple example, but notice, in addition to the division operation, we identified two additional steps—one for input and one for output—that our program will have to do in order to work.

And as programs get more complex, the value of planning ahead becomes more and more important. When professionals develop software, they will often spend days or weeks or sometimes even months thinking about and planning their code before actually writing it. There are a wide variety of ways that people can actually do this planning, and the process has changed a lot over time. Quite a ways back, people would often use flowcharts to describe plans for code, and they're sometimes still used today. But nowadays, there are a much wider variety of approaches used in software design, with the outcome being design documents, UML diagrams, etc. Now I don't want to focus on any one particular way of doing this planning. What's far more important than any specific tool is that before you sit down and start typing out code, you first think about and plan out your approach.

Now for our little program, we've already got a brief outline of what we want to do, those three main steps. For a problem of this size, that's plenty of planning, so we're just going to go ahead and start coding this up. Generally, you can stop planning at the point when it's obvious how to turn your plans into code. Now it can be a good idea to start by writing comments to ensure that we know what needs to be done in each part of the program that we'll write. In this case, we can take our three main steps and write a comment for each of the main steps: get information from the user, calculate the number of payments, and present information to the user. We'll next fill in the actual code for each of these steps.

We'll start at the beginning—requesting and reading in some data from the user. We need two pieces of data from each user: the amount to save for and the amount that they're regularly going to set aside each period of time. For this simple version of the program, we won't ask for how long each period of time will be; instead, we'll just calculate a total number of payments. So we'll need to ask the user for two pieces of information and then store each of those in a variable. Let's use the variable balance to store the total balance and the variable payment to store the amount set aside each period.

OK, so we should have code that will read in pieces of information. What's the next thing that we should do? If you answered something like, "Do the division to calculate the number of payments," that's actually not quite right. Don't just focus on the code itself; keep your eye on the overall process for developing the code. The next thing we should do is test the code that we wrote, to make sure it works. Regular testing is so important in software development, and it's only by testing along the way that you're likely to catch errors that may otherwise slip through.

Let me tell you a story. In December of 1998, NASA launched the Mars Climate Orbiter. Its mission was to travel to Mars, where it would go into orbit around Mars, observe its climate, and communicate that information back to us on Earth. The total cost of the mission was over half a billion dollars. After about a nine-month trip through space, the orbiter finally made it to Mars. But instead of going into orbit and performing its job, contact with the orbiter was suddenly lost, and it

would never be recovered. Now from what we can tell, it burned up in the atmosphere of Mars. The mission was a catastrophic failure, and we never got the vital data that the orbiter was supposed to collect.

Well with such a devastating loss at a cost of half a billion dollars, NASA was very interested in determining what caused the problem. Their investigation managed to determine exactly what the problem was, which is pretty amazing when you think about it. It turned out that there was one piece of software that was writing data using English units, pounds, while the software on the orbiter itself was expecting data to be measured in metric units, newtons. This error had been occurring throughout the trip, but in the way they had set up the equations, the differences between pounds and newtons were small enough that the differences went completely unnoticed until that critical point when the orbiter was ready to go into orbit, and the whole mission went kaput.

Now spaceflight might be the most challenging task that humans have ever faced, so it's not surprising that there will be errors along the way. And you could blame this error on several different things, ranging from communication problems to process planning. But the end result is that a very, very small problem in a piece of software was at least partially responsible for the loss of hundreds of millions of dollars of work.

And that leads me back to the importance of regular testing. After all, everyone encounters bugs in their code, not just new or bad programmers. Even the world's greatest programmers will have bugs in their code. Bugs occur when we make small mistakes, maybe in typing something wrong, or maybe in thinking about something wrong, or maybe in just not realizing all the consequences of some piece of code. When we write code, we never think we have a bug; we think it's all working correctly. The only way we're ever going to find these bugs is to run tests that help us know that they're there.

Now it's tough to blame the NASA programmers for not catching the bug that led to the Mars Climate Orbiter loss; like I said, this is a really complicated problem. I'm sure they had many tests of various parts of their software; but the fact is if there had been tests for the exact way

that data ended up being created and used, they probably would have caught this problem and then avoided the disaster that occurred.

Now I remember one specific bug in some code I wrote that one of my students spent weeks trying to deal with. I thought I had tested that code thoroughly, and in fact, I had done a lot of testing on it. But I had still somehow missed testing one whole section of code, and as a result, the student spent weeks trying to find and fix the problem. But it showed how thorough testing, as code is being written, can actually save lots of time later on.

Now there are some people who will say you should test after every line of code that's written. After all, every time you write a line of code, you've potentially introduced an error. And more tests over a wider range of situations are never going to make a program less reliable. Now realistically, line-by-line testing is a little too extreme, but it makes a good point: You can almost never have too much testing. My rule of thumb is: Test each logical section of your code. We should never write large amounts of code without stopping to check to make sure that it works.

Here we've written a grand total of just two lines, but it's still a good point to test; we've written the first logical section of our code. We want to see if we really did write code that will read in the values we wanted and store them correctly. One of the simplest ways we can test code is to run it and print out the values of variables. So we'll type in a print statement, like you see here, and run our code. Now when we run the code and enter a couple of values for balance and payment, like 100 and 10, we do get the values printed back out. So all of this looks pretty good so far. It looks like the code we've written is working.

Now I should note that it can be kind of frustrating to have to stop the fun work of writing new code just to perform tests on something you expect to be working already; however, it's important to remember that the more code you write from the point you introduced a bug, the more trouble it'll be to find and fix the bug later on—and sometimes it will be too late fix at all, like in the case of the Mars orbiter. So remember: Your

life will be easier, your code better, and ultimately more fun, if you are in the habit of testing more frequently.

So let's write and test our next section of code—the calculation. To determine how many payments we'll need to save up for our goal, we'll just divide the balance by the payment. Let's store that in a variable num_remaining_payments. To test this, we'll want to print out that variable to make sure that we got the right answer. And that happens to be the last of our three tasks—presenting the information to the user. So we've actually got a program, and we need to go ahead and test it. So let's run the code and say we're wanting to save $100, and we can save $10 a week. Oh no. What happened? We've got some error here, and if we read it, it says that there is an error in line 9, and that name "balace" is not defined. Huh? Oh, wait. I see. I just mistyped balance; I accidentally left out the *n*. OK, that's easy to fix.

OK, so let's try running that again. We run it. We type in 100 and 10—and ah, another error. What is wrong with this stupid computer? OK, let's see what the error says. It says line 9. That's the same one we just fixed. So why is it giving me another error for something I just fixed? OK, I guess we better see the specifics; it says, "Unsupported operand types for /: 'str; and 'str.'" Now that might not be very comprehensible to you at first, but what this means is that we tried to do a division—that's the / operand—between two strings—that's what the str means. So the computer thinks that we're trying to divide a string by a string, which doesn't make sense; there's no such thing as division between two strings.

OK, our division is between the variables balance and payment, so that must mean that the computer thinks that both balance and payment are strings. Doh. I know, I forgot to convert them when I first read them in. I even missed it when I ran my first test to see if those lines were working. OK, this should be an easy fix. I'll just add float around each of the input lines to make sure we're getting floats rather than strings. Remember, our input is coming in as strings, so we need to convert it to ints or floats if we want numbers. OK, so I'm reading in variables as floats now, and then I'm calculating the division. So that looks better. Let's try this again.

We run, type in 100 and 10, and hey, it's working. I got the right answer. So maybe I'm done?

Of course, I'm not done. Just because this passed one test doesn't mean it's actually doing what I want it to. Testing means testing lots of cases. So we have to try to run a few more tests. OK, if I enter 325 and 60, we get an answer that looks right, even if it's got a lot of decimal places and truncates at the last decimal place. Good so far. Let's try another test. Oh, I know. Let's try entering a payment of zero and see what happens. Ah, another error, this time, a division by zero. And that kind of makes sense. If we're not setting aside any money, we'll never save up any. But having the program give us an error and crash doesn't seem so great, so let's think about what we can do to fix that.

The problem comes up when we're trying to divide by zero, so basically, we want to make sure that the user enters a payment amount that's not zero. This program is assuming that we set the same amount aside each time, so we have to have a positive amount, or it won't work. How do we perform a check? A conditional statement—an if-then statement. So let's put something in the code right after we read in the payment to make sure it's not zero. We'll have an if-then statement, and the condition seems obvious; we want to check whether the payment is equal to zero. But what should we do if it is zero? Well, there are a few options. Maybe we want to ask the user to put in a new value. Or maybe we want to say, "That's no good. You can't reach your goal if you set no money aside," and just close the program. Or maybe we want to say, "Zero in is never going to let you save money. How about if we use some small value, like one?" We need to think about what ounce of prevention functionality we really want to include if someone enters bad information, especially bad information that could crash our program.

Let's say that we go with the first option for now. So we'll write an if statement. For the condition, we check whether the payment is equal to zero. Notice that we need the double equal sign. Then if the condition payment == 0 is true, we'll print out a message to the user, saying, "You need to save something, so please enter a non-zero value." Notice that in this case, we're getting another input, so we can enter something

new. OK, let's run this to test it. If we enter 100 and 0, now we get the message, and we get a chance to enter some other value, like 10. That's a whole lot better than what we had before.

What if we run it again, and despite the warning the user decides to enter zero again? Well in that case things still fail. There's a way we can keep asking the user for a value until we get one that's good, and when we learn about loops we'll see how to do this, but for now, we'll just hope that this is enough to handle our problems. We've given the user a warning, so hopefully the user will read the directions.

Let's do another test. Let's try entering zero for the balance. Well, that one seems to work OK. It says we need no payments, which makes sense. Let's try one more test. OK, it seems like regular values are working OK; we can handle the cases where we enter zero. What if we enter a negative number, like a payment of –10? It looks like we get a negative answer. The math is correct here, but it doesn't make a lot of sense. We don't need to be talking about saving up negative amounts, and saving up negative dollars will never get us to our value. So we probably ought to put in another check just to make sure that we handle the negative cases correctly.

For the balance, if the user enters a negative number, there could be several reasons. Maybe it's an error on their part, or maybe it means there's already enough money saved—the amount they need is less than zero. We'll assume that it's the second case here, and so we want to treat the balance as zero so that we compute that no more payments are needed. So in this case, we'll write a statement telling the user what we're doing, and just set the balance to zero.

OK, how about handling negative payments? We've already got a check, so let's just modify the check we had previously to also handle negative cases. We need to adjust the condition, so we're excluding payments less than or equal to zero. And we need to adjust the message, to say that they need a positive number. OK, so let's test our new code. So if we enter a negative payment, we get the message to enter a positive value, and it seems to work. Now if we run it again, and we enter a

negative savings goal, we get the message, but then we also get asked for a payment again. That doesn't make a whole lot of sense; we don't need to save anything, so why make a payment? So let's turn that first statement into an if-then-else statement, and do the whole bit about asking for payment information only if needed. That means we need to put in an else, and also to indent all the payment part. And there's one more thing. Since we are still computing balance divided by payment, we need to set some payment value if we're going to have a negative balance, so we'll just set it to one. OK, now if we run this, and we enter a negative balance, it goes right to the end. Great. And if we run this again with reasonable values, we still get the right messages when we have a positive balance. So it doesn't seem like we broke anything. Everything seems to be working.

Now you might look at this program and see additional areas that could be improved. What improvements would you make first? Well here are some I see: The program could interact more with the user. Maybe we want to ask for the user's name, and use that in the output, or ask what they're saving for and use that. Maybe we want to compute the number of payments differently so that we round up to the next highest integer. We could make all sorts of further modifications to the program to improve it.

This does bring up the other big idea that I want you to learn from this lecture, and it's one you've probably already seen in practice. It's called iterative development. When we make improvements, we're often adding additional features onto already working code. The process is called iterative since it consists of a series of iterations, each one adding a little bit more to the previous round; but the key is that we don't move on to the next iteration until the previous one is working. In our program example here, we didn't move on to trying to handle the trickier cases with negative numbers and zero until we first made sure the basic cases were working.

Let me give you one of my favorite analogies for building software. Two very different but very strong architectural structures are arches and pyramids. When software development goes well, the whole process

is like building a pyramid. First, you lay out a base of stones, then you add a second layer of stones, then another layer on top of that, and so on. With the pyramid, you add and adjust each new layer one by one, getting stones into a stable position before moving on to a new layer. At any given point, the structure you've already built is very stable. You could stop part-way-through, and you would still have a solid, stable structure. In terms of code development, we want to build code using a pyramid style whenever we can—that is, we want to build up new code on top of other code that we know is working. Each iteration, we can add one more layer to the pyramid, building a new stable layer on top of an existing stable structure.

Now, unfortunately, many times, people will develop code like they're building an arch. An arch only holds its shape if every brick in the arch is there. So these programmers will write lots of code, and finally, they reach that one glorious moment when they've written the last line of code that they think they need, and then they run the code, and they find that the whole thing fails. There was some piece was missing along the way, or that wasn't working, and as a result, the entire program has problems. Unfortunately, there's no solid base to go back to, and so they have no definite idea where to look for the problem. They may have to start all over. So you always want to make sure that the code you have is stable, working code before you try to add on something else. Repeated testing will help ensure that you always have a stable base where you can return, without needing to re-examine everything.

Now let's try making some improvements to the code we have. We have a solid, stable base to work from, so let's build, pyramid-style, on top of our previous code, to create a more complex program. Let's say that we'd like to have a friendlier interface—for instance, let's say that we want to print a message about what the program is for, and get the user's name and what's being saved for, and use that information in our messages. How might you modify the program to do this?

Well, here's one solution; it's not the only one though. Notice that we start out by printing a message saying that the program is to help you determine how long you'll need to save. We then ask the user to

enter their name and what they're saving for. Next, when we ask for the balance, we can actually include the user's name and the name of the item; it makes the message a little bit friendlier. Again, at the very end, when we print out the message about how many payments are needed, we can put the name of the item in there. So if we test this out and run it, I can enter my name and that I'm saving for a car. Notice that the message directly asks, "OK, John. What is the cost of the car?" And after I enter the cost and the payment amount, I'll get another personalized message at the end.

Again we now have a stable system. We could choose to add something else onto it at this point. Let's say that we wanted to handle the case where a user entered a negative payment a second time, even after a warning. How would you write some code that would detect that that had happened, and do something different, like set the payment value to one? Here's one way to do that. We can add a nested if statement. We'll check again to see if the new value entered is again negative. If so, we'll scold the user a little, and tell them that we're going to just assume the payment is one and set the value accordingly. For the sake of time, I'll skip testing this right now, but that would be the next thing to do.

OK, so once we've tested and reached a new stable state, we can do another iteration and improve it again. Let's say that we wanted to make sure the number of payments is always an integer, rounding up if needed. The number of payments is a floating-point number. If we convert a floating-point number to an integer, it will remove the fractional part, and just give us the whole portion. So one option would be to convert the number of payments to an integer and add one if we had to drop a fraction. Can you figure out what the code would look like to do that?

Again, here's one possible solution; it's not the only one. We convert num_remaining_payments to an integer and see if it's actually smaller than the floating-point number; if so, that means that there was some fractional part that was lot. If so, we add one. So now, if we run this program, and I say I want a ticket that costs $100, and I'm saving $75 a pop, the program will tell me I need to make 2 payments to get the

ticket. Or if I had entered $25 savings, I'd be told 4 payments would be needed. So the program seems to be working for both the case where I need to round up and the one where I don't.

At this point, we've added several layers of improvement on our program, making sure that we reached a stable state after each one. We already have a much more complex program overall. If you had tried to write this program all at once, no telling how many errors you might have had along the way, and how much trouble you'd have had fixing it.

So there are three big takeaways from this lesson. First, as you develop code, it helps to plan ahead for what you're going to do before you actually write the code. Second, the more testing, the better. Everyone—and I mean everyone—is going to make mistakes when they code, and you're only going to find these by testing your code. And finally, you want to make sure that you're building your code incrementally—pyramid-style not arch-style—so that you always have a working base to go back to.

As we look at code in later lectures, we won't usually take the time to show the development process quite to this level—to show each line being typed and testing each section. You'll mostly see me presenting a bunch of code that looks clean and runs correctly. But I want you to remember that the actual process of development is very different from seeing a bunch of finished code that looks clean and runs correctly. Clean code is always the result of many repeated runs and tests. You're going to make mistakes, deal with bugs and errors, and have to test your code repeatedly. This is a totally normal and unavoidable part of writing code; everyone encounters these challenges. The secret to handling these challenges is to start with an ounce of planning, test until you have a stable base of code, and then iterate through the code to make additional improvements. If you stick with this approach, you'll be able to make the computer do some incredible stuff, and feel great that you were able to organize things so precisely.

05

# Loops and Iterations

We will often find ourselves wanting to do the same thing repeatedly, and we can describe this behavior by loops. In this lecture, you will learn about while loops and for loops. The for loop is the choice whenever you have a well-defined set of items to go through or a clear number of times to run the loop. If you're uncertain of how many times you'll need to repeat a loop but can clearly define when you'll be done, use a while loop.

## [ WHILE LOOPS ]

› With **loops**, we take our first big step toward avoiding repetition. The most basic loop structure is the **while loop**, which is the same thing as an if-then statement that is repeated over and over. In fact, when we write a while loop, we write it just like an if-then statement. All we do is replace the word "if" with the word "while."

› An if statement starts with "if," then has a condition that can be true or false, then a colon, and then a body that is indented. The indented part of the code is the stuff that happens if the condition is true.

› This code checks whether some value is zero or negative, and if it is, it asks a person to input a positive value instead.

```
if value <= 0:
    value = int(input ("Enter a Positive value!"))
```

› What happens if we run this, though, and the person again enters a number that's not positive? The code just goes on, accepting that wrong entry. We would have to add another check after the first one, and then check that one, and so on. If someone were stubborn enough, he or she could keep entering negative values, and we'd eventually run out of if statements.

› But there's a really easy way to fix all this: with our while loop. With the while loop, we have the exact same structure we had before, but now we change the word "if" to be the word "while." The rest of the statement is the same. We have a condition, followed by a colon, followed by the code we want to execute, indented.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))
```

› The main difference is that with the while statement, we're going to keep doing the stuff in the body, as long as the condition is true, without needing to type in all those if statements. While sets up a loop that is like an indefinitely long string of ifs.

› When we come to this while statement, we'll first check the condition, just like we did before with the if statement. If the condition is true, then we do the stuff in the body, and if not, we skip it.

› What if we have a negative value, such as -1? When our code reaches this line, we find that the value is indeed less than or equal to zero, so the condition is true. So, we run the indented line, where we ask for a positive value and get a new value from the user.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))

MEMORY:
value: -1

WINDOW:
Enter a Positive value!
```

› But let's say that the user is stubborn, or ignoring the command, and enters another negative number, such as -5. The new number entered updates the value in memory.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))

MEMORY:
value: -5

WINDOW:
Enter a Positive value!-5
```

› With the while statement, once we finish the body of the code, we go back and check the condition again. In this case, the value is still negative, so we run the body of the loop again. The user could—yet again—not enter a positive value.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))

MEMORY:
value: 0

WINDOW:
Enter a Positive value!-5
Enter a Positive value!0
```

› When we check the condition, it's still true, so we do the loop again. Our program can continue this indefinitely, until the user finally enters a positive value. Now when we check our condition, it's false, so we skip the loop and go on to the code after the loop.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))

MEMORY:
value: 1
```

```
WINDOW:
Enter a Positive value!-5
Enter a Positive value!0
Enter a Positive value!1
```

› Let's look at a slightly modified version of this loop. Just like with the if statement, we can have multiple lines of code in the body of the loop, the indented part. In this case, let's say that we have two lines: an input and a print.

› We come along to the while statement and have a negative value, so the condition is true. Because of that, we ask the user for input.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))
    print("You entered", value)

MEMORY:
value: -1

WINDOW:
```

› Let's say that the user enters the value 3, a positive value. The condition for the loop is no longer true, but we still have one line in the body. So, we go on to the next line of the body, which prints the value that we put in just now.

```
while value <= 0:
    value = int(input ("Enter a Positive value!"))
    print("You entered", value)

MEMORY:
value: 3

WINDOW:
Enter a Positive value! 3
You entered 3
```

- Notice that we only check the condition again after we have gone through the whole body. Once we've started running the body of the loop, we don't check our condition again until the body of the loop is finished. It doesn't matter if the condition of the loop becomes false somewhere in the middle of the loop—we only care about whether it's true or false at the end.

- In the following loop, we set the value to zero and output the data.

```
while value <= 0:
    value = 0
    print ("The value is:", value)

WINDOW:
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
The value is: 0
...
```

- Notice that, in this case, the condition is always going to be true. Inside the loop, we set the value to zero, so every time we check the condition, value <= 0, it's going to be true. If we run this code, assuming that the value wasn't positive to begin with, it's going to just write "The value is zero" repeatedly. We can never get out of the loop. We refer to this as an **infinite loop**, and it's something that we usually want to avoid. But it's actually a pretty common bug.

› Because of this, you want to make sure that you know how to stop a program that's in an infinite loop. If you're running your code in the PyCharm or another **integrated development environment**, you'll probably have some sort of stop button, usually designated by a red square. Pressing this will stop the loop. Other times, you'll need to stop the program another way, such as by closing the window it's running in or entering some command that will cause the program to quit.

## [ FOR LOOPS ]

› Let's say that we want to find the ages of all the people in a group. We can count the number of people in the group, and then we can go through a loop that same number of times.

```
num_people = int(input("How many people are there? "))
i = 0
total_age = 0.0
while (i < num_people):
    age = float(input("Enter the age of person" +str(i+1)+ ": "))
    total_age = total_age + age
    i = i+1
average_age = total_age / num_people
print("The average age was", average_age)
```

› In the first line, we find out how many people there are in the group by asking the user. The next lines set up our counter for the number of people, *i* (because we're iterating, or counting through, an index), and the sum of the ages in the group. We're going to add all the ages and then divide by the number of people.

› The next line we encounter begins a loop. We go through the loop one time per person. Inside the loop, we're going to ask for the age of the person and add that age to the total.

- Notice that the message we print out contains the person number. In the input statement, notice that we're asking for person *i*+1, because otherwise we'd be asking for person 0 first, which would be confusing.

- Finally, after the loop is over, we compute the average age by dividing the total by the number of people, and then output it.

- If we run this program, enter in 3 people and the ages 10, 30, and 20, we find that the average age is 20.

- This type of loop, where we increment some value by one on every **iteration** (meaning one time through a loop), is really common. Because it's so common, there is a Python command specifically built to accommodate this type of loop—called a **for loop**.

- In terms of how the commands are written, the for loop is just a simpler way of writing a particular version of a while loop. However, in practice, these two types of loops are used very differently. A while loop is used when we're not sure how many times we'll need to iterate through the loop; a for loop is used when we have a precise set of things to loop through, or know exactly how many times to iterate.

- We've just been looking at loops like the following one. We have some variable, which we can call an **iterator**, that gets initialized to some starting value, gets incremented every time through the loop, and gets checked until it reaches some maximum value. In this example, the iterator *i* is initialized to 0, and then will take on the values 0 through *n*-1 as we go through the loop.

```
i = 0
while i < n:
    ...
    i = i+1
```

- We can write this same loop using a for loop, as follows. These two loops do the exact same thing. The iterator *i* is still going to take on the values 0 through *n*-1 as we go through the loop.

```
i = 0
while i < n:
    ...
    i = i+1
for i in range(n):
    ...
```

- We start with the word “for.” We then give the iterator variable we want to use, which is *i* in this case. Then, we have the word “in.” And then we have the range, followed by a colon.

- For example, if we have “for i in range 4,” the values 0 through 3 will be printed out.

```
for i in range(4):
    print(i)

OUTPUT
0
1
2
3
```

- The range command actually gives us a little more control. Let’s look back at our while statement. There are three things that we can vary in the statement: the starting value, which is 0 in this case; the number that we are comparing to, which is *n*; and the amount we increment by every step, which in this case is 1.

- We have control over all three of these things in the range statement. We can specify the starting value, the value we don’t want to meet or exceed, and the amount we increase by on each iteration.

- If we list only one value in the parentheses, it’s assumed that we start at 0 and increment by 1, not exceeding the value that’s in the parentheses. If we list two values in the parentheses, it’s assumed that we increment by 1, starting from the first value and not

exceeding the second value. Finally, if there are three numbers, they give all three pieces of information: starting value, limit, and increment amount.

› For loops can get more complicated, and you can see some of the maybe unexpected behavior if you try putting in negative numbers. For example, the following case counts down from 5 and stops once we're no longer greater than 1. Notice that Python automatically tells that you're counting down when it sees the negative value in the third spot of the range.

```
for i in range (5, 1, -1):
    print (i)

OUTPUT
5
4
3
2
```

› This for loop is the same as the following one. A key feature is that it repeats based on a greater-than comparison, instead of a less-than comparison.

```
i = 5
while i > 1
    print (i)
    i = i - 1
```

## Readings

Gries, *Practical Programming*, chap. 9.

Matthes, *Python Crash Course*, chap. 7.

Zelle, *Python Programming*, chap. 8.

## Exercises

What would be the output of the following code?

1
```
i = 10
while i > 1:
    print (i)
    i /= 2
```

2
```
i = 0
value = 0
while value < 20:
    value += i
    i += 1
    print(value)
```

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

4
```
for i in range(3,5):
    print (i)
```

5
```
for i in range (1,10,3):
    print (i)
```

6
```
for i in range (1, 10, -3):
    print (i)
```

7
```
for i in range (10, 1, -3):
    print (i)
```

Write code to do each of the following.

**8** Get a number from the user, and then count from 1 to that number. Try writing it using both a while loop and a for loop.

**9** Convert the following while loop into a for loop.

```
i = 2
while(i<7):
    print(i)
    i = i + 3
```

**10** Write a short program that defines a number from 1 to 10, and then keeps asking the user to guess that number until the correct number is guessed.

05 TRANSCRIPT

# Loops and Iterations

Tired of the same boring routine, day after day? Wishing there was a way you could avoid doing the same tasks again and again? Introducing loops, the amazing new tool that takes away the pain of repeated tasks. You'll be amazed as loops take away the drudgery of your programming and free up your time to work on more interesting stuff. Here's how it works: You describe the task you want to do—just one time. You get the loop started, and you tell it when to stop. And Presto! The loop keeps doing your work for you, as often as you want; it can go on forever. But wait! Act now, and you'll get a second loop, letting you choose exactly how many times you want to repeat the same task. That's two loops! So let loops take over for you all those boring tasks you've been doing, over and over.

Yes, in programming, laziness is a virtue. For programmers, being lazy means that you don't want to do tedious or repetitive work. So, you look for a simpler, easier approach in which you avoid repetition. With loops, we take our first big step toward not repeating ourselves. And the most basic loop structure is the while loop. A while loop is the same thing as an if-then statement that's repeated over and over. In fact, when we write a while loop, we write it just like an if-then statement. All we do is replace the word "if" with the word "while".

An if statement starts with "if," then has a conditional, which can be true or false, then a colon, and then a body that's indented in. The indented part of that code is the stuff that happens if the condition was true. So, our code here checks whether some value is zero or negative, and if it is, it asks a person to input a positive value instead. But what happens if we run this through and the person, again, enters a number that's not positive? Well, the code just goes on, accepting that wrong entry. We would have to add in another check after the first one, and then check that one, and so on. If someone was stubborn enough, they could keep entering negative values, and we'd eventually run out of if statements.

But there's a really easy way to fix all of this, and that's with our while loop. With the while loop, we have the exact same structure as before, but now we've just changed that word "if" to a "while". All the rest of the statement is the same. We have a condition, followed by a colon, followed by the code we want to execute, indented in. And the main difference here is that with the while statement, we're going to keep doing the stuff in the body as long as the condition is true, without needing to type in all of those if statements. While sets up a loop that is like an indefinitely long string of if's.

Let's see how it works for the example we've been looking at. When we come to this while statement, we'll first check the condition—just like we did before with the if statement. If the condition is true, we do the stuff in the body; and if not, we skip it. So, what if we have a negative value, such as −1? When our code reaches this line, we find that the value is indeed less than or equal to zero, so the condition is true. So, we run the indented line where we ask for a positive value and get a new value from the user. But say the user is stubborn, or is just ignoring the command, and enters another negative number, like −5. The new number entered updates the value in memory.

With the while statement, once we finish the body of the code, we go back and check the condition again. In this case, the value is still negative, so we run the body of the loop again. The user could, yet again, not enter a positive value. When we check the condition, it's still true, so we do the loop again. Our program can continue this indefinitely until the user finally gets the idea and enters a positive value. Now, when we check our condition—finally, when they've entered a positive value—it's false, and so we skip the loop and go on to the code after the loop.

Let's look at a slightly modified version of this loop. Just like with the if statement, we can have multiple lines of code in the body of the loop—the indented part. In this case, let's say we have two lines—an input and a print. Now, let's see what happens here. We come along to the while statement, and have a negative value, so the condition is true. Because of that, we ask the user for input. Say the user enters the value 3, a positive

value. What do you think happens next? Notice that the condition for the loop is no longer true, but we still have one line left in the body.

Well, the right answer is that we continue with the body. We go on to the next line of the body, which prints the value that we put in just now. Notice, we only check the condition again after we've gone through the whole body. Once we've started running the body of the loop, we don't check our condition again until the body of the loop is finished. It doesn't matter if the condition of the loop becomes false somewhere in the middle of the loop; we only care about whether it's true or false at the end.

Let's look at another example. For once, this is one I do not want you to type on your own into your computer, at least not just yet. For now, just think about it. In this loop, we set the value to zero, and output the data. What will happen here? Notice that in this case, the condition is always going to be true. Inside the loop, we set the value to zero, and so every time we check the condition—value less than or equal to zero—it's going to be true. If we run this code, assuming the value wasn't positive to begin with, it's going to just write the value is zero over and over and over. We can never get out of the loop. We refer to this as an infinite loop, and it's something that we usually want to avoid, but it's actually a pretty common bug. As you keep writing code, I guarantee that you'll accidentally create an infinite loop at some point.

Because of this, you want to make sure that you know how to stop a program that's in an infinite loop. If you're running your code in the PyCharm or another integrated development environment, you'll probably have some sort of stop button; it's usually designated by a red square. Pressing this will stop the loop. Other times, you'll need to stop the program another way, such as by closing the window it's running in, or entering some command that will cause the program to quit. On Windows machine, that's often Control-C. On a Mac, it might be Control-Command-C.

OK, now would be a good time for you to try entering this on your own. You can enter your own simple loop, like the one you see here, that would cause the computer to keep on sending me the same message forever. That's right—if you ever feel you need a pick-me-up message,

you can get your computer to give you unlimited positive reinforcement. Once you've entered this program and tried running it, you can practice stopping the code.

Let's look to see whether or not this next example will cause an infinite loop. Here's one loop. The loop's condition is that value is greater than zero. The first time we get to the loop, value is 3, so obviously, the body of the loop will be reached. In the body, we have a couple of print statements; but in between, we first set the value to −3—a value that's negative. Since the loop condition is that the value is positive, does this end the loop right away?

No, the loop will continue. Remember that we only check the condition at the end of the body. So, later in the body, we set the value to 7. It's once again greater than zero. So, when we get to the end of the body, does the body of the loop run again, or do we exit?

Well, the body will run again. When we reach the end of the body, we'll check the condition. And since the value at that time is greater than zero, the condition is true, and so we run the body again. In fact, since the value will always be greater than zero at the end of the loop, we'll always run the body again—that is, we're in an infinite loop.

Of course, we normally want to avoid infinite loops, so let's take a look at a more typical loop. Suppose we have a variable *a* that we initialize to zero. We then have a loop that repeats as long as *a* is less than 3. Within the loop, we print out *a*, and increment it by one. So, what do you think the output here will be?

OK, let's trace through this example. We start out by assigning zero to variable *a*. We then reach the while statement, and we need to examine the condition. In this case, we check whether *a* is less than 3, which is true, so we go into the body. The first thing we do in the body is print out the value of *a*, or 0 in this case. The next line of the body increases the value of *a* by one. Since it was 0, now it's 1. At this point, we check our condition again; 1 is indeed less than 3, so we'll go through the body of the loop again. We print out *a*, and increment *a* by one, again. Now the

value of *a* is 2. Again, we check our condition, which is still true, since 2 is less than 3. We print out the value of *a*, and increment by one again. Now, when we check the value of *a* the condition is false; *a* is equal to 3, and so it's no longer less than 3, so we are done with the loop. Notice that we've had values ranging from 0–2. After the loop is over, the value is equal to 3. By the way, one time through a loop is called an iteration.

Here's one more example. Take a look at this code. What do you think would be printed by running this code? In this case, we'll print the values from 10 down to 1. When we start the loop, our value 10 is greater than 0, so we execute the body. That will print 10, and then lower the value to 9. Since the value is still above 0, we do the loop again; so we print 9, and then lower the value to 8. This continues until we print 1 and lower the value to zero. Since the value is no longer greater than zero, we'll stop, and not go through the loop again.

There are many times when we program that we want to go over a range of numbers, so this technique of increasing or decreasing a value by one on every iteration is actually really common. It might be that we're counting something, so we want to increase the number by a fixed amount each time. Let's say that we want to find the ages of all the people in a group, for instance. We can count the number of people in the group, and then we can go through a loop that same number of times. Let's take a look at this example and see if you can follow what's happening, and then we'll trace through it together.

In the first line, we find out how many people there are in the group by asking the user. The next lines set up our counter for the number of people, *i*, and the sum of the ages in the group, which we initialized as zero. We're going to add up all the ages and then divide by the number of people.

Now, it's good to pick variable names that make sense–something that tells you what that variable is doing. Calling our variable *i* might look like an exception, but actually, this does make sense; you're counting through integers. And after all, *i* often appears just like this in mathematics, designating that some sort of iteration or indexing is

going on. So whenever you're counting through a loop like this, it's common practice to call a variable *i*. It just means that you're iterating or counting through an index. In fact, if there need to be nested loops, where there's one counting loop inside of another, then it's common to use *j* for counting through the inner loop.

OK, back to our code. The next line that we encounter begins a loop. You'll see that the loop's a lot like the one we saw earlier, where we go through the loop one time per person. Inside the loop, we're going to ask for the age of the person and add that age to the total. Notice that the message we print out has the person number in there. You might want to look at that input statement for a minute to understand what it's producing, or type it in and try it out yourself. Notice that we're asking for person *i* plus 1, since, otherwise, we'd be asking for person zero first–which will be confusing to most users. OK, finally, after the loop is over, we compute the average age by dividing the total by the number of people, and then output it. Let's get a quick run-through. If we run this program; enter in 3 people; and the ages 10, 30, and 20, we find the average age is 20.

This type of loop where we increment some value by one on every iteration is really common. In fact, because it's so common, there's actually a Python command specifically built to accommodate this type of loop; it's called a for loop. In terms of how the commands are written, the for loop is just a simpler way of writing a particular version of while loop. However, in practice, these two types of loops are used very differently. A while loop is used when we're not sure how many times we're going to need to iterate through the loop. A for loop is used when we have a precise set of things to loop through, or we know exactly how many times we want to iterate.

We've just been looking at loops like the one you see here. We have some variable which we can call an iterator that gets initialized to some starting value; it gets incremented every time through the loop, and it gets checked until it reaches some maximum value. In this example, our iterator *i* is initialized to zero, and then we take on the values 0 through n–1 as we go through the loop.

We can write this same loop using a for loop, like you see here. These two loops do the exact same thing. The iterator *i* is still going to take on the values 0 through n–1 as we go through the loop. We start with the word "for". We then give the iterator variable we want to use, which is *i* in this case. Then we have the word "in". And then we have the range, followed by a colon. There are actually things we can put here at the end besides range, but we'll see those later. For instance, if we have "for *i* in range 4," the values 0 through 3 will be printed out.

Now, the range command actually gives us a little more control. Let's look back at our while statement. There are actually three things we can vary in the statement. First, there's the starting value, which is zero in this case. Then, there's the number that we're comparing to, which is *n*. And finally, there's the amount we increment by every step, which in this case is one. We actually have control over all three of these things in the range statement. We can specify the starting value, the value we don't want to meet or exceed, and the amount we want to increase by on each iteration.

If we list only one value in the parentheses, it's assumed that we start at zero and increment by one, not exceeding the value that's in the parentheses. If we list two values in the parentheses, it's assumed that we increment by one; starting from the first value, and not exceeding the second value. And finally, if there are three numbers, they give the three pieces of information: starting value, limit, and increment amount.

So, look at this example. What should this be printing? This should print out the numbers from 0–6. We started at 0, checked on each iteration whether or not we were less than 7, and incremented by one. Now look at this example. What would it output? In this case, we start at 1. We go up by two each time. And, we only do this as long as we're less than 7. So, we would output 1, 3 and 5. I'd really suggest taking some time right now to test this out yourself. This is a really simple loop—just two lines. Try lots of variations until you understand what's going on.

For loops can get more complicated, and you can see some of the maybe unexpected behavior if you try putting in negative numbers. Really, go

ahead and put some negative numbers in, and see what happens. For example, look at this case, and see if you can figure out what it does.

This actually counts down from 5 and stops once we're no longer greater than 1. Notice that Python automatically tells that you're counting down, when it sees the negative value in the third spot of the range. This for loop is the same as the one you see here. A key difference is that it repeats based on a greater-than comparison, instead of the less-than comparison. Again, let me encourage you to try this out on your own before going forward.

Let's look at a couple more issues related to loops. First, let's go all the way back to where we started this discussion—the if statement. Here's the if statement that we were looking at before. Remember that we turned it into a loop by replacing the "if" with a "while." Well, we've also seen how an if statement can have an else clause, like you see here. We understand how this works for if statements. Well, guess what? We can do the same thing for while statements. Again, the only change is that the word "if" changes to "while". And what happens now is that we check the conditional, and if true, we do the body of the loop, and then check the condition again. And this will continue as long as the condition is true.

But, whenever the condition becomes false, we then execute the else part of the statement. That's true whether it's the first time we check the condition or the thousandth time—once it's false, we do the else part. After the else part, we're all done, though; we don't go back through the loop any more. So, the code that we see here will keep asking for a value until we get something greater than zero. Once we do, whether that's the value when we start out or that's the value we end up at, we then output a statement—"You entered" some value.

If you remember our if statements, we could put an if inside of another if—we called this nesting. Just like we had nested if statements, we can also have nested loops, where one loop is inside of another.

Here's an example for printing out the times tables. First, we have a loop using *i* to count from 0–9. Now, for each value of *i*, we'll go through

another loop, this one using *j* to count from 0–9. For each combination of *i* and *j*, we'll output one line, saying *i* times *j* equals the product of *i* and *j*. So, this loop will print out a multiplication table for all the numbers from 0–9. Let's say that you wanted to modify the code, to make the times tables run from 1–10 instead of 0–9. How might you do that? Well, one of the simplest ways is to change the range to be 1–11 instead of 10. That will ensure that each loop begins at 1, and goes on as long as it's less than 11, instead of as long as it's less than 10.

A different way to achieve the same goal would be to leave the loops themselves unchanged, but change the values that are printed, basically replacing *i* with *i* plus 1, and *j* with *j* plus 1. So, although the loops would go from 0–9, the printout would show 1–10.

Let's look at one more example. Imagine that you have a number of people, and you want to print out a list of every possible pair of people. You only want each pair once, though, so pair A-B is the same as B-A. Do you see why this is a good problem for nested loops? You have an outer loop for the A member of each pair, and a second inner loop to get the remaining members available for the B member of each pair. Let's assume we've read in the number of people into a variable numpeople. How might you use nested loops to print out the different pairs, where a pair has the number of the first person and the number of the second?

Here's one way of creating these pairs using nested loops. First, we have our outer loop, in which *i* ranges from 1 to numpeople. Notice that the range to use is 1 comma numpeople plus 1. That ensures we start at 1, and we include numpeople. Our inner loop needs to start with the person after person *i*. Notice that person *i* will have already been paired up with the people with an earlier number when that person was the one in the outer loop. So, we only need to pair person *i* with the people with a number greater than *i*. Our inner loop, therefore, starts at *i* plus 1 and goes again all the way through numpeople plus 1.

Let me mention a couple of optional commands that you can add to loops that'll sometimes make your code more efficient and look a whole lot cleaner. The first of these optional commands is called the continue

statement. Sometimes you have a complicated loop with a lot of stuff going on inside.

In this example, all I want you to notice is that it's going through a loop, and it's got an if statement inside that's trying to get us to skip certain dates. Basically, the loop is doing something, but when day is a multiple of day to skip, it skips parts of the loop. Now, looking at this code, there's nothing wrong with it, but it's got a couple of aspects that aren't ideal. For one, there's a lot of indentation. Indenting is fine when needed, but unnecessary indentation can make it harder to read the code. And second, the indented code isn't that different from the other code. It's not as though it's part of a inner loop or some rare branch of the code—in fact, it's the more typical branch. So, it might be nice to have an alternative way to write this.

All we do is indent in a continue statement just below the first conditional, which we also change from a not-equal-to to an equal-to. This code does the same thing as the previous code. What the continue statement does is tell the computer I'm done with this iteration of the loop, let's skip to the end. It's a way of skipping everything else in the body of the loop that you're in, and going back to the beginning. In this case, we can put in a check to see if we are on a day we want to skip, and if so, we just go on to the next iteration of the loop; we skip all that other stuff. So in short, the continue statement offers a nice alternative for writing a loop, allowing you to skip through an iteration of a loop without adding in another layer of indentation for a large chunk of code.

One other optional loop command that can make your life easier is the break command. Break is like a more extreme version of continue. Break says I'm done with this loop for good; get me out of here entirely. When you encounter the break statement, you immediately exit the loop—you even skip the else part of the loop, if there is one.

In this example, we're looping through a set of numbers printing them out. And if we get through all of them, we print a message saying we got through all the numbers. But, if we encounter the "bad number" along the way, we print out a message and exit the loop entirely. Notice

that the for statement starts at 1 and increments by twos as long as the iterator is less than 10. If we were to look at the output of this loop, we'd print out 1, then, 3, then 5; then when we'd hit 7, and we'd just output the error message and stop. Notice that we don't continue, and we don't execute the else clause of the loop.

On the other hand, we could have a different loop where we start at 2, instead of at 1. In this case, we'd get through all numbers, never hitting the number 7; and eventually, we'd print out the got through all numbers message from the else clause.

A break command is a good way of getting out of a loop when you might otherwise encounter a problem, or when you've reached a point that you know you're all done. Again, both continue and break commands are optional; you never need one. But, they can be useful tools to clean up your loops—both for loops and while loops—and avoid spending more time in them than is needed. Keep in mind that these will only break or continue from one loop. If you have nested loops, they'll only let you break out of, or continue within, the innermost loop that the statement's a part of.

So, think of loops whenever we need to get a bunch of input, or generate a lot of output, or just do any action over and over again. The for loop is the choice whenever we have a well-defined set of items to go through, or a clear number of times to run through the loop. If we're a little uncertain of how many times we'll need to repeat a loop, but can clearly define when we'll be done, that's the time for a while loop. If we know we'll be taking 1000 steps, we should be using a for loop; while if we know we're just taking steps until we reach our destination, that's the time for a while loop. Loops save us lots of time, but an even bigger advance is never having to re-type our data, or re-type entire programs at all. And this brings us to the very useful topic we'll explore next time—that's the subject of files.

## MORE PYTHON PRACTICE

Let's use a loop to explore an interesting mathematical problem. The Collatz conjecture, or 3N + 1 conjecture, says that given some number, there's a pair of manipulations that can be repeated until it eventually will reach 1. The manipulations are given by two rules, depending on whether the number is odd or even. If the number is even, you should just divide it by 2 to get the next number. If the number is odd, you multiply it by 3 and add 1 to get the next number in the sequence.

Go ahead and try to write code that will generate this sequence for a given input number, and tell you how many steps it took to reach 1. First, read in the number from the user, and then generate the sequence for that number.

Here is some code to do this. We'll start by reading in the starting number, *n*, and we initialize the count of the number of steps to zero. Then, we have a loop. The loop will continue as long as our number *n* is not 1. Inside the loop, we increase our count of the number of steps, and then we generate the next number in the sequence. We determine whether the current number is even or odd by using what's called the modulus operation, for which we use the percentage sign. And this operation will give you the remainder when the first number is divided by the second. So, if we take a number modulo 2, we'll get either a 0 or a 1. If the answer is 0, it must have been even; and if the answer is 1, it must have been odd. Now, depending on whether the result was even or odd, we generate the next number in the sequence by either dividing by 2 or multiplying by 3 and adding 1. Either way, we print out the result. And the loop will end when the sequence finally gets to 1, and, at that point, we can print out how many steps it took. That's all there is to it.

06

# Files and Strings

In this lecture, you will learn more about the process of how a program can interact with files sitting in storage. Whether the data file is something that already exists before you run the program or it's something you will create from within the program, you need to form a link between the program and the file sitting in storage. Files are closely tied with strings, because the typical file format that you will write to and read from will essentially be one long string. The locations of those files are also given by strings, so it's important to know how to work with strings.

## [ OPENING AND CLOSING FILES ]

› There are basically three things we do with data files in our programs.

  ◊ First, we have to make a connection with the file. We call this opening the file. We have to tell the computer that there is this thing outside our program that we're going to be using inside our program for input or for output.

  ◊ Second, once a file is opened, we will eventually start working with it—reading data from it or writing data to it.

  ◊ Finally, once we're done, we'll close the file: The program has to say that we're done with the file. This is going to break the connection that we made when we opened the file. Closing the file makes sure that everything in the file is left in a nice condition so that nothing is corrupted. Also, it prevents us from accidentally opening too many files at the same time.

- To open a file, we can use the command Python helpfully calls "open." The following is what an open command looks like.

```
myfile = open("Filename", "r")
```

- First, we need a name for the file. This is the name that we're going to use for the file inside our program. It's the name that we're going to use to refer to whatever that file is when we write our code, and it's a variable that we can name like any other.

- The name we use inside our program doesn't have to have any relation to the name of the file in the computer itself. It's just our internal way of thinking about the file. Our internal name for a file is a variable whose name makes sense within the context of our program. When we open a file, we'll make a particular connection between that internal variable we use in our program and the specific actual file stored outside the program.

- In this case, the name of the variable we'll use is "myfile." We then need to assign an actual value to that variable, so we have the assignment operator, the equal sign.

- Next comes our open statement. Notice that it has parentheses right after it. Inside the parentheses are two strings.

- The first of these strings is the name of the file in the computer system. This is the name of the file that you would see if you looked at a file explorer or directory on your computer. If you create a file and save it, this is the name you used to save it. In this case, the name of the file is "Filename."

- The final part of the if statement is a second string, and this string tells the computer how we're planning to use the file. If the string has the letter *r* in it, it means that we are going to read from the file—basically, it's going to be giving us input.

- If it has the letter *w* in it, it means that we are going to write to the file—basically, it's where we can put our output. And if it has the letter *a* in it, it means that we are going to **append** to the file. That means that we are going to write to it, but we're not writing from scratch; we are going to just add on more stuff to the end of an existing file.

- If you try to open a file for reading and the file doesn't already exist, you're going to get an error. Obviously, it can't form a link to read in something from a file that doesn't exist. If you open a file for writing or appending, it'll create the file for you. If you open a file for writing and the file already exists, you will write over the old version. So, be careful when you write to files.

- Opening a file is our way of creating a connection, so to break our connection, we are going to need to close our files. The following is the command you need to close a file.

```
myfile.close()
```

- You start out with the name for the file variable. This is your internal name that you've been using—in the example, it's "myfile." Then, you add a period, the word "close," and two parentheses.

- When we deal with files, we have some code like the following. First, we open the file, then we do some something, and then we close the file.

```
myfile = open("Filename", "w")
#Do something here
myfile.close()
```

- But there's another way we can write this code in Python. It makes it easier to know when you have the file open and when you don't, and it helps ensure that your file always gets closed.

```
########## OPTION 1
myfile = open("Filename", "w")
#Do something here
myfile.close()

########## OPTION 2
with open("Filename", "w") as myfile:
    #Do something here
```

- These two sets of code do the same thing. In the second one, we have a command, the one starting with "with." This opens the file, just like in the first line of the first set of code. Then, the stuff you want to do with the code is indented, just like with conditionals and loops. For all that indented code, we can use the opened file, named "myfile" in this case. When we leave the indented portion, the file will be closed for us automatically.

- Either way you do this is okay, but an advantage to this second version is that you won't ever forget to close your file, because it's done for you automatically. However, the file is only open for the section of code that is indented.

- A downside to this approach is that if you have multiple files open at once, there could be a lot of indenting. It tends to work better if you have just one file open for a while that you are then going to close. If you will have many open files, then it's probably better to open and close them on your own.

## [ READING FROM AND WRITING TO FILES ]

- In between opening and closing files, we can do any of the normal code that we always have, but in addition, we can read from or write to the file, depending on how we opened it.

- Writing works like the print statement that we've been using. The following is an example.

```
myfile = open("Filename", "w")
myfile.write("This line is written to the file.")
myfile.close()
```

- We start out with the name of the file we're writing to. This is the internal name we gave to the file. In this case, it's "myfile." Next, we have a period, followed by the keyword "write," followed by parentheses. Notice that this is like the close command. Finally, inside the parentheses, we have a string.

- This is like the print command, but there are some important differences. The write command can only write strings. It cannot output numbers, but it can convert numbers if needed. Also, the write command can only write one string; you can't put in separate strings spaced by commas, the way you could with a print statement.

- Finally, the write command does not put in a newline character at the end of each line you write. With print, every time we print something out, it comes out on a new line. With write, that's not the case, and if we want there to be a new line, we need to explicitly write out a newline character.

- Let's turn to reading, instead of writing.

```
myfile = open("Filename", "r")
linefromfile = myfile.readline()
myfile.close()
```

- First, notice that we opened the file for reading at the beginning. When we read from a file, we're generally going to read in one line at a time. That line is going to come to us as a string; we will get a string that's one line from the file. In other words, when we have the "readline" command, we read an entire line from the file as one single string. That string will include the newline character at the very end of the line as part of the string.

› We need to assign that string to some variable, so our command to read in from the file will start with a variable and an assignment operation. In this case, the variable name is “linefromfile.” To get that line, we start with the name of the file, which in this case is “myfile.” Then, we have a period, then the keyword “readline,” followed by parentheses. Notice how similar this is to the close command.

## [ ORGANIZING AND ACCESSING FILES ]

› Once we know how to open and close files and how to read lines from them or write strings to them, we need to learn how files are organized into directories and how our programs can access those files, wherever they might be.

› We know how to open a file by specifying the file’s name inside the parentheses, but it’s not just the name of the file that can be specified there. We can actually specify a string that contains both a **path** and the filename. The path gives the directory in which the file resides, also called the “folder” that the file belongs in. If no path is given, it’s assumed that the file is in the same directory as the file for Python itself.

› The directory structure is the way all the files on the computer are organized, in a large hierarchy. Some of these details will vary depending on which operating system you’re using. On Windows machines, the base directory is designated by some letter followed by a colon. “C:” is the most common base directory. To specify a position, you specify each subdirectory using a backslash character. On Macs, running the OS X operating system, a forward slash is used to separate the directories.

› The Python commands can usually be accessed from /usr/bin/local.

› You can also specify file locations relative to the current file’s directory. To specify a relative path, the key thing to remember is that the “..” directory is the directory one level higher in the hierarchy. So, a path such as ..\..\Programming would mean going up two levels in the hierarchy and then down into the “Programming” directory.

## Reading

Gries, *Practical Programming*, chap. 10.

Zelle, *Python Programming*, chap. 5.

## Exercises

Write code to do the following.

1. Open a file named "data.txt" in the current directory for reading.

2. Open a file named "data.txt" in the directory above the current one for writing.

3. Close the files in exercises 1 and 2.

4. Given an open file "infile," read and print each line in the file.

5. Ask a user for a filename, and then write the numbers 1 to 10, one per line, to that file.

6. Assume that you have a data file named "data.txt" that consists of integers, one per line. Find and print the average of those numbers. (Hint: You will want to keep a running total of the sum of numbers and how many numbers you've read in.)

TRANSCRIPT 06

# Files and Strings

Storage has always been a key component of computing; however, the earliest computers didn't have the ability to store their own files. In 1801, Joseph-Marie Jacquard developed a loom for which the weaving patterns could be programmed. The program, in this case, was instructions for how to set the loom to achieve a particular pattern. The necessary information was stored on a series of paper cards with holes punched in them. When Charles Babbage, in the 1820s and 1830s, designed what's now considered the first general-purpose computer, his design used this method of storing data on punched paper cards.

And, this method of storing computer data on so-called punch cards persisted for many, many decades, and was still a popular means of storing data as recently as the 1970s. A program would consist of a stack of punched cards, where each statement got its own card, and any data might be on a separate stack of cards. Although the computer wouldn't remember a program for you, you could hang onto those stacks of cards—your files—and feed those cards into the computer again in the future, each time you ran the program.

Clearly, we're much better off these days being able to use digital files, but the notion of maintaining our data long-term so that it can be retrieved and reused still remains. Our files today sit in digital storage, whether inside the computer or at a remote location; and in order to be used, these files have to be brought into main memory. When we run a program, we're bringing that program into main memory and executing it. Executing means that we let the processor actually process all the commands given. And, when we read in data from a file—for example, loading a document into a word processor—we're transferring the data in that file to main memory, where it can be used by our program.

In this lecture, we're going to learn more about the process for how our program can interact with those files sitting in storage. Whether our data

file is something that already exists before we run our program, or it's something we'll create from within the program. In either case, we need to form a link between our program and this file sitting off somewhere in storage. Files are closely tied with strings since the typical file format that we'll write to and read from will essentially be one long string. The locations of those files are also given by strings. So, we'll also look more closely at how we work with strings.

There are basically three things we need to do with data files in our programs. First, we have to make a connection with the file. We call this opening the file. We have to tell the computer that there's this thing outside of our program that we're going to be using inside our program for input or for output. Second, once a file is opened, we'll eventually start working with it—actually reading data from it, or writing data to it. Finally, once we're done, we'll close the file; the program has to say that we're done with the file. This is going to break the connection that we made when we opened the file. Closing the file makes sure that everything in the file is left in a nice condition so that nothing is corrupted. Also, it prevents us from accidentally opening too many files at the same time.

Let's look first at the open and close operations. To open a file, we can use the command Python helpfully calls open. Here's what an open command looks like: First, we need a name for the file. This is the name that we're going to use for the file inside our program. It's the name that we're going to use to refer to whatever that file is when we write our code; it's a variable that we can name just like any other. Now let me be clear—the name we use inside of our program doesn't have to have any relation to the name of the file in the computer itself; it's just our internal way of thinking about the file. I like to compare names inside the program to job titles we give to people.

For example, the constitution talks about things that the president does, such as the president can veto legislation. That's true regardless of which particular person is president at the time. Now, when we inaugurate a president, we can connect the president with a particular person. We form this link between the office and a particular person,

like Jefferson or Lincoln. But, when we talk about the president's duties and powers, it doesn't matter who the particular individual is.

Similarly, our internal name for a file is a variable whose name makes sense within the context of our program. When we open a file, we'll make a particular connection between that internal variable that we use in our program and the specific actual file stored outside of the program. In this case, the name of the variable we'll use is myfile. We then need to assign an actual value to that variable, so we have the assignment operator—the equal sign. Next comes our open statement. Notice that it has parentheses right after it. Inside the parentheses are two strings.

The first of these strings is the name of the file in the computer system. This is the name of the file that you would see if you looked at a file explorer or directory on your computer. If you create a file of some sort and save it, this is the name you used to save it. In this case, the name of the file is Filename, but it could have been input.txt, or some other name like Document1.dat, or whatever name the file system needs.

OK, the final part of the if statement is a second string, and this string tells the computer how we're planning to use the file. If the string has the letter *r* in it, it means that we're going to read from the file—basically, it's going to be giving us input. If it has the letter *w*, it means we're going to write to the file—basically, it's where we can put our output. And, if it has the letter *a*, it means we're going to append to the file. That means that we're going to write to it, but we're not writing from scratch; we're just going to add on more stuff to the end of an existing file.

You should know that if you try to open a file for reading, and the file doesn't already exist, you're going to get an error. Obviously, it can't form a link to read something in from a file that doesn't exist. If you open a file for writing or appending, it'll create the file for you—if it doesn't already exist. If you open a file for writing, and the file already exists, you'll write over the old version. So, be careful when you write to files.

Let's practice this. Say you have a program, and you know that you'll want to read in data from one file, and write out data to another. You

want to get your input from a file called MyDataFile.txt, and you want to output to a file called results.txt. How would you open those two files?

Well, you would first have to pick an internal variable name that you want your program to use for the two files. In this case, I've chosen infile for the input file and outfile for the output file. Then, I call open on each of them. First, I give the name of the file—either MyDataFile.txt or results.txt—and then I write *r* for the one I'm opening to read from and *w* for the one I'm opening to output to.

OK, so open was our way of creating a connection, and so to break our connection, we're going to need to close our files. The notation here might look a little weird to you at first. As we learn more about Python, it will make more sense; but for now, just know that this is the command you need to close a file. You start out with the name for the file variable. This is your internal name that you've been using—in the example, it's myfile. Then, you add on a period, the word close, and two parentheses. That's it. Yeah, I don't really expect you to understand why you have that notation, right now, but just know that this is the way to close your files. So, when we deal with files, we have some code like what you see here. First, we open the file, then we do some stuff, then we close the file.

It turns out that there's another way we can write this code in Python. It kind of makes it easier to know when you have the file open and when you don't, and helps make sure that your file always gets closed.

The two sets of code you see here do the same thing. In the second one, we have a command—the one starting with "with". What this does is it opens the file, just like in the first line of the first set of code. Then, the stuff that you want to actually do with the code is indented in, just like we saw with our conditionals and loops. For all that indented code, we can use the opened file, named myfile in this case. When we leave the indented portion, the file will be closed for us automatically. Now, either way, you do this is OK, but an advantage to this second version is that you won't ever forget to close your file since it's done for you automatically. However, the file is only open for the section of code that's indented.

Let's look again at the earlier example we had, where we opened two files—one for reading, one for writing. The code to open those files is shown here. First, see if you can remember how we would close them.

We will write "outfile.close()" and "infile.close()." Note that the order we close the files does not need to match the order we opened them—we can close them at any time, and in any order. It would be perfectly fine to reverse the order. Now, what if we wanted to use that other formulation for opening files, where we have a line starting with "with," and the file will be closed automatically. How might that code look?

In this case, we would have two with statements. The first one would say "with open mydatafile.txt, r as infile." Nested inside of that, the second would say "with open results.txt w as outfile." We would have to be indented in from there for our text. Now, you can see that a downside to this approach is that if you have multiple files open at once, there could be a lot of indenting. It tends to work better if you have just one file open for a while that you're then going to close. If you will have lots of open files, then it's probably better to open and close them on your own.

All right, so we've got the ability to open, and the ability to close files. We need to look at what we can do in the middle. We can do any of the normal code that we always have, but in addition, we can read from or write to the file, depending on how we opened it. Let's look at writing, first, since it's easy to do. Writing works a whole lot like the print statement that we've been using.

Here's an example. We start out with the name of the file we're writing to. This is the internal name that we gave to the file. In this case, it's myfile. Next, we have a period, followed by the keyword write, followed by parentheses. Notice that this is a whole lot like the close command that we had before. As we learn more about Python, we're going to see more and more commands that kind of look like this, where there's a period, and then a word and parentheses. Finally, inside those parentheses, we have a string. Unlike the print command that we've used before, we can't output numbers, only strings. Let me say that again: When we use this write command, we can only write out strings.

We could output a variable that's a string like you see here. Or, we could create a string, even converting numbers if we have to. In the example you see here, we form one string by taking a string and concatenating it with another string that we get from converting a number. We have the string "the number of countries is," and then we want to add "196" on to the end. We convert the number 196 to a string by writing str and putting the number inside the parentheses. It's just like what we've done before when we want to convert a string to a number and put int or float around it. In any case, the output has to be a string.

Now, I said this was like the print command, but there are actually some important differences. As we just saw, the write command can only write strings—no numbers or anything like that. Also, the write command can only write one string; you can't put in separate strings separated by commas, the way you could with a print statement. And finally, the write command does not put in a new line character at the end of each line you write. With print, every time we print something out, it comes out on a new line. With write, that's not the case; and if we want there to be a new line, we need to explicitly write out a new line character.

Have you got all that? Let's try another example and see. Say we've opened up a file using the with formulation. We have a couple of variables—volume1 and volume2—that have been computed previously. How would we write out to the file a line the first volume is volume1, and then a second line, the second volume is volume2?

We would have two lines, very similar to each other. They'd be indented under the with statement. Both would start with the name of the file variable—which in this case is outfile—followed by .write, and the in parentheses, the string we want to write. Notice that to include the volume1 and volume2 data, we need to convert those values to strings; and we need to use the addition operator to append them to the text string that we wrote so that it's still all one string inside the parentheses. We also add on a newline character. Remember that the strings we write don't automatically have a new line at the end, the way they do with a print command, so writing the newline character is the way we make sure it's actually a new line.

OK, let's turn to the other side, where we want to read instead of write. First, notice that we opened the file for reading at the beginning. When we read from a file, we're generally going to read in one line at a time. That line is going to come to us as a string. We'll get a string that's one line from the file. Let me emphasize this, since it can get confusing: When we have the readline command, we read an entire line from the file as one single string. That string will include the newline character at the very end of the line as part of the string. We need to assign that string to some variable, and so our command to read in from the file will start with a variable and an assignment operation—the equal sign. In this case, the variable name is linefromfile. To actually get that line, we start with the name of the file, which in this case is myfile. Then, we use the notation similar to what we've seen before—we have a period, then the keyword readline, followed by parentheses. Notice how close this is to the close command.

So, if you look at the code you see here, it's going to first open a file named Filename for reading. Then it reads the first line of that file and puts that string into the variable linefromfile. It then closes the file. If we wanted to read another line, we would have just had to add another readline command to read another line from the file. Let's say you wanted to actually output those lines to another file. Well, the code you see here would do that. We'd open another file for writing. Then, after reading lines from the input file, we'd write them to the output file. I'd recommend you pause to study this, and make sure you understand how it's working, before moving on.

Let's see how this would work. Say you had a file containing some obscure but possibly important piece of information, like a Swiss bank account number, or the airspeed velocity of an unladen swallow. Assume the information is in a file called important.txt. How would you get that numerical value out of the file?

First, we'd have to open up the file for reading. In this case, I called it infile. Then, I read in the first line of the file by calling infile.readline(), and assigning the result to the variable linefromfile. I can then convert that string to a floating point number, giving me the answer. At the end, I should close the file.

Let me mention one more tip here. One of the things you often want to do is process an input file to take all the information in it and do something with it. So, you want to go through all the lines of the file, one by one—you want to take one line, then the next, then the next, and so on. Well, that should sound to you like a loop—and it is. We could write that loop the way you see here. When we try to read a line from the file, but we're at the end of the file—there's no more lines to read—we're going to get an empty string back. So, we can check each time we read a line to make sure it's not the empty line and do whatever we want. Then we read the next line and repeat this loop forever. Notice that we had to read a line first in order to get this to work.

This is actually a really common operation when working with files—to go through the whole thing—and Python actually gives us an easier way to do it. If you look at this second set of code, it works the same way as the first set. But, we don't have to explicitly write readline, and we don't have to explicitly check for the empty string each time. All that is done for us. We just have a new type of for statement. We start with the keyword for. We then state the variable that we want to put each line in, which in this case is linefromfile. Finally, we have in, the name of the file we're reading from, and a colon. Again, this code is doing the same thing we did earlier—we read in one line at a time through the entire file. It's just a cleaner, and easier to understand, way to do this.

Now, I said that when we read a file, we'll read one line at a time. That's the way we will generally process files, but it's not the only thing we can do. In addition to the readline command, or the for-in structure for looping over all lines, Python allows us to read the entire file as one single string. The way to do this is with the read command. If the file is opened as the variable myfile, we call myfile.read() to get the entire contents of the file as a string. We can then assign this string to another variable. Reading the whole file this way is sometimes advantageous, in that all the data from the file is collected in a single string. From this one string, we can break the string up into many smaller strings by dividing it up into lines.

Before going farther, let me answer one question that sometimes comes up. Suppose you've written a program in a word processor and

saved it. Are you now able to write a program that can read that file that you saved from your word processing program? Well, basically, no. Unless you specifically saved the file as a text-only file, it's going to have a bunch of extra stuff in there that you don't expect—having to do with fonts, and margins, and all sorts of other formatting. It'll be saved in a format that's not even human readable. Basically, you do not want to read files unless they're simple text files—what your word processor might call plain text. If you saved from your word processor as a text file, then sure, you can go ahead and read it in using the readline command I've just described.

So, how does your word processor save a document? And how do things like image files and audio files get stored? Well, these are all stored as what are called binary files. Rather than being in human-readable text, they're in a more compact, but less reader-friendly format of ones and zeroes. When a program reads from a binary file, the program will have to read binary information, and then convert that into the data that it can use. When writing to such a file, a program must open a binary file, convert its data into a binary format, and write it.

In Python, it's actually quite easy for us to take our programming commands for opening and closing text files, and adapt them to open and close binary files instead. We just add a *b* onto the end of the string saying how we're opening the file. Instead of *w*, *a*, or *r* for write, append, and read; we'll write wb, ab, or rb for the equivalent opening on a binary file. That part's easy. And if you ever find you need to read and write binary files, there are tools, such as the pickle module, available to make that easier for you, too.

So, at this point, we know how to open and close files; we know how to read lines from them or write strings to them. We need to talk a little bit more about how files are organized into directories, and how our programs can access those files, wherever they might be. We've talked about opening a file by specifying the file's name inside the parentheses. But, it's not just the name of the file that can be specified there. We can actually specify a string that contains both a path and the file name. The path gives the directory in which the file resides, also

called folder, that the file belongs in. If no path is given, it's assumed that the file is in the same directory as the file for the Python code itself.

The directory structure is the way that all the files on the computer are organized, in a large hierarchy. Now, some of these details will vary depending on which operating system you're using. On Windows machines, the base directory is designated by some letter followed by a colon—C: is the most common base directory. To specify a position, you specify each subdirectory using a backslash character. For my computer, for instance, my PyCharm projects are located in C:\Users\John\PyCharmProjects. On Macs, running the OS X operating system, a forward slash is used to separate the directories. The Python commands can usually be accessed from /user/local/bin.

You can also specify file locations relative to the current file's directory. To specify a relative path, the key thing to remember is that the .. directory is the directory one level higher in the hierarchy. So, a path such as ..\..\programming would mean going up two levels in the hierarchy, then down into the Programming directory. OK, let's say that we want to specify a file named data.txt that's in a subdirectory of the current directory called data. On a Mac, we would specify this as data/data.txt. On Windows, we would specify this as data\\data.txt. Why the double backslash?

To understand this, we need to first understand a little bit more about strings and how they work. As we've noted, Python files are written and read by default as strings. So, it's going to be helpful to understand strings not only for specifying paths but also for dealing with the read and write operations. Remember that a string can be specified by using either single quotes or double quotes, surrounding a bunch of text. The two versions are interchangeable, and the choice doesn't affect anything about the way the string is stored internally in the computer.

What if we wanted to use a word like that's in the text, though? Well, if we use double quotes, it's pretty unambiguous—the single quote in the middle is an apostrophe. But, if we used single quotes to denote the string, the computer can't tell if the apostrophe is just part of the string,

or if it's marking the end of the string. We run into the same problem the other way around if we want to specify a string containing quotation marks. If we use single quotes to denote the string, things are OK, but double quotes just confuse the matter. And if the string contains both single and double quotes, there's no easy solution.

So, the way that strings handle cases like this is using what's called an escape character—specifically, the backslash. In a string, if a backslash character is encountered, it means that the next character should be interpreted differently than usual. We've seen one instance of this before: the backslash-n indicates a newline character. The backslash lets us know this is not our ordinary character *n*, but a special one denoting a new line. Similarly, we can specify single quotes or double quotes in a string by preceding them with a backslash.

So, a string denoted by single quotes can just include a slash before every single quotation mark, and a string denoted by double quotes can just include a slash before every double quote. It doesn't hurt to include a backslash before each one.

Besides the single and double quotes and the newline, there are two other useful escape characters. First, the backslash-t denotes a tab. Can you think of what the other escape character might be? It's the double-backslash. Since a backslash is used to mark other characters as escape characters, we need to have an escape character for the backslash itself.

This brings us back to the directory specification strings. If we're in a Windows system, we need to specify a double-backslash every time we write a string, since that's the only way the backslash will be interpreted correctly. Note that this goes only for strings specified in the program. If a user types in a backslash, or single or double quote, the input command will automatically convert the characters to the correct characters in the string representation.

Since we've been on the topic of strings, I want to mention one more string issue. Sometimes, we want to specify a string with multiple new

lines. We've already seen that we can do this by including the newline character, backslash-n, into our string. But, it can be kind of a pain to have to explicitly type in newline characters, when it would be nicer to just see the text written with line breaks already displayed. So, there's an alternative way to express strings, and that's with three quotes in a row. Three quotes in a row will begin a string and let it continue, new lines and all, until the matching three quotes are encountered. When specified this way, all the new lines in the middle become part of the string itself.

The triple-quote string also is also useful for creating multi-line comments. The hashtag comment is the most common. But in cases where a comment might need to span multiple lines, it's acceptable to use a triple-quote multiline string as a comment. Keep in mind that as you work with files, you're bringing in data from external sources that you have little control over. There are a variety of problems that can happen when reading: a file might not be in the right format, or it might not exist in the location you expect. And, it can be easy to cause problems when writing—opening up an existing file to write will wipe out the old file. So stay vigilant. As long as you take care when working with files, huge boosts in productivity are possible.

Writing data into a file is basically a matter of formatting your data into strings. You want to output data in a way that it can be read in easily with the read command. Thus, keep in mind that you'll want to separate your data appropriately so that you can process the data as it's read in, to pull out the necessary information. This means working with strings, which also comes in handy when you need to read or write to a location other than the Python default directory.

There are a variety of other string manipulation routines we can use with files, but to understand how they work, it will help us to see some new ideas about how to structure our data, including the programmer's version of a list. So, in our next lecture, we'll look more closely at tools for working with large amounts of data—especially in lists, but also in other data structures. I'll see you then.

07

# Operations with Lists

We often have large collections of the same type of data. In Python, lists help us keep track of this data in an orderly way. In this lecture, you will learn about the fundamentals of working with lists. In addition, you will learn about other things you can do in Python that make lists even more useful, including appending, indexing, slicing, and making lists of lists. Each of these is part of what makes Python a particularly useful language.

## [ THE FUNDAMENTALS OF LISTS ]

- Lists are one of the programming features that Python supports particularly well. Python makes it very easy to create lists and do all kinds of things with them. If you use lists in other programming languages, you don't have quite the flexibility that you do in Python.

- In much of programming, "array" is the more general term, but in Python, the usual and broader term is "list," with "array" being used to refer only to one specific type of memory-efficient list.

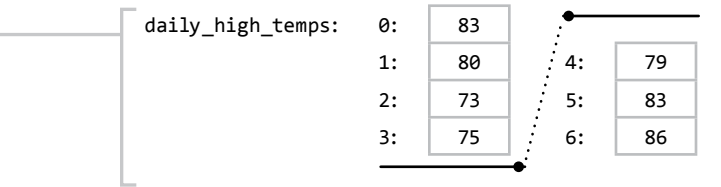
- A variable always corresponds to a box in memory. When we think of a list or array, we have a whole stack of those boxes. We give one name to that whole stack of boxes. This is going to let us organize our data—to keep common things together.

- In Python, we write lists as a series of values separated by commas and enclosed in brackets. The command you see might be storing the daily high temperatures for a week, so we assign the variable "daily_high_temps" a list, specified by square brackets containing 7 values separated by commas.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
```

› In memory, you can think of this as 7 boxes being created, each with a different value contained. Let's say that we want to actually look at one of those values. Each of those boxes in the list is going to have a whole number associated with it—called its **index**.

› The boxes are going to be numbered consecutively. But the first box is numbered 0, not 1. In pretty much all of computer science, we start numbering with 0. So, the first box is 0, the second box is 1, the third is 2, and so on. For the temperatures for the week, we would have the 7 items of the list numbered 0 through 6.




› Each element (or value) in the list is going to have some index. If we want to get the value in one of those boxes—basically, get an element out—we need a way to refer to it. We can refer to an element of a list by putting the index in brackets right after the variable name.

```
variable_name[index]
```

› In the temperature list, printing out the fifth value, "daily_high_temps [4]," would print out 79.

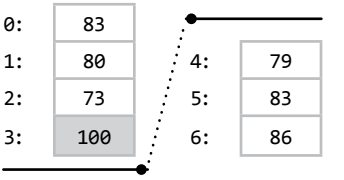
```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
print (daily_high_temps[4])

OUTPUT:
79
```

› The index doesn't have to be a number; it can be a variable, as long as that variable has an integer value. So, we can assign the value of 1 to *i* and then print "daily_high_temps[i]" and get 80.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
i = 1
print (daily_high_temps[i])

OUTPUT:
80
```

› We can also assign values to the list elements, just like any other variable. So, in the following case, writing "daily_high_temps[3] = 100" assigns the value 100 to element 3.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
daily_high_temps[3] = 100

OUTPUT:
daily_high_temps:
```




› What if we wanted to print out all the elements of a list—for example, just this list of 7 temperatures? How might we write code to do that?

› What about using a loop? How would we create a loop to print every element? We can loop through all the different index values and print out the value of each element of the list. The following is one way we might write such a loop.

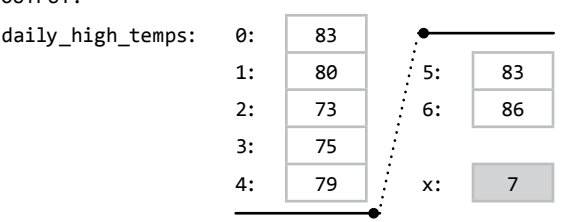
```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
for i in range(7):
    print(daily_high_temps[i])
```

```
OUTPUT:
83
80
73
75
79
83
86
```

› We pick our index—in this case, *i*—and we let it range from 0 to 6. The command "range(7)" means that it will take values starting from 0 on up, as long as it is less than 7. So, in this case, it takes the values 0 through 6. So, for this code, it is going to print out the element for each of those indices, which will print the whole thing.

› Let's say that we didn't know how long the list was. There are a few ways we could figure this out. One way is to first find out the length of the list. Fortunately, there's a command to do this: the "len" command, short for "length." You put the variable in parentheses, and the command returns the number of elements in the list. So, in this case, the length comes out to 7.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
x = len(daily_high_temps)

OUTPUT:
```




› In the previous loop, we had a range of 7. We could replace the "7" with "len(daily_high_temps)," and that would be the same result. Note that this code would work regardless of the size of the list.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
for i in range(len(daily_high_temps)):
    print(daily_high_temps[i])

OUTPUT:
83
80
73
75
79
83
86
```

› There's a second way we can do this. It's a little more complex to understand, but in many cases it's easier to write and more useful. If we want to loop over all the items in a list, we don't even need to have an index, at least not directly.

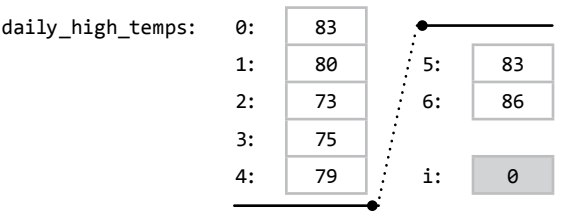
› We can use a for loop, as follows. We don't use a "range" command—we just say for all the *i* in the list. This code is going to print off all the values in the list, just like before.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
for i in daily_high_temps:
    print(i)

OUTPUT:
83
80
73
75
79
83
86
```

› When we set up a for loop like this, the variable is going to take on the values of the individual elements of the list. So, when we start the loop, *i* will get the value of the first element of the list, which is 83 in this case. When we print *i*, we get 83. On the next iteration, *i* gets the value 80, and that's what we print. This will continue until *i* takes on every value in the list.

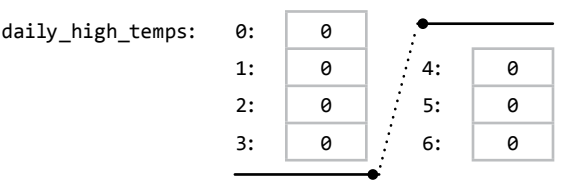
› Note that *i* is getting assigned the value from the list. So, if we modify the value of *i*, it does not change the value in the list.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
for i in daily_high_temps:
    i = 0

OUTPUT:
```




› If we wanted to change the temperature values in the list, we'd need to use a different loop, such as index values. Notice that this code, where we assign a value to the elements themselves, would set all those values to 0.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
for i in range(7):
    daily_high_temps[i]=0

OUTPUT:
```



## [ APPENDING ]

› In addition to creating lists by separating items with commas, we can also merge lists together by using the addition operation. This creates a new list by taking all the elements in the first list and then appending on those from the second list. In this case, we create list 3 by adding lists 1 and 2 together.

```
list1 = [3.1, 1.2, 5.9]
list2 = [3.0, 2.5]
list3 = list1 + list2

OUTPUT:
```

| list1: | 0: | 3.1 |
|---|---|---|
| | 1: | 1.2 |
| | 2: | 5.9 |

| list2: | 0: | 3.0 |
|---|---|---|
| | 1: | 2.5 |

| list3: | 0: | 3.1 |
|---|---|---|
| | 1: | 1.2 |
| | 2: | 5.9 |
| | 3: | 3.0 |
| | 4: | 2.5 |

› We can also increase a list by using the "+=" operation. This will let us append additional items onto the end of an existing list. The following is an example, where we've appended list 2 onto the end of list 1.

```
list1 = [3.1, 1.2, 5.9]
list2 = [3.0, 2.5]
list1 += list2

OUTPUT:
```

| list1: | 0: | 3.1 |
|---|---|---|
| | 1: | 1.2 |
| | 2: | 5.9 |
| | 3: | 3.0 |
| | 4: | 2.5 |

| list2: | 0: | 3.0 |
|---|---|---|
| | 1: | 2.5 |

› If we don't have a list, but rather just one item, there is an "append" command that lets us add one element onto the end of an existing list. In the following example, we have the name of the list, then a period, then the word "append," and then in parentheses the thing we want to append on. In this case, we take list1 and we add on the value 3.9 to the end of it.

```
list1 = [3.1, 1.2, 5.9]
list1.append(3.9)

OUTPUT:
```

| list1: | 0: | 3.1 |
|---|---|---|
| | 1: | 1.2 |
| | 2: | 5.9 |
| | 3: | 3.9 |

› Appending can be really useful for building up lists. There are many times when we want to keep adding things to a list, but we don't know at first how long it's going to be.

## [ INDEXING ]

› In addition to different ways to build up a list, there are also different ways to index into a list. The basic way we index into a list is by putting the element number of the list in brackets. But what happens if we put in an index number that's too large? The array has 7 elements, so they will be numbered 0 through 6. In the following, we're trying to access element 7.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
print (daily_high_temps[7])

OUTPUT:
IndexError: list index out of range
```

› In this case, we get an error, saying that we're out of range. That's actually a good thing—the computer is not letting us access something past the end of the list.

› What will happen if we put in -1 for the index? Instead of getting an error, which is what you might expect, in Python, putting in -1 gives us the value of the last element of the list, 86. This is a convenient tool in Python to let us pull items out of the end of the list.

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
print (daily_high_temps[-1])

OUTPUT:
86
```

## [ SLICING ]

› Python also lets us pull out a portion of a list with an operation called **slicing**. With a slice, we pick where to start and where to stop. So, we don't have just one index value; we have a colon separating two values. You can think of these values as similar to the "range" examples when working with loops.

› The first value, which is *a* in the following code, is the starting point. The second value, which is *b* in this example, is the number that you want to stop before.

```
varname[a:b]
```

› Remembering that the index of the first element in the list is 0, if you want the first three elements, you would enter 0:3, saying that you want elements 0, 1, and 2. If you want elements 4 and 5, you could enter 4:6. If you leave off the number before or after the colon, it means that you want to start from the beginning or go to the end. If you just put a colon, it means from the beginning to end, which is another way of saying that you are getting a copy of the whole list.

```
listvariable[0:3]          #first three elements
listvariable[4:6]          #elements 4 and 5
listvariable[:6]           #first six elements
listvariable[-3:]          #last three elements
listvariable[:]            #copy of list (all elements)
```

- Slicing can let us do some interesting things. We can reassign values to slices, for example, replacing part of a list with another list. We can also insert some new values into the list.

- Strings are actually a slight variation of a list. That means that slicing on strings works basically like slicing on lists. The one big difference is that we can't assign to a string the same way we can with lists. That is, we can't delete parts of strings, or insert strings in the middle, as we can with regular lists. You need a separate set of commands to manipulate strings.

## [ LISTS OF LISTS ]

- We can make a list out of anything. We can have lists of integers, lists of floats, lists of strings, and even lists of lists. The following is a list of three lists, each of which has three elements. What will happen if we print out the first element of this list?

```
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
print (list_of_lists[1])

OUTPUT
[4, 5, 6]
```

- In this case, it prints out element 1 of the list of lists, which is the list "[4, 5, 6]."

- Python lets you create lists in which you have a combination of different types. Most languages require an array to be all stuff of the same type, but in Python, it's okay to have a list of items with an integer, a float, a string, and another list, for example. This is useful when you want to group unlike things together.

## [ TUPLES ]

› Another thing that is very similar to a list that Python supports is the **tuple**, which is like a list whose values can never change and, unlike a list, often will contain different types of data. Tuples can be specified by giving a list of values separated by commas.

```
car_tuple = "Buick", "Century", 23498
make, model, mileage = car_tuple
print(make)
print(model)
print(mileage)
print(car_tuple[1:])

OUTPUT:
Buick
Century
23498
('Century', 23498)
```

› In this example, there is a variety of different stuff put together. In this case, “car_tuple” is a tuple, instead of a list. If we print out a tuple, it shows up as having parentheses around it—not square brackets.

› We can assign values from a tuple to a set of variables separated by commas. In the example, car_tuple has a car make, model, and year, and we can assign the tuple to three separate variables: the make, model, and year.

› We could have done this with lists, but that’s not the best use of lists. We can also access parts of the tuple using indexes or slices, just like with lists. However, we cannot change the value of a tuple once it’s created.

## Readings

Gries, *Practical Programming*, chap. 8.

Matthes, *Python Crash Course*, chaps. 3–4.

Sweigart, *Automate the Boring Stuff with Python*, chap. 4.

## Exercises

```
1   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[1])

2   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[2:5])

3   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[:3])

4   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[8:])

5   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[:])

6   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[-1])

7   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist[-3:])

8   mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    print (mylist)
```

9
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
for i in mylist:
    print (i)
```

10
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
mylist[3] = 100
print (mylist)
```

11
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
for i in mylist:
    i = 0
print (mylist)
```

12
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
mylist.append(100)
print(mylist)
```

13
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
mylist[1:5] = []
print (mylist)
```

14
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
mylist[2:8] = [100, 200]
print (mylist)
```

15
```
mylist = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
mylist [2:2] = [100]
print (mylist)
```

Write code to do the following:

**16** Given a list of integers named “ages,” form a new list named “minor_ages” consisting of all those ages from the “ages” list that are less than 18.

**17** Create a list containing two lists: one with names of 3 people and one with ages of 3 people (you can choose the names/ages).

07 TRANSCRIPT

# Operations with Lists

One of the earliest triumphs of information technology occurred to solve a problem faced by the U.S. census. The U.S. Constitution requires that a census be conducted every 10 years. However, by 1880, the U.S. population had grown so much that the process of tabulating all the data was becoming a problem. In fact, the 1880 census wasn't tabulated until 1887. Well, the census directors realized that the upcoming 1890 census would be an even bigger job, which raised the possibility that it might take more than 10 years just to finish tabulating the data. And that means they wouldn't even be finished with the 1890 census before it was time to take the one in 1900. They knew they needed a different approach.

Well, the Census bureau turned to a former Census employee named Herman Hollerith, who developed for them a tabulating machine to count data entered on cards. His tabulating machine was able to take the census data from 1890 and process it in about a year. Wow. This tabulating machine is considered a predecessor of our modern computers. It was designed to process large sets of very similar information. Hollerith kept on developing improvements to the tabulating machine, and ever since, we've never had to worry about the ability to tabulate the census. Eventually, Hollerith's company and three others merged to form a company named CTR. And CTR later changed its name to one you're probably more familiar with, IBM.

The need to work with large sets of similar data and keep up with it in an organized way was not unique to the census. In fact, there are so many times that we want to deal with data like this, that just about every programming language provides a tool for organizing data like this. In Python, these are lists. Lists are one of the programming features that Python supports particularly well. Python makes it very easy to create lists and do all sorts of things with them. If you use lists in other programming languages, you don't have quite the flexibility that you do in Python.

Let's first see what life is like without lists. Say we want to store a group of credit card purchases. We're going to store the amounts of each purchase so that we can do something with them later, like find the average. So, say our first purchase is $3.50. We can store it in a variable FirstAmount like you see here. OK, let's say we had two more purchases, a $5 one and a $10 one. Well, we would need more variables—SecondAmount and ThirdAmount—to store those. If we wanted to compute the average, we can do so; but you can already tell, this is starting to look pretty ugly. But, what if we had more transactions, like 10? This is quickly becoming unmanageable. And, you can imagine how unworkable this would be if there were 100, 1000, or 1 million of these.

Well, this is where the array, or list, comes in. In much of programming, an "array" is the more general term; but in Python, the usual and broader term is "list," with "array" being used to refer to only one specific type of memory-efficient list. As beginning programmers, though, we don't need to worry that much about memory efficiency, so we'll stick with Python's more general and flexible list.

Now, variables always correspond to a box in memory. So, each of the variables we've looked at so far is one box that can hold one value. Well, when we think of a list or an array, we have a whole stack of these boxes. We give one name to that whole stack of boxes. This is going to let us organize our data, to keep common things together. Let's look at some of the cool things that you can do with lists.

Let's first look at how you can create a list in Python. In Python, we write lists as a series of values, separated by commas, and enclosed in brackets. The command you see here might be storing the daily high temperatures for a week, so we assign the variable high_temps a list, specified by square brackets containing seven values, separated by commas. In memory, you can think of this as seven boxes being created, each with a different value contained.

Now, let's say that we want to actually look at one of those values. Each of those boxes in the list is going to have a whole number associated with it called its index. The boxes are going to be numbered

consecutively. But here's a really important thing to take note of: the first box is numbered 0, not 1. In pretty much all of computer science, you'll find this—we start numbering with 0. So, the first box is 0, the second one is 1, the third is 2, and so on. For our temperatures for the week, we would have the items of the list numbered 0 through 6.

The reason why our first index is 0 is related to how lists are stored in memory. When we assign a list to a variable, what that variable will store is actually the location of the first element of the list in memory. So, to find where an individual element of the list is, we'll add its index to the location of the first element. So, the first element will be right at the position stored—it's offset by 0 from the location stored. The second element is one memory unit after the beginning, and so it will have index 1. And so on.

OK, we've said how each element or value in the list is going to have some index. If we want to get the value in one of those boxes—basically get an element out—we need a way to refer to it. We can refer to an element of a list by putting the index in brackets right after the variable name. Say we had our temperature list from before, and we wanted to print out the first value. We could print out daily_high_temps [0]. By the way, that's how I'll often refer to an array element, by saying "sub." It's shorthand for the word "subscript," which is the mathematical way that we'd usually represent an array element. In this case, we would print out the zeroth element of the list—that is, the first element, or 83. Printing out daily_high_temps [4] would print out 79. So, remember, the variable name is referring to the entire list. Putting the index in brackets allows us to get one element out of that list.

The index doesn't even have to be a number, it can be a variable, as long as that variable has an integer value. So, we can assign the value of 1 to *i* and then print daily_high_temps [i] and get 80. We can also assign values to the list elements, just like any other variable. So, in the case you see here, writing daily_high_temps [3] = 100, that assigns the value 100 to element 3, which is the fourth element. Let's practice this a bit. Let's say we want to create a list of how many days are in each month of the year. First, how would we create this list?

Here's one way: We create a variable, I chose to call it days in month; and then in square brackets, we list the number of days in each month. So, we list the number of days per month: 31 for January, 28 for February, and so on. Now, what if we wanted to print out the number of days in the first month; how would we do that? We would print out days_in_month [0]. Remember that we would have to use the subscript zero. If we wanted the number of days in September, which is the 9th month, we'd need to print days_in_month [8].

Finally, let's allow our list to adjust to account for a leap year. Let's say that we know it's a leap year. What command would we give to update the list appropriately? We'd set days_in_month [1] to the value 29. Or, we can be more flexible, and create a new variable named year, and use this to determine whether or not to make this leap-year adjustment. For this, we can put in a conditional, to check whether the year is a multiple of four—I'll ignore the special cases for leap years when the year is a multiple of 100. So, if it is a multiple of four, we'll make the adjustment to days_in_month [1].

Now, let's think for a minute. What if we wanted to print out all the elements of a list, say just this list of seven temperatures? How might you write code to do that? Well, what about using a loop? How would we create a loop to print every element? We can loop through all the different index values and then print out the value of each element of the list. Here's one way we might write a loop like this. We pick our index, in this case, *i*, and we let it range from 0–6. Remember that the command range(7) means that it will take values starting from 0 on up as long as it is less than 7. So, in this case, it takes the values 0–6. See? You've already seen the idea of counting by starting at zero. So, for the code we see here, it's going to print out the element for each of those indices, which will print the whole thing.

Now, say we didn't know how long this list was. How could we do this? Well, there are actually a couple of ways. One way we can do it is we can first find out the length of the list. Fortunately, there's a command to do this. The len command—short for length—is a way to get the length of a list. You put the variable in the parentheses, and the command returns

the number of elements in the list. So, in this case, we get that length, and it comes out to 7. So, remember our previous loop, where we had a range of 7? We could replace the 7 by len, and then in parentheses daily_high_temps. And that would be the same result. Notice that this code would work regardless of the size of the list—the list could have 1 element or 1000, and it would all work just fine.

There's a second way we can do this. It's a little more complex to understand, but in many cases, it's easier to write, and it's more useful. If we want to loop over all the items in a list, we don't even have to use an index, at least not directly. We can use a for loop, like this. That's right; we don't use a range command, we just say for all the *i* in the list. Now, what do you think is going to be printed here? This code is going to print off all the values in the list, just like before. Is that what you expected? Let's look at what's happening here. When we set up a for loop like this, the variable is going to take on the values of the individual elements of the list. So, when we start the loop, *i* will get the value of the first element of the list—which is 83, in this case. When we print *i*, we get 83. On the next iteration of the loop, *i* gets the value 80, and that's what we print. And this will continue until *i* takes on every value in the list.

Now, note something important: *i* is getting assigned the value from the list. So, if we modify the value of *i*, it does not change the value in the list. Let's look at this slightly modified code, where we set *i* to 0. On our first iteration of the loop, *i* takes on the value of the first element, 83. Then, when we encounter the line i = 0, it's going to change the value of *i*, but not change the value in the list itself. If we wanted to change the temperature values in the list, we'd need to use a different loop, like the index values that we saw earlier. Notice that this code, where we assign a value to the elements themselves, would set all of those values to 0.

Let's practice a little bit more. Assume we had our days-in-month list from earlier. Let's say that we wanted to compute the total number of days in the year. To do this, we'll have a variable, and we'll add the number of days from each month to that variable. Now, there are a few ways we could do this. See if you can come up with one of them yourself.

OK, we're looking for a sum, and its name can be num_days, which we'll initialize to 0. We can loop through from 1–12 and add the number of days on. In each case, we give the list name, and then the element number, *i*, in the square brackets. In this case, we know that the list will have 12 elements since there are 12 months in the year. We could have been more general and made the range cover a list of changeable length. A different option would be to actually loop through the different elements. In this case, we let *i* take on the values of the list elements, by writing for i in days_in_month, and so we will add *i* to the sum each time. As an aside, since computing sums is a pretty common thing, there's actually a command in Python that will do this for us—we just call it the sum command, putting the list into the parentheses.

OK, that gives us all the fundamentals of working with lists. However, Python gives us a bunch of other things we can do that make lists even more useful. In particular, I want to go over appending, indexing, slicing, and lists of lists because each of these are part of what makes Python a particularly useful language.

First, let's look at another way to build up lists of elements. We've seen how we can create lists by separating items with commas like you see here. Well, we can also merge lists together by using the addition operation. This creates a new list by taking all the elements in the first list and then appending on those from the second list. In this case, we create list three by adding lists one and two together. We can also increase a list by using the plus/equals operation. This will let us append additional items onto the end of an existing list. You can see an example here, where we've appended list two onto the end of list one.

Now, if we don't have a list, but rather just one item, we can also add it to a list. The format for this command is, again, going to look a little weird right now, but you've seen things like it before when we were closing files; and eventually, we're going to explain why commands look like this. There is an append command that lets us add one element onto the end of an existing list. Here's what it looks like. We have the name of the list, then a period, then the word append, and then in parentheses,

the thing that we want to append on. In this case, we take list one, and we add on the value 3.9 to the end of it.

Now, appending can be really useful for building up lists. There are many times when we want to keep adding things to a list, but we don't know at first how long it's going to be. For instance, maybe we want to get a list of all the ages of the people in a group. We don't know ahead of time how many people there will be, so we just keep adding on to the list until we're done. Take a look at the code here. We start out with an empty list, and we ask for ages. Every time we get an age, we add it on to the list. If the user enters –1, which is obviously not a valid age, we take that as the indication that we're done. Notice that this code will work to collect a list of all ages, whether there are no people or 20.

Besides different ways to build up a list, there are also different ways to index into a list. We've already seen the basic way we index into a list, by putting the element number of the list in brackets. This is the example we've looked at before, where we have a list of temperatures during a week, and we print out the first temperature. Well, what do you think will happen if we put in an index number that's too large? Remember that the array has just 7 elements, so they'll be numbered 0–6. And here, we're trying to access element 7.

Well, in this case, we get an error, saying that we're out of range. That's actually a good thing; the computer is not letting us access something past the end of the list.

OK, what if we put in negative one for the index? What do you think will happen? Well, if you guessed that it would be an error, that was a reasonable guess, but Python has a better idea. Putting in –1 actually gives us the value of the last element of the list—86. This is a convenient tool in Python to let us pull items off of the end of the list. And, this continues. If we look at index –3, we actually get the third item from the end, or 79.

Python also lets us pull out a portion of a list with a pretty cool operation called slicing. With a slice, we pick where to start and where to stop. So,

we don't have just one index value, we actually have a colon separating two values. You can think of these values just like in the range examples we saw when we were working with loops. The first value—which is *a* in the code here—is the starting point. The second value—which is *b* in this example—is the number that you want to stop before. Remember that the index of the first element in the list is 0.

So, for example, if you want the first three elements, you would enter 0:3, saying that you want elements 0, 1, and 2. If you want elements 4 and 5, you could enter 4:6. Now, if you leave off the number before or after the colon, it means you want to start from the beginning, or go to the end. So, the first six elements would be just :6, leaving off the first number because you want to start from the beginning. The last three elements would be −3:. And, if we just put a colon, it means from the beginning to end, which is another way of saying we want a copy of the entire list.

Let's look quickly at how you could use this with our temperature example. Let's assume that these are the 7 days of the week, starting with Sunday. If we wanted just the temperatures on weekdays and not weekends, we could take the slice 1:6. Because counting starts at zero, this would leave off the first day; and because of the way Python reads a range, this would also leave off the sixth, which is the last day of the week, leaving us with the rest. Also, notice here that if we print out the list, we see the brackets and everything.

Slicing can let us do some interesting things. We can reassign values to slices, for instance, replacing part of a list with another list. For example, look at this code. In this case, we've deleted all the weekdays from our list, leaving us with only the Sunday and Saturday temperatures. The slice operation took that middle part of the list and replaced it with an empty list.

Now, let's say that we actually wanted to insert some new values into the list. We could take a slice specified as 1:1. Now, normally this slice would not mean much—it would be empty since there's no value starting with 1 but ending before 1. However, in this case, the slice refers to the spot right before element 1. So, when we set that slice to some new list, it's like we're inserting that list, right in between elements 0 and

1. If we wanted to insert the new values at the beginning, we could put them at slice 0:0, or just plain :0. And, if we wanted to put them after the last element, we could put them at 2:2. Notice that while an index like 2, which is one past the highest index value, is not allowed normally; it is allowed when slicing.

Let's practice with slicing just a bit more. Remember our array where we stored the number of days in a month? How would you get a new list for just the summer months, with the months June through August? Well, we would just use our slice operation. June is the sixth month, which means we'd start with element number 5, remembering that numbering starts at 0. We'd want three months, which would mean our ending index is three more, or 8. This slice will get the days per month for the 3 summer months.

Now, what if I wanted the number of days for the first 3 months of the year, and then also for the last 3 months of the year. How might I generate those two lists? In this case, we can get the first three elements by slicing with just :3, and the last three elements by slicing with −3:. That is, we can take all elements from the beginning, but with index less than 3; and then take elements starting 3 from the end, all the way to the end.

OK, let's try one final thing. How might we form one list containing just the first 3 and last 3 months' worth of data, but not the 6 months in the middle? Here are two ways of doing that: We could take the two lists we just formed—the first 3 months and last 3 months—and add them together to get our new list. A second option would start with a copy of the entire list; that's the slice with just a colon—meaning a slice from the beginning to end. Then, we would take the middle of that list—that's the slice from 3–9—and set it to an empty list. This would also leave you with a list containing just the first 3 and last 3 months' worth of data.

Now, I want to tell you something that might surprise you a little. We've actually been using lists in this course for a long time—since the very first lecture in fact. You know all the strings we have, where we're using big lists of characters, all strung together? That's right, strings are actually a slight variation of a list. That means that many of the things

we've just been looking at with lists can also be done with strings. So, if we have the string Arthur, King of the Britons, we can slice out the first six elements and get Arthur. Or, we can get elements 8–11, which is King. Or, we can get the last four characters, which are tons. Cool, huh?

Basically, slicing on strings works just like slicing on lists. The one big difference is that we can't assign to a string the same way we did with lists. That is, we can't delete parts of strings, or insert strings in the middle like we were doing with lists. You need a separate set of commands to manipulate strings. For example, if you take the string and follow it with period lower parentheses, you get a new string—the same as the previous but all in lower case. Or, if you follow it with a period capitalize parentheses, you get a new string that capitalizes only the first letter and has the rest in lower-case.

Notice that we can make a list out of anything. We can have lists of integers, we can have lists of floats, we can have lists of strings, and we can even have lists of lists. Here's a list of three lists, each of which has three elements. If we print out the first element of this list, what do you think will happen? Well, in this case, it prints out element zero of the list of lists, which is the list 4, 4, 6.

How about this one? Notice that we have two sets of brackets here. What do you think will happen in this case? OK, the output here is 7. To see why let's look at the statement. What happens is that we first find list_of_lists[2], which is the list 7, 8, 9. Then, we take element 0 of that list, which in this case gives us our answer—the number 7. We could even have lists of lists of lists. Or, lists of lists, of lists, of lists—that's a tongue twister. The idea is the same with all of these.

Lists of lists can be very useful. Say you want to represent a chess board. A chess board has 8 rows and 8 columns, and each space can either be empty or can contain a piece. We could represent a chess board by making a list of 8 rows of 8, each of which contains a string giving the piece that is in that square. The example you see here shows one board configuration, where the letters indicate the various chess pieces, and a period indicates an empty square. Now, actually getting

the computer to play chess well, that's a little more complicated. And the way we'd present the data to a user, or get information from a user, about a move or the state of the board, that would be different. But, this basic structure—this is the way that we' would represent something like a chess board within the computer.

OK, I want to mention one more thing, though it's as much of a caution as it is a benefit. One thing that Python lets you do is create lists in which you have a combination of different types. Most languages require an array to be all stuff of the same type. But, in Python, it's OK to have a list of items like this, with an integer, a float, a string, and another list. There are times that things like this are useful when you're wanting to group unlike things together. For example, maybe we want to have credit card information, so we create a list with the name of the credit card, the credit card number, and the expiration date, all in one list. But, there are better ways of grouping stuff across different types, and that's objects, which we'll learn about later. Anyway, this is one of those things that's allowed in Python, and if you do have a list with mixed types, Python will let you treat that as an object.

Before we go on, let's look at one other thing that's very similar to a list that Python supports—the tuple. A tuple is like a list whose values can never change; and unlike a list, it often will contain different types of data. Tuples can be specified by giving a list of values, separated by commas. Here's the example we just saw with a credit card, where there's a variety of different stuff put together. In this case, credit card is a tuple, instead of the list like we had before. If we print out a tuple, it shows up as having parentheses around it, not square brackets. We can assign values from a tuple to a set of variables, separated by commas. In the example, our car tuple has a car make, model, and year, and we can assign the tuple to three separate variables—the make, model, and year. We could have done this with lists, also, though again, that's not the best use of lists. We can also access parts of the tuple using indexes or slices, just like with lists. However, we can't actually change the value of a tuple once it's created. So, imagine that we're responsible for keeping track of today's population census, and we have information about the number of people in each of the households in the U.S. You

can see now how we could potentially store that information in a list while it would be nearly impossible to store that information in a bunch of individual variables.

In our next lecture, we're going to look at how we can put together some of these tools—like lists, and loops, and files—to make a more complex and interesting program that lets us analyze weather data. See you then.

## MORE PYTHON PRACTICE

OK, here's an exercise you can try, to get some practice using lists. Write a program that asks a user to type in two pieces of information about however many people they want—a person's name, and then the person's age. You can have them type "stop" for the name when they're ready to stop. Then, print out which person is the oldest.

Need a hint? Try creating two lists—one for the names, one for the ages—so that the names and ages are in the same order in both lists. These are called parallel arrays, or parallel lists. Then, figure out from the age list which age is the largest. The corresponding name will be the name of the oldest person.

Here's one example of how this could be implemented. At the beginning, we have our two lists—one for names, one for ages. We have a loop to keep getting names and ages until someone enters stop. Each time they enter a name and age, we append them to the appropriate list. Then in the second part, we try to find the person with the biggest age. We start by assuming the first person entered is the oldest, and then we loop through all the other ages. And if we find an age bigger than the biggest one so far, that becomes our new maximum. Notice that we keep track of the index for that oldest age.

Finally, when we've gone through all the ages, we have the index for the oldest person. So, we print out the name of the person with that index. Ta-da! All done.

08

# Top-Down Design of a Data Analysis Program

When we're faced with a programming problem, we'll often want to go right into the process of writing code. In very simple cases, this can work, but most programming requires a more powerful approach, one that lets us take a problem we face and break it up into bite-sized pieces. This insight led to one of the most well-known software development techniques: top-down design. For the program in this lecture, you will develop a system that will let you load and analyze weather data collected over time.

## [ TOP-DOWN DESIGN ]

› **Top-down design** is one of the most commonly applied methods for developing computer programs. With top-down design, you begin by looking at the big picture—the top—which is often too much to consider all at once in fine detail. You then break that overall task into a series of more manageable tasks—that is, you work your way down from the top.

› Top-down design is a good way of approaching programming problems. If you continuously analyze a problem and break it down into smaller conceptual ideas, eventually you'll reach the point where the idea you need to express can be written in just a single line of code. Using top-down design usually leads to well-organized code, where the purpose of each section of code is clear and concise.

## [ READING IN THE DATA AND STORING IT INTO A LIST ]

› Let's assume that we have a file containing some temperatures measured in a particular city over a period of time, and we want to understand something about the weather patterns. What has been the pattern on a birthday or anniversary? What about weather during a week we have in mind for a trip?

› The following is an example of a piece of the data file we'll be able to use. It contains 15 years' worth of data. For each day, we have a date, the high temperature on that date, the low temperature on that date, and the rainfall on that date, all separated by commas.

```
1/1/2000,79,37,0
1/2/2000,79,68,0
1/3/2000,73,50,0
1/4/2000,51,26,0
1/5/2000,57,19,0
1/6/2000,59,46,0.08
1/7/2000,53,48,1.61
1/8/2000,53,44,0.76
1/9/2000,72,43,0.01
1/10/2000,75,35,0
1/11/2000,77,42,0
1/12/2000,79,64,0
1/13/2000,72,57,0
1/14/2000,66,43,0
1/15/2000,73,39,0
1/16/2000,78,55,0
1/17/2000,77,51,0.01
1/18/2000,81,57,0
1/19/2000,78,48,0
1/20/2000,64,43,0
```

- This type of file is called a comma-separated value (CSV) file, or sometimes a comma-delimited file. For data that starts out in Excel or another spreadsheet program, it's easy to use the spreadsheet program to output the data values, separated by commas, as a CSV file.

- To get weather data like this, you can go to any one of several weather sites, some of which will let you download it into a spreadsheet, but you can also cut and paste data into a spreadsheet yourself. Once you have it in the spreadsheet, you can save it as a CSV file.

- Let's assume that we have this file of weather data and that it's sitting in our directory where we're writing our Python program to make it easy to find.

- Our Python program might be called "Your Special Day," and we'll set it up to answer some queries about past temperatures on that calendar day, such as the average daily high and low.

- To design this program using a top-down approach, we want to think about what has to happen in the broadest terms first. At this broadest level, we have three basic steps common to many programs: We read in our data, analyze our data, and present the results to the user. This is the end of the "top" level of the design. At this stage, we don't worry about how these particular pieces are handled—just that these are the main tasks.

- Usually, in design, we want to do a bit more than just sketch out one level before we write code. But notice how some of this can already translate to code.

```
#Read in Data
#Analyze Data
#Present Results
```

- The "code" that we have here is just three comments. We're not actually writing commands yet, because we haven't really designed any of the low-level details. All we can do at this point is give the outline of each of the main sections, and we're designating those with comments.

- These comments we put in help us understand what the goal and meaning of a section of code is. Although comments don't actually turn into any machine instructions, they are critical to helping people understand programs, including the person writing the program. Comments will help you design the program and remember what's going on when you come back to your code later.

- If we go one level down from our top-level design, our first task is to read the data in. To do this, at a high level, the normal pattern will be to open the file for reading, to actually read in the lines of the file, and then to close the file.

- Let's first look at what we have so far in terms of how it would appear in comments. We are basically taking each level of the design and putting in a comment describing what it does. All of these comments can get a little messy, so sometimes it helps to put in some additional characters to help visually separate code, such as the following sets of 10 hashtags.

```
########## Read in Data ##########
#Open File
#Read lines from File
#Close File
########## Analyze Data ##########
########## Present Results ##########
```

- Again, we have some pretty high-level ideas, so let's break those down a bit more. We'll start with the reading-in-data part. To open the file, we need to know the filename, which we'll assume the user is going to give us. Then, we need to actually give the open command. At this point in our design, we're basically at the level of individual lines of code. Each of these ideas corresponds to one or maybe a few lines of code. So, we can actually start writing the code now.

- We'll start by asking the user to input a filename, and then we'll call the open command for that file, noting that we want to read the file. We'll write our open command, calling the file that we open "infile" and listing the filename and the letter *r* in the parentheses.

```
########## Read in Data ##########
#Open File
filename = input("Enter the name of the data file: ")
infile = open(filename, 'r')
#Read lines from File
#Close File
########## Analyze Data ##########
########## Present Results ##########
```

- Before writing more code, we should test our code. We want to make sure that we didn't make a mistake at this point. In this case, we have just two lines of code, so we can run this to see if it seems to work. When we run, we get a message asking for the file, and then we type in the name of the file, which is DataFile1 in this case. And it seems to open okay. Note that DataFile1 has to already be created and sitting in the same directory as your Python program. You should continue testing your code along the way and fixing problems as you find them.

**KEEP IN MIND**

If you forget the syntax for a specific command, you can always look it up online. A good reference for the syntax of various commands is the Python tutorial:

https://docs.python.org/3/tutorial/index.html.

- Once we've done some tests, next we should think more about our design. As with any design process, we need to think about our plan before jumping forward and implementing something. We've already completed the file opening, and our next task is to read in the data from the file. We'll be reading in line after line, doing basically the same thing. So, this is going to mean looping through all the lines of the file.

- For each of those lines, we need to read in the line as a string—remember that our input comes in as a string. Then, we need to pull out the individual pieces of data from the string and finally store them in a list that we can access later.

› We'll develop this code in stages. First, we need our loop. We'll loop through the lines of the file with a simple for loop. The following for loop is going to read each line from a file, and it will be stored in the variable "line." Notice that we also set up an empty list, "datalist," that we can use to hold the data that we collect.

```
########## Read in Data ########## #Open File
filename = input("Enter the name of the data file: ")
infile = open(filename, 'r')
#Read lines from File
datalist = []
for line in infile:
    #get data from line
    #Put data into list
#Close File
########## Analyze Data ##########
########## Present Results ##########
```

› We need to extract the individual elements from a line that we've read in. We have a string that has all this weather data in it. We want to pull out the first part (date), the second part (high temperature), the third part (low temperature), and the fourth part (rainfall). These are all separated by commas. So, we'd like to have some command that would let us take the string and get back each of the parts that's separated by a comma.

› We can do all of this with a built-in Python command that lets us pull out parts of a string that way. The command is called "split." We use it by taking the string name, which in this case is "line," followed by a period, and then in parentheses, we put the character that tells us how to separate the parts (a comma in this case). The split command returns a set of values that we can assign to several variables—*a*, *b*, and *c* in this case.

```
a, b, c = line.split(',')
```

› The thing it returns is a tuple, and we'll be able to assign the tuple to several values.

- In our design, we need to split the line up into its parts, and then we need to store each value in the appropriate place—a variable of the right type.

- The “line.split(‘,’)” will give us back the four elements of the line we care about: the date, high temperature, low temperature, and rainfall. We probably want temperature and rainfall values to be numbers, and the split command is going to return strings. So, we can convert these values using the type converter. Let’s convert each of the temperature strings to integers. For rainfall, let’s convert the string to a float.

- Instead of keeping the date as one single string, it is probably better to break up the date into separate variables—a month, day, and year. To do this, we can use the split command again, dividing the date into its three components. This time, our separator is the slash. Then, we turn each of those date components into an integer.

- We still need to put this data for this line into a list so that we can retrieve it later in the analysis phase. So, we we’ll create a list with all the data we just found for one date: the day, month, year, low temp, high temp, and rainfall. We’ll append that list into the “datalist” variable that we created. So, “datalist” is going to be a list of lists.

- Let’s also reverse the order of the low and high temperature in our own list versus what was in the original data file, and let’s reverse the order of month and day.

```
########## Read in Data ##########
#Open File
filename = input("Enter the name of the data file: ")
infile = open(filename, 'r')
#Read lines from File
datalist = []
for line in infile:
    #get data from line
    date, h, l, r = (line.split(','))
    lowtemp = int(l)
```

```
        hightemp = int(h)
        rainfall = float(r)
        m, d, y = date.split('/')
        month = int(m)
        day = int(d)
        year = int(y)
        #Put data into list
        datalist.append([day, month, year, lowtemp, hightemp,
          rainfall])
    #Close File
    ########## Analyze Data ##########
    ########## Present Results ##########
```

› The following is what the resulting datalist will look like for the first several items in the input file.

```
1/1/2000,79,37,0
1/2/2000,79,68,0
1/3/2000,73,50,0
1/4/2000,51,26,0
1/5/2000,57,19,0
...
datalist:
[ [1, 1, 2000, 37, 79, 0.0], [2, 1, 2000, 68, 79, 0.0], [3, 1,
  2000, 50, 73, 0.0], [4, 1, 2000, 26, 51, 0.0], [5, 1, 2000, 19,
  57, 0.0], ... ]
```

› The last thing we need to do in this reading-in-data part of our overall design is close the file. That's a simple process, needing just a single line of code.

```
#Close File
infile.close()
```

## [ PROCESSING THE LIST AND ANALYZING THE RESULTS ]

› Once we've finished the first piece of the program, we're free to think about the next overall piece in our top-down design. We've read in our data from the file, and we've stored it into a list. We now need to process that list and analyze the results.

› Remember that our goal with this program is to find out historical data about a particular date. So, we'll need one additional piece of input: the date we're dealing with. Then, we'll need to get the relevant historical data for that date. Finally, we'll need to analyze that historical data to find the information we care about.

› To get the date from the user, we'll need to ask the user for the month and the day. There are many ways to do this, but to keep it simple, we'll ask first for the month and then for the day. There will be just one line of code for each piece of the design. We just have a few input commands, each time converting the input to an integer.

```
########## Analyze Data ##########
#Get date of interest
month = int(input("For the date you care about, enter the
  month: "))
day = int(input("For the date you care about, enter the
  day: "))
#Find historical data for date
#Perform analysis
```

› Next, we want to pull out all the historical data for the date we care about. We have read the original data file into a datalist, which is a list of lists. Each of the lists inside has the day, month, year, low temp, high temp, and rainfall. Now, we have a particular month and day that we care about, and we want to find all of the historic data for those dates.

- We want to go through the datalist and for each of the lists in there, check whether the day and month match the ones we care about. If they do match, we'll store that information in a different list.

- We first create an empty list called "gooddata," which will hold the data for the dates that match our target date. We then loop through all the datalist. Notice that in the for loop, we will refer to each element of datalist as "singleday," because it is the data for one single date.

- In the loop, we compare the first two elements of "singleday," which correspond to the day and month, to the day and month that the user entered in. If they match, then we take the remaining elements of "singleday" and put them into a new list, which gets appended to "gooddata."

```
########## Analyze Data ##########
#Get date of interest
month = int(input("For the date you care about, enter the
  month: "))
day = int(input("For the date you care about, enter the
  day: "))
#Find historical data for date
gooddata = []
for singleday in datalist:
    if (singleday[0] == day) and (singleday[1] == month):
        gooddata.append([singleday[2], singleday[3],
          singleday[4], singleday[5]])
#Perform analysis
```

- After we get through this code, we should have a list of the information corresponding to the date we care about. Next, we need to analyze it.

- Let's figure out some key information about the highest and lowest temperature and the average high and low. To do this, we'll loop over all the dates and keep track of information as we look at each record.

- First, we'll count the number of dates so that we can get an average. Next, we'll keep track of the highest and lowest temperatures we've seen so far. Finally, we'll add up all the maximum and minimum temperatures so that at the end of the loop we can calculate an average high and low.

- The corresponding code starts by initializing all of the variables we need for the analysis. We set the sums to 0, and for the maximum and minimum temperatures seen so far, we start out by setting them to some extremely low and high values. Because we set a super-high minimum temperature, the first date we encounter will have a lower minimum; likewise, it'll have a higher maximum than the very low value we start out with.

- Then, we loop through all the data, and we update each of those values along the way. For every date, we increase "numgooddates." We add the maximum to "sumofmax" and the minimum to "sumofmin." We then check to see if the minimum is lower than the minimum we've seen so far, and if so, we update that. We also check to see if the maximum is larger than the maximum seen so far, and if so, we update that.

- After the loop, we calculate the average maximum and minimum by dividing the sum by the number of dates.

```
########## Analyze Data ##########
#Get date of interest
month = int(input("For the date you care about, enter the
  month: "))
day = int(input("For the date you care about, enter the
  day: "))
#Find historical data for date
gooddata = []
for singleday in datalist:
    if (singleday[0] == day) and (singleday[1] == month):
        gooddata.append([singleday[2], singleday[3],
          singleday[4], singleday[5]])
```

```
#Perform analysis
minsofar = 120
maxsofar = -100
numgooddates = 0
sumofmin=0
sumofmax=0
for singleday in gooddata:
    numgooddates += 1
    sumofmin += singleday[1]
    sumofmax += singleday[2]
    if singleday[1] < minsofar:
        print(minsofar, singleday[1])
        minsofar = singleday[1]
    if singleday[2] > maxsofar:
        maxsofar = singleday[2]
avglow = sumofmin / numgooddates
avghigh = sumofmax / numgooddates
```

## [ GIVING THE OUTPUT ]

› Once we've completed all the parts of the analysis, we're going to look at the last of the three major parts: giving the output. We just print out the highest and lowest values we've seen, and we print out the average low and high.

```
########## Present Results ##########
print("There were",numgooddates,"days")
print("The lowest temperature on record was", minsofar)
print("The highest temperature on record was", maxsofar)
print("The average low has been", avglow)
print("The average high has been", avghigh)
```

## Reading

Zelle, *Python Programming*, chap. 9.

## Exercise

Modify the program developed in this lecture to also keep track of the chance that there will be rain on the particular day. Below are a few hints if you need them.

- ◊ You already are reading in the data you need to.
- ◊ You will need to add some additional lines to the analysis and presentation parts of the code.
- ◊ Keep track of how many days had rain as you go through the list "gooddata."
- ◊ Compute a percentage of days with rain and report that.

TRANSCRIPT 08

# Top-Down Design of a Data Analysis Program

When we're faced with a programming problem, it can be exciting to immediately start thinking about how our solution is going to get turned into code. We want to jump right in and start seeing results. Even if we give the program a little thought ahead of time, jotting down some notes or putting some comments in our program, we'll often want to go right into the process of writing code. We'll start at Line 1 and go forward, as though we were writing an essay, but in code.

In very simple cases, this can work—we just write one line of code after another, testing along the way, until we're done. But most programming requires a more powerful approach, one that lets us take a problem we face and break it up into bite-sized pieces. This insight led to one of the most well-known software development techniques there is top-down design. Top-down design emerged as a major approach in the 1970s as software became more complex and applied to a wider range of applications, and it's still one of the main development paradigms used today.

For our main program in this lecture, we're going to develop a system that'll let you load and analyze weather data collected over time. But the main point I want you to get from this lecture is not just the specific code we write, or even how we analyze data. The more important lesson is the overall process we use to build the program, a lesson that builds on what we've already seen about the importance of testing and iterative development. For any project consisting of more than a few lines of code, we can always benefit from having a process to follow that makes the task more manageable.

Top-down design is one of the most commonly applied methods for developing computer programs. As an analogy, suppose that you're

responsible for planning a very large family dinner. Well, thinking about every detail of the dinner all at once can easily become too overwhelming, so instead you might first think about which courses you want to serve. For instance, you might want an appetizer, a main course, and a dessert—easy enough. Once that's settled, the next step might be thinking about what dish to serve for each of those courses. For the dessert, for instance, you might choose to have cake, so you think about what's needed for the cake. You need to cook the cake, prepare the frosting, and then put the frosting on the cake. Cooking the cake has a series of steps all its own.

Well, this process is an example of top-down design. With top-down design, you begin by looking at the big picture, the top, which is often too much to consider all at once in fine detail. You then break that overall task into a series of more manageable tasks—that is, you work your way down from the top. For instance, if you're planning a big vacation, you might need to plan out your transportation and your hotel and your activities each day. Each of those might get broken down even more—your transportation might include getting to the airport, flying on a plane, and then getting to your hotel. You could use the same top-down approach for preparing a presentation, writing a book, organizing a home improvement project, and many other kinds of major projects.

Top-down design is also a good way of approaching programming problems. When faced with a problem, it helps to break it into smaller, more manageable pieces. If you continuously analyze a problem and break it down into smaller conceptual ideas, eventually you'll reach the point where the idea you need to express can be written in just a single line of code. Using top-down design usually leads to well-organized code, where the purpose of each section of code is clear and concise.

So, let's see how we can apply top-down design to our particular programming problem. Here's the setup: Let's assume that we have a file containing some temperatures measured in a particular city over a period of time, and we want to understand something about the weather patterns. What's been the pattern on a birthday or anniversary? What about weather during a week we have in mind for a trip?

Here's an example of a piece of the data file we'll be able to use; this one happens to have data from where I live, and I just copied 15 years' worth of data off of a website I found. For each day, we'll have a date, the high temperature on that date, the low temperature on that date, and the rainfall on that date, all separated by commas. This type of file is called a comma separated value, or CSV file. Or, it's sometimes called just a comma delimited file. For data that starts out in Excel or another spreadsheet program, it's easy to use the spreadsheet program to output the data values, separated by commas, as a CSV file.

To get weather data like this, you can go to any one of several weather sites—a quick web search for weather data will bring up some websites that will provide this historical data for you. Sometimes they'll even let you download it into a spreadsheet, but at worst you can cut-and-paste data into a spreadsheet yourself. Once you have it in the spreadsheet, you can save it to a CSV file; when you're saving it, you should have an option to save it as a .CSV file.

So, we'll assume that we have this file of weather data and that it's sitting in our directory where we're writing our Python program, to make it easy to find. Our Python program might be called Your Special Day, and we'll set it up to answer some queries about past temperatures on that calendar day. Maybe we want to find the hottest and coldest temperatures on record, the average daily high and low, the largest one-day temperature swing, the high and low in a month surrounding the date in question, et cetera.

So, let's see how we'll design this program using a top-down approach. We want to think about what has to happen in the broadest terms first. At this broadest level, we have three basic steps common to many programs: we read in our data, we analyze our data, and we present the results to the user. Believe it or not, this is the end of the top level of the design. At this stage, we don't worry about how these particular pieces are handled, just that these are the main tasks.

Now, usually in design, we want to do a bit more than just sketch out one level before we write code, but notice how some of this can already

translate into code. The code that we have here is just three comments. That's right; we're not actually writing commands yet since we haven't really designed any of the low-level details. About all we can do at this point is give the outline of each of the main sections, and we're designating those with comments.

Now, these comments that we put in are important—they help us understand what the goal and meaning of a section of code is. Some programmers neglect comments as a waste of time; I mean, after all, comments don't actually turn into any machine instructions. If we leave them out, the code will run exactly the same.

Comments, though, are critical to helping people understand programs, including the person writing the program. Comments will help you design the program, and comments will help you remember what's going on when you come back to your code later. I've had chances to go back to code I wrote 20 years ago, and I'll tell you, I wish I had put in more comments. Even if you think, "I'll never see this code again," your comments can still help make sure that your code is following the design plan you had all along. And, if there's any chance that you'll go back to look at your code in the future, you'll be glad you spent the time putting in comments.

Now, let's get back to our top-level design, and try to go one level down. Our first task is to read the data in. So, let's think about what we need to do to read the data in, again at a high level. The normal pattern will be to open the file for reading, actually read in the lines of the file, and then close the file. So, let's first look at what we have so far in terms of how it would appear in comments. We are basically taking each level of the design and putting in a comment describing what it does. Now, all these comments can get a little messy, so sometimes it helps to put in some additional characters to help visually separate code, like the sets of 10 hashtags I've put in.

OK, again, we have some pretty high-level ideas, so let's break those down a bit more. We'll start with the reading-in-data part. To open the file, we need to know the file name, which we'll assume the user is

going to give us. Then, we need to actually give the open command. At this point in our design, we're basically at the level of individual lines of code. Each of these ideas corresponds to one or maybe a few lines of code, so we can actually start writing the code now. We'll start by asking the user to input a file name, and then we'll call the open command for that file, noting that we're wanting to read the file.

Now, what if you've forgotten at this point how to write a particular command? Well, besides reviewing lectures from this course, there are always online references that I want you to feel ready and willing to use. A good reference for the syntax of various commands is the Python tutorial at Python.org. For example, we can find a section on reading and writing files and then when we're there, we see, right at the top, a description of the open command. So, we'll write our open command, calling our file that we open infile, and listing the file name and the letter r in the parentheses. So, if you forget the syntax for a specific command, no worries, you can always look it up.

All right, at this point, after we've written our few lines of code, what's the next thing we should do? If you said, "Write more code," hold on. First, we should test our code. We want to make sure that we didn't make some silly mistake at this point. In this case, we have just two lines of code, so we can run this to see if it seems to work. When we run, we get our message asking for the file, now we type in the name of the file, which I happened to name DataFile1 in this case, and it all seemed to open OK, no apparent errors, so we'll go on from here. Now, note that I had to already have DataFile1 created and sitting in the same directory as my Python program.

Now, just to make sure it's really reading, we can put in a line of test code just to read in and print out the whole file. Calling infile.read() will read in the entire file, and so by saying print infile.read(), we should be able to see the whole file. If we run this code, sure enough, we see the entire file output to the screen. This line of code where we printed the file was just for diagnostic purposes, so once we're done with the test, we should delete it. Or, we can turn our test into a comment by putting a hashtag in front of it. That way, we can see precisely how and where

we tested our code, and we can always reactivate that test later on by taking out the hashtag. Once the code is fully working, though, we want to get rid of comments like that so that our code isn't too confusing. Remember, you should be testing all along the way, and fixing problems as you find them. Now, for the sake of time, I'm going to skip showing you these tests as we keep going in this lecture, but don't forget that you should be testing regularly.

All right, if you've done enough tests, it's time to write some more code, right? No. Next, we should think more about our design. As with any design process, we need to stop and think about our plan before jumping forward and implementing something. We've already completed the file opening, and our next task is to actually read in the data from the file. We'll be reading in line after line, doing basically the same thing. So, this is going to mean looping through all the lines of the file. For each of those lines we need to read in the line as a string—remember that our input comes in as a string. Then, we need to pull out the individual pieces of data from the string, and finally, store them in a list that we can access later.

OK, we'll develop this code in stages. First, we need our loop. We'll loop through the lines of the file with a simple "for loop." The "for loop" that we see here is going to read each line from a file, and it'll be stored in the variable line. Notice that I've also set up an empty list, datalist, that we can use to hold the data that we collect.

OK, back to the design again. We need to extract the individual elements from a line that we've read in. Remember what lines of the data file look like? We have a date in month/day/year format, followed by a comma, followed by a high temperature, followed by a comma, followed by the low temperature, followed by a comma, and finally a rainfall amount.

Now, I'm going to introduce you to a new command that'll help you process data like this. Let's think about how this command should work. We have a string that has all this data in it, and we want to pull out the first part, the date; the second part, the high temperature; the third

part, the low temperature; and the fourth part, the rainfall. These are all separated by commas. So, we'd like to have some command that would let us take the string and get back each of the parts that's separated by a comma.

We can do all this with a built-in Python command that lets us pull out parts of the string that way. The command is called "split." We use it by taking the string name, which in this case is line, followed by a period, and then in parentheses after the word split, we put the character that tells us how to separate the parts. In this case, we're using a comma to separate the parts. The split command returns a set of values that we can assign to several variables, *a*, *b*, and *c* in this case. This thing it returns is a "tuple," and we'll be able to assign the tuple to several values.

Let's see an example. Say we had a phone number given with all the parts separated by dashes. The code you see here will help us reformat a phone number into a different form. The split command will help us pull out each part: by putting a dash in the parentheses, we're saying that we want each part that's separated by a dash, so, in this case, we get the area code, the prefix, and the suffix. Then, we can reconstruct a new string by putting parentheses around the area code.

You can also call the split command without saying what the separator will be. In this case, it'll use whitespace as the separator. Whitespace means characters that don't show up as text, things like spaces, new lines, and tabs. So, in the example, we have a couple of sentences in our sourcetext string. If we call split, it will pull out each word individually. Notice that any whitespace, including a newline character, will work as a separator. In the end, we get a list of words, although it's important to remember that punctuation like commas and periods is not whitespace, so it'll become part of the words.

Try this out yourself. Let's say we have a greeting like you see here, and we want to print it out. But, instead of having the "/n" for newline, this greeting has a "nl" written in angle brackets—the less-than and greater-than symbols. How might you print this out so that each of those "nl" symbols gets treated as a newline? Here's what your code might

look like. You can first split the greeting into different lines by using the split command and the "nl" in brackets as the separator. Then, just loop through that list, printing each line individually.

OK, let's come back to our main program. In our design, keep in mind that we have two things going on here: we need to split the line up into its parts, and then we need to store each value in the appropriate place—a variable of the right type. So, what do you think the code will look like for the first part of that, actually splitting the line?

Remember that our separator was a comma, so the line here, where we call line.split with a comma character inside the parentheses, will give us back the four elements of the line we care about: the date, high temperature, low temperature, and rainfall. Now, we probably want temperature and rainfall values to be numbers, and the split command is going to return strings. So, we can convert these values using the type converter we've seen several times before. Let's convert each of the temperature strings to integers. For rainfall, we'll convert that string to a float.

We could also do a little bit more. It might be OK to keep the date like it is, as one single string, but it might be better if we had the month, day, and year separated out into separate variables. Based on what we've seen so far, how would you take the date and break it up into a month, day, and year? Yes, we can use the split command again, dividing the date into its three components. This time, our separator is going to be the slash. Then, we turn each of those date components into an integer.

We still need to put this data for this line into a list so that we can retrieve it later in the analysis phase. So, we'll create a list with all the data we just found for one day: the day, month, year, low temp, high temp, and rainfall. We'll append that list onto the datalist variable that we created. So, datalist is going to be a list of lists. While we're at it, let's reverse the order of the low and high temperature in our own list versus what was in the original data file, and let's reverse the order of month and day. We don't want to have to stick to the order in the original data; we can store the data internally in whatever order makes sense to us. So, even

though the original data had month before day and high temperature before low temperature, we can store it in a different order that makes more sense for us.

Here's what the resulting dataset will look like for the first several items in the input file. Notice that each value was converted to a number of the appropriate type, and the lists within the bigger list store the data in the order we specified. Now, there's one more thing we need to do in this reading-in-data part of our overall design, and that's close the file. That's a simple process; it only needs one line of code.

Now, at this point, you should've tested your code at several points along the way, especially whenever you finish one section of code. Let's stop for just a second and see where we stand. We had our overall design, in which at the top level we had three subpieces. The first of those, reading in data, had three subpieces of its own. One of those parts had a couple of subparts, and another had even more. We gradually worked our way down from the top to each of those, until in the end, we had just one or a few lines of code corresponding to each piece of design. If we look at the code itself, and in particular at the comments that we see in the code, this top-down design is pretty clear—you can see the overall design from these pieces.

So, we've finished the first piece of the program, and now we're free to think about the next overall piece in our top-down design. We've read in our data from the file, and we've stored it into a list. Now, we need to process that list and analyze the results. Remember that our goal with this program is to find out historical data about a particular date, so we'll need one additional piece of input: the date that we're dealing with. Then, we'll need to get the relevant historical data for that date. And finally, we'll need to analyze that historical data to find the info that we care about.

In order to get the date from the user, we'll need to ask the user for the month and the day. There are lots of ways we could do this, but to keep it simple, we'll ask first for the month, then for the day. So, for this design, how would you write the code that reflects this? There'll be just one line

of code for each piece of the design. We just have a couple of input commands, each time converting the input to an integer.

OK, now let's turn to a trickier part. We want to pull out all the historical data for the date we care about. Remember what the original data file looked like. We've read this in to a datalist, which is a list of lists. Each of the lists inside has the day, month, year, low temp, high temp, and rainfall. Now, we have a particular month and day we care about, and we want to find all the historic data for that date. What we want to do is go through the datalist and, for each of the lists in there, check whether the day and month match the ones that we care about. If so, we'll store that information in a different list.

Let's see what the code looks like, but just to keep things simple, we'll focus only on this section of code. We first create an empty list called gooddata. This is going to hold the data for the dates that match our target date. We then loop through all the datalist. Notice that in the "for loop," we will refer to each element of datalist as singleday, since it's the data for one single date. In the loop, we compare the first two elements of singleday, which correspond to the day and month. We compare them to the day and month that the user entered in. If they match, then we take the remaining elements of singleday and put them into a new list, which gets appended to gooddata. So, after we get through this code, we should have a list of the information corresponding to the date that we care about. Now, we need to go ahead and analyze it.

First, let's figure out some key information about the highest and lowest temperature, and the average high and low. To do this, we'll loop over all the dates, and keep track of information as we look at each record. First, we'll count the number of dates so that we can get an average. Now, in this case, I know that there are 15 years of data, so this should be 15, but maybe there's a missing day, or we don't know how much data we have, or it's a leap year day, February 29th, or whatever, so we might as well count. Next, we'll keep track of the highest and lowest temperature that we've seen so far. And, finally, we'll add up the max and min temperatures, so that at the end of the loop we can calculate an average high and low.

The corresponding code starts by initializing all of our variables that we need for the analysis. We set the sums to zero, and for the max and min temperatures seen so far, we start out by setting them to some extremely low and high values. Obviously, since we set a super-high minimum temperature, the first date we encounter will have a lower minimum. Likewise, it'll have a higher maximum than the very low value we start out with.

Then, we loop through all the data, and we update each of those values along the way. For every date, we increase numgooddates. We add the max to our sumofmax, and our min to sumofmin. We then check to see if the min is lower than the min that we've seen so far and, if so, we update that. We also check to see if the max is larger than the max that we've seen so far and, if so, we update that. After the loop, we calculate the average max and min by dividing the sum by the number of dates.

OK, let's turn back to our design. We've completed all the parts of the analysis. We're going to look at the last of the three major parts, and that's giving the output. This is the most straightforward part of this particular program—we just print out the highest and lowest values that we've seen, and we print out the average low and high. The code for this is pretty straightforward; we just have a set of print statements for values that we've already found.

So, now we have a complete program. Let's review this. Say we're thinking of having a Halloween party, and we want to know how hot it's likely to be. If we run the code, enter DataFile1 for our file, and then 10 and 31 for the date, we find that the temperatures ranged from 45 to 86 on that date, and our average low is around 56, and our average high is around 80. So, that tells us that we're likely to have pretty moderate weather. We could easily plan to have outdoor activities as a big part of the Halloween party, even into the evening.

Now, we could run this again with a different date. If we were planning something on the Fourth of July, it looks like we have temperatures ranging from 67–99, with an average low of 71 and an average high of 91. So, we can expect pretty warm weather, and if we're having an

outdoor activity, we probably want to make sure there's plenty of shade and ice.

Now, in our top-down design, we started by decomposing our program into three parts, and each of those got further broken down into parts. But this was not the only way we could've broken up the program. For instance, we could've first asked the user for information, and then done everything else. Instead of asking for the file name just before loading the file, and the date just before doing the analysis, we could've asked for both at the very beginning. Neither way is incorrect; you might have developed your own program along those lines instead.

Now, looking at our whole program, we've got about 60 lines of code, and that includes the empty lines and the comments. As computer programs go, this is still considered pretty small, but it's a lot bigger than what we've been looking at so far, so it's worth taking the time to make sure you understand it.

What we did was take our overall problem; we decomposed it into the main tasks: reading in data, analyzing data, and then presenting data. Then we broke each of those tasks down further. For example, we broke reading in data down into three parts: opening the file, reading in lines from the file, and closing the file. And, each of those was further decomposed, where possible, until what was needed for any one section was just an obvious few lines of code.

So, by using a top-down approach, we had an orderly way of approaching the entire problem, and we were able to keep the complexity of the design manageable. And, in the end, the actual code that we wrote was almost an afterthought—it was an obvious extension of the last stage of design.

Remember, programming is about more than just getting instructions to the computer; it's also about organizing ideas in a logical and precise fashion. Top-down design helps ensure that the ideas expressed in code are, in fact, organized—it keeps each part of your code at a manageable level. At any one level of the design, you don't have to

worry about all the tiny details below, or about the big picture above, you just worry about the level that you're in, and maybe the ones immediately above and below.

This idea of taking a problem and decomposing it into smaller parts is a powerful one, and it can make the work of designing software a whole lot simpler. Once you have an overall approach to design, there's another set of programming tools that can make actually transforming your design into code even easier. These tools are functions, and we'll be discussing these in our next lecture. We'll see how we can use not only pre-built functions but also how we can write our own functions to do whatever we want.

09

# Functions and Abstraction

In this lecture, you will begin exploring functions, which are commands, or groups of commands, to get things done. They're like miniature programs that take in some input, perform some action, and return some output. Use of functions also demonstrates maybe the most important idea in computer science: **abstraction**. With abstraction, we simplify all of the details and view a complex system through a simpler interface. Good programmers are intensely aware of abstraction and make use of it repeatedly in different forms.

## [ FUNCTIONS AND ABSTRACTION ]

- One of the main ways that we take advantage of abstraction when programming is through **functions**. The print and the input commands are examples of functions. We have the name of a function, followed by parentheses. There might or might not be something inside the parentheses.

- The term "function" is not universal. Other names for it include "routine," "subroutine," and "procedure." You'll also hear "method," although that's usually only in the context of object-oriented programming. Sometimes people use the term "function" with a more specific meaning, in which a function always returns a value.

- Some functions, such as the print function, just do something. When a program asks a function to do its thing, the term we use is **calling** the function. When you call the function, something happens, such as text getting printed to the screen.

- Other functions, such as the input function, not only perform some action, such as printing to the screen, but they also return a value. For the input function, the function returns a string that is whatever the person typed in. This return value can get assigned to a variable or whatever else is needed.

- When we think of function calls, the typical way to think of them is as a "black box." The function takes in some input, or not. Then, the function does something. Then, it returns some output, or not. For the person using the function, it's a mysterious box that takes input, does its thing, and produces output.

- As you write the details of your code, you'll find yourself writing functions. You'll also find that within those functions, you'll make use of more function calls, which are when you initiate some action that has been defined with a function. Calling "print" means that we're using the print function, and calling "input" means that we're using the input function. And you can treat these calls as black boxes of their own.

- Writing functions to work in Python has two different stages: defining the function and then calling the function to put it into use.

- We define a function by starting with the key term "def," short for "define." Next, we have the name of our function—the command we want to use to call the function. Following that are parentheses. If our function is going to take in some form of input, that information is going to be specified inside the parentheses. Then, we have a colon, just like we've seen in conditionals and loops. Finally, we have the commands that the function should do. These are indented, again just like we saw with conditionals and loops.

```
def functionname(...):
    #details
```

- The term we use for the first line defining the function is called its **header**. The actual commands it should do—the part that's indented—is called the **body**.

› Suppose that we want to create a function that just processes something and doesn't take any input or return any values—maybe it prints a particular warning message.

```
def warn():
    print("Warning! Use program at your own risk.")
```

› Notice that we start with the word "def," short for define. We then have a name for the function—in this case, "warn." There's nothing inside the parentheses, because this function doesn't take any input. After the colon, we indent the commands that we want. In this case, there's just one command, a print statement that displays a warning message.

› We can call this function when writing code. We may have some code and, in the middle of it, decide that we need to print out a warning message. We can do so by calling "warn," just like we would any other command.

```
def warn():
    print("Warning! Use program at your own risk.")
a = 3
print (a)
warn()
print("Welcome!")
```

› Notice that we need to have parentheses after the name of the function. When we execute this code, it works like you were expecting it to. The code before the function call works just like always, in this case printing the value "3." Then, when we get to the function call, it executes the function, in this case printing the warning message. After that, it executes the rest of the code, just like always.

```
def warn():
    print("Warning! Use program at your own risk.")
a = 3
print (a)
warn()
print("Welcome!")
```

```
OUTPUT:
3
Warning! Use program at your own risk.
Welcome!
```

› We could have just output that warning message directly; the code here works exactly the same way, and we didn't have to create a function to do it. But there are several reasons that we'd want a function to do this.

› Let's say that you have some code where you're asking users for sensitive information and you want to give them plenty of warning that they're about to do something dangerous.

```
a = 3
print (a)
print("Warning! Use program at your own risk.")
print("Welcome!")

OUTPUT:
3
Warning! Use program at your own risk.
Welcome!
```

› You might print a warning before each time you ask for information. This is a lot of repetition of the exact same thing, and when you're repeating the same commands over and over, this is often a good time to use a function.

```
print("Warning! Use program at your own risk.")
name = input("Enter your name:")
print("Warning! Use program at your own risk.")
address = input("Enter your address:")
print("Warning! Use program at your own risk.")
ccn = ("Type in your credit card number:")
print("Warning! Use program at your own risk.")
expiration = ("Enter the expiration date for your card:")
```

- It's straightforward to convert the warning message to a function. We simply define a function—warn—that prints the warning message. Then, we can replace every occurrence of the print statement with a call to "warn."

```
def warn():
    print("Warning! Use program at your own risk.")
warn()
name = input("Enter your name:")
warn()
address = input("Enter your address:")
warn()
ccn = ("Type in your credit card number:")
warn()
expiration = ("Enter the expiration date for your card:")
```

- Notice that this looks much cleaner. It also makes it easier to make changes that need to happen everywhere. If you want to change the warning message, we only have to make one change, instead of hunting down every place we had the message and changing it in each of those locations. We can also easily expand our warning.

- The use of functions allows us to conceptually separate a piece of functionality from the rest of the code—that is, we can take some set of commands and pull them away from the rest of the code. So, we don't have to know about the rest of the code to use those commands, and the rest of the code doesn't need to know the details of those commands.

- You should aim to use functions any time you find that you're doing the same task in different areas of the code. The warn function is an example. Using a function not only means less typing, but it also means less of a chance that you'll have a bug, because any errors are going to appear every time, instead of just once, and once you fix the bug, it's fixed everywhere that program is used. In addition, any time you have a concept that you can consider as a single unit, a discrete idea, it's a good idea to use a function to encapsulate it.

› Encapsulating each discrete idea in a function might seem like a minor, or even unnecessary, thing to do with smaller programs, but with larger programs, the ability to break up and organize code is critical. Programming languages are meant to help people, and the main thing that abstraction does is let us control the mental complexity of any piece of code that we're dealing with at one time.

› Remember that functions are able to return values. For example, the input function returns whatever the person typed in. To return a value for a function, we just include a line that says "return" something. The following function gets a person's name as a string. It gets the user's first name, then last name, and then combines them with a space in between. It returns that combined name. When we call this function, we can assign the value it returns to a variable, as follows.

```
def getName():
    first = input("Enter your first name:")
    last = input("Enter your last name:")
    full_name = first + ' ' + last
    return full_name
name = getName()
```

› Let's try a variation on this. Starting from the code we had, let's make a small modification so that it returns in a "last name, first name" format.

```
def getName():
    first = input("Enter your first name:")
    last = input("Enter your last name:")
    full_name = last + ', ' + first
    return full_name
name = getName()
```

› We just changed the way we formed the string so that it was "last" plus "," plus "first."

› We could have even just returned that string right as we made it. Notice that the following returns the combined name directly, without putting it in another variable.

```
def getName():
    first = input("Enter your first name:")
    last = input("Enter your last name:")
    return last + ', ' + first
name = getName()
```

› Sometimes we might want to return more than one variable. The following example has us reading in the first and last name, and rather than returning one single combined string, we return two strings. Notice that in the function definition, when we return, we return two values. When we call the function, we need to provide two values for these to be stored into. In this case, the first string returned goes into "userfirst" and the second string returned goes into "userlast."

```
def getName():
    first = input("Enter your first name:")
    last = input("Enter your last name:")
    return first, last
userfirst, userlast = getName()
```

› Although abstraction does have many wonderful virtues, there are a few potential downsides. One downside might be that you could use a function more efficiently if you knew how it works. So, when people need to squeeze every last drop of efficiency out of a program, they will sometimes peek behind the veil of abstraction. But these cases are relatively rare. To a surprising degree, it's better to make use of abstraction when you can to make your code conceptually simpler.

› A second, more serious, downside that can occur with abstraction is a pitfall that we need to be especially mindful of when programming. Occasionally, functions will have what are called **side effects**. The problem here is that in the process of doing the main thing it's supposed to do, the thing it's advertised to do, the function also does something else.

› Sometimes, the side effect is something that the person writing the function thought would be harmless, and thus doesn't even advertise. Sometimes, this is a bug—some action the programmer never meant the function to perform. Usually, these side effects go unnoticed, at least for a while. But, eventually, they can cause serious problems when someone doesn't realize they're going to be there.

## [ DOCSTRINGS ]

› It's helpful to provide documentation for functions right up near where they are first defined, and there is a standard way to do it. The documentation should say, concisely and specifically, what that function does. There is even a special syntax used to provide these comments for a function, and it's called a **docstring**.

› A docstring begins and ends with a triple set of quotation marks. Triple quotation marks are how we create a string that can include new lines, thus spanning multiple lines. This string should come right after the function header.

› In the following case, we have a function named "greet" that will take in a name as a **parameter** and will print "Hello, name" for whatever name is passed in. So, we can create a docstring afterward, with three double quotation marks, saying "Print a greeting: Hello, name."

```
def greet(name):
    """Print a greeting: Hello, name. """
    print("Hello, "+name)
help(greet)

OUTPUT:
Help on function greet in module __main__:
greet(name)
    Print a greeting: Hello, name
```

› Docstrings are a lot like comments, in that they don't actually compute anything. But they have a second advantage. If you use the command "help" for any function, passing in the function name as a parameter, you will get information about the function, including its docstring.

## Readings

Gries, *Practical Programming*, chap. 3.

Matthes, *Python Crash Course*, chap. 8.

Sweigart, *Automate the Boring Stuff with Python*, chap. 3.

Zelle, *Python Programming*, chap. 6.

## Exercises

What would be the output of the following code?

1
```
def something1(a, b):
    for i in range(a):
        print(b,end='')
something1(5, 'X')
```

2
```
def something2(a, b):
    for i in range(a):
        b = b*b
    return b
print(something2(3,2))
```

3
```
def something3(a):
    return a-1, a+1
a, b = something3(5)
print(a, b)
```

4
```
def something4(a):
    sum = 0
    for b in a:
        if b < 0:
            sum -= b
        else:
            sum += b
    return sum
print(something4([2, -4, 3, -1, 7, -4]))
```

5
```
def something5(a):
    sum1 = 0
    sum2 = 0
    for i in range(len(a)):
        if i%2 == 0:
            sum1 += a[i]
        else:
            sum2 += a[i]
    return sum1, sum2
x, y = something5([1, 2, 3, 4, 5, 6])
print(x,y)
```

Write code for the following.

6 A function that takes in a number and a string and prints the string that many times.

7 A function that takes in two lists of the same length and returns a new list of that length, containing the smaller of the elements at that index value from the two lists.

8 A function that takes in three numbers and returns the one in the middle.

Simplify the following code using a function.

9
```
salary1 = float(input("Enter previous salary"))
benefits1 = float(input("Enter previous benefits"))
bonus1 = float(input("Enter previous bonus"))
salary2 = float(input("Enter new salary"))
benefits2 = float(input("Enter new benefits"))
bonus2 = float(input("Enter new bonus"))
if salary2 > salary1:
    salaryincrease = salary2 - salary1
else:
    salaryincrease = 0
if benefits2 > benefits1:
    benefitsincrease = benefits2 - benefits1
else:
    benefitsincrease = 0
if bonus2 > bonus1:
    bonusincrease = bonus2 - bonus1
else:
    bonusincrease = 0
```

TRANSCRIPT 09

# Functions and Abstraction

We're going to begin an exploration of functions. These are commands, or groups of commands, to get things done. They're like mini-programs that take in some input, perform some action, and return some output. Use of functions also demonstrates what I consider to be the most important idea in computer science: abstraction. Now, when we talk about abstraction, here's what we mean: we simplify all of the details away, and we view a complex system through a simpler interface. We say that we've abstracted away the complexity. This happens all over in programming and in computer science in general.

We've already seen abstraction when we talked early on about the three main parts of a computer: memory, processor, and input/output. The very fact that we could talk about those in such general terms is an example of abstraction. We can talk about memory as something that lets us store information, and we don't have to understand the physics of how this occurs. We don't have to understand the circuitry that makes up a processor to understand the basic commands it supports. And input/output? Wow, that's a pretty complex system of operations to get the signals from a keyboard into the computer and to take values and determine how to light up individual pixels on the screen to display the text. Well, we get to just abstract it into input and print statements.

The programming language itself is another place where abstraction is extremely important. We get to describe computer commands at a very high level that people can understand, instead of those detailed instructions that the computer understands.

Now, one of the main ways that we take advantage of abstraction when programming is through functions, and though you might not realize it, we've already been using functions. The print and the input commands are examples of functions. We have the name of a function, followed by parentheses that might or might not have something inside of them.

Now, the term function is not universal. Other names you'll hear are routine, subroutine, procedure, and you'll also hear method, although that's only used in the context of object-oriented programming, which is the way we'll use method ourselves later on in the course.

Now, sometimes people use the term function with a more specific meaning, much closer to the mathematical definition. In that definition, a function always returns a value, so for commands like print that don't return anything, we'd have to use a different term. But people discussing Python are usually more flexible, and all these terms have pretty much the same meaning. Just recognize that when people say function, routine, subroutine, procedure, or method, they're pretty much talking about the same idea.

Now, some functions, like the print function, just do something. When a program asks a function to do its thing, the term we use is calling the function. When you call the function, something happens, like text getting printed to the screen. Other functions, like the input function, not only perform some action, like printing to the screen, they also return a value. For the input function, the function returns a string that's whatever the person typed in. This return value can get assigned to a variable or whatever else is needed.

When we think of function calls, the typical way to think of them is as a black box. The function takes in some input, or not, then the function does something. We don't want to—in fact, we don't need to—think about how it does what it does. Then, it returns some output, or not. For the person using the function, it's a mysterious box that takes input, does its thing, and produces output.

Now, consider the perspective of someone creating a function. If you're creating a function, you get input from somewhere, you do your thing, and you send your output back out. You don't really care where the input is coming from or where your output is going. If you're writing the details of how the print statement would work, you don't need to know where that string came from or why the user wanted it printed, you just need to take whatever string is passed in and figure out how to print it

out to the screen. And, if you're writing the input command, you don't need to know how that string is going to get used, you just need to get the string from the user and pass it along as the output.

As you write the details of your code, you'll find yourself writing functions. You'll also find that within those functions, you'll make use of more function calls. Function calls are when you initiate some action that's been defined with a function. Calling print means that we're using the print function, and calling input means that we're using the input function, and you can treat these calls as black boxes of their own. You can think of this whole process like a series of layers. You don't need to worry about what's happening in the layers above you—where that input is coming from or where the output is going. You don't need to worry about the layers below you—how all of those function calls are working internally. You just need to worry about your own layer.

Another way we talk about this in programming is to describe this as managing complexity. We want to limit the complexity that we have to think about to just the concepts at the layer that we're working in. A programmer doesn't need to understand every level of complexity, at least not at the same time. Programmers only need to understand the details at the one level they're working in at the time.

Writing functions to work in Python has two different stages: defining the function, and then calling the function to put it into use. We've already been calling functions, so let's look now at how functions are defined.

We define a function by starting with the key term "def," short for define. Next, we have the name of our function—the command that we want to use to call the function. Following that are parentheses. If our function is going to take in some form of input, that information is going to be specified inside the parentheses. Then, we have a colon, just like we've seen in conditionals and loops. And finally, we have the commands that the function should do. These are indented in, again just like we saw with conditionals and loops. The term we use for that first line defining the function is called its header, and the actual commands that it should do—the part that's indented in—is called the body.

Suppose we want to create a function that just processes something and doesn't take any input, and doesn't return any values—maybe it just prints a particular warning message. Notice that we start with the word "def," short for definition. We then have a name for the function; in this case "warn." There's nothing inside the parentheses since this function doesn't take any input. After the colon, we indent the commands that we want. In this case, there's just one command, a print statement that displays a warning message.

Now, when we're writing code, we can actually call this function. We might have some code and in the middle of it, decide that we need to print out a warning message. We can do so by calling "warn," just like we would any other command. Notice that we need to have parentheses after the name of the function. When we execute this code, it works like you were hopefully expecting it to. The code before the function call works just like always, in this case printing the value 3. Then, when we get to the function call, it executes the function, in this case printing the warning message. After that, it executes the rest of the code, just like always.

Now, this might seem a little silly. I mean, couldn't we have just output that warning message directly? The code you see here works exactly the same way, and we didn't have to bother creating a function to do it. So, why do you think we'd want a function to do something like this? Let me illustrate one of the ways that functions can be useful.

Let's say that you have some code where you're asking users for sensitive information, and you want to give them plenty of warning that they're about to do something dangerous. You might print a warning each time before you ask for information like you see here. Well, this is a lot of repetition for the exact same thing, and when you're repeating the same commands over and over, this is often a good time to use a function. It's straightforward to convert the warning message to a function. We simply define a function, "warn," that prints the warning message. Then, we can replace every occurrence of the print statement with a call to "warn." Notice that this code looks a whole lot cleaner. It also makes it easier to make changes that need to happen everywhere.

Let's say that you wanted to change the warning message. Well, here's a different warning message—we've changed "Use program at your own risk" to "You are about to enter sensitive information." We needed to make only one change instead of hunting down every place we had the message and changing it in each of those locations.

We can also easily expand our warning. Here, we've augmented our "warn" function further. After printing our warning, we ask whether the person wants to continue. If the answer is "N" for no, then we quit the program entirely. Notice that to quit, we call a function named quit. That's a function provided in Python that we haven't seen before, and it's used in cases like this where we're ready to quit the program but are buried somewhere in the middle of the code.

The point here is that the use of functions has allowed us to conceptually separate a piece of functionality from the rest of the code. That is, we've been able to take some set of commands and pull them away from all the rest of the code. So, we don't have to know about the rest of the code to use those commands, and the rest of the code doesn't need to know the details of those commands.

Look at that same code again, where I've inserted some comment lines to break it up visually. If you look at the top part, the "warn" function, notice that we can write it while being completely oblivious to where it's being called. All we have to know is that we're giving out the warning message and asking to continue or not. There's no reason we need to know anything about the code at the bottom. And, likewise, for the code at the bottom, we don't need to know anything about how the warn command is working. All we need to know is that when we're ready to give out a warning, we make a call to "warn."

So, when should you aim to use functions? First, any time that you find that you're doing the same task in different places in the code, it's almost always good to use a function. The "warn" function that we were just looking at is an example. Using a function not only means less typing, it means less of a chance that you'll have a bug since any errors are going to appear every time, instead of just once. And, once you fix the bug,

it's fixed everywhere that program is used. Second, anytime you have a concept that you can consider as a single unit or a discrete idea, it's a good idea to use a function to encapsulate it.

For instance, say you're writing a program to manage your home business. Let's say that you have a program that lets you enter clients. It does some other stuff to figure out how much they owe you, and then it prints a bill. You have three different lists: one for the names, one for the addresses, and one for the balance owed by each client. There's a loop where we collect client information as long as there's, at least, one client. Then, at some point later, we loop through our client names, and for each one who has a positive balance, we print out a message about how much that person owes.

In a situation like this, we would want to identify what part of the code represented a single idea and pull it out into a function. For instance, the idea of getting information about a single client seems like a discrete task that we'd do, so we pull it out into a function, called "getClient." In this case, it's the exact same code we had before—we get the name and address from the user, and we append information onto the lists of client names, addresses, and balances. Where we had that code originally, we now have just a call to "getClient." We've made an abstraction here; we have the concept of getting client information. So, in our main code, when it's time to get client info, we don't worry about the details, we just call "getClient." And, our "getClient" routine doesn't worry about when it's called; it just does its own thing.

We could do the same thing with the invoice printing. When it's time to print an invoice, we can pull out the print statements from the main code into a routine called "printInvoice." Then, when we want to print, we just call "printInvoice."

Encapsulating each discrete idea in a function might seem like a minor or even unnecessary thing to do at the moment, but let me assure you, as we get to larger and larger programs, the ability to break up and organize our code like this is going to be critical. Remember, programming languages are meant to help people, and the main thing

that abstraction does is let us control the mental complexity of any piece of code that we're dealing with at one time. We've been looking at code that's gotten no bigger than about 50 or 60 lines so far. Imagine having a program with 10,000 lines of code. There'd be no way you could possibly understand all that code at the same time. But, if it's broken up into these bite-size pieces contained in functions, it's easy enough to understand any one piece. And, as we keep advancing in this course, we're going to be using functions more and more to separate concepts in code.

Let's go a little bit deeper into functions. Remember that functions are able to return values? For instance, the input function returns whatever the person typed in. To return a value for a function, we just include a line that says return something. Take a look at this function, which gets a person's name as a string. It gets the user's first name, then last name then combines them with the space in between, and it returns that combined name. When we call this function, we can assign the value it returns to a variable, like you see here.

Let's try a variation on this. Starting from the code we had, try making a small modification so that it returns in a last name comma first name format. Here's what that would look like. We just changed the way we formed the string, so that it was "last + ', ' + first." We could have even just returned that string right as we made it. Notice that we just returned the combined name directly without putting it in another variable.

Let's practice this a bit. Maybe you have a number game, and you need to ask a person to guess a number. How might we write a routine that asks someone for their guess and returns that value? Let's call the routine "getGuess."

Well, here's one way we could write it. We have a definition: def getGuess():. Then, indented from there, we have the body, which in this case is just two lines of code. In the first line, we get the guess from the user. In the second, we return that guess. OK, that's a pretty simple routine.

Sometimes, we might want to return more than one variable. This example has us reading in the first and last name, and rather than returning one single combined string, we return two strings. Notice that in the function definition, when we return, we return two values. When we call the function, we need to provide two values for these to be stored into. In this case, the first string returned goes into “userfirst,” the second string returned goes into “userlast.”

Now, as an aside, let me briefly open the black box about what it means to return multiple variables. This is not allowed in every other language. And, technically, Python is actually only returning one value. What’s getting returned is a tuple. Remember that a tuple can be written as a list of values separated by commas, inside of parentheses. In this example, we create a tuple with three values, corresponding to the first, middle, and last name for someone. We can get values from a tuple by assigning to a comma-separated list like you see here, where first, middle and last get the elements from the tuple. We can print the variables we were assigned, obviously. Or, we can access an individual element of a tuple using brackets, just like we did for lists.

So, when we’re returning multiple values, we’re actually returning a single tuple that contains those values, and then, when we assign that value, we’re assigning the tuple. So, if we assign the function output to a single variable, like in this case to the variable “name,” that variable will actually be a tuple. And, if we assign to multiple variables, what we’re doing is taking the tuple that we got back and then assigning the elements of the tuple to those variables. Either way is acceptable.

Say you want to create a function that asks a user to enter a birthday as a month, day, and year, and then returns that date from the function. Let’s call the function “getBirthday.” How might we write this function, and how would we call that function? OK, think about this as a tuple. After our function header, we can have a body in which we ask the user for the three pieces of information. Then, we can return those three items as a tuple. A call to this function could take a single result, which will be the tuple itself, or we could assign the result to three different elements: the month, day, and year.

Now, let's look at how to give input to functions. This is the information that we put inside the parentheses when we call a function. We call this information the parameters of the function. But how do we actually use this? We take in parameters by listing them in the function header, like you see here. In the function definition, we put inside the parentheses the name that the function is going to use for the parameter. In this case, we have a function that's going to sum up the numbers from one to whatever value is passed in as a parameter. We name that parameter *x* in this case. When we call the function, like you see below, we put in whatever particular value we want the parameter to take, which in this case is 100.

Oh, suppose we remember a more compact formula for summing up positive integers from one to *x*. Notice that we could replace the inner workings of the function with a simpler formula. It doesn't change anything as far as the code calling the function is concerned. In a sense, that change to the inner workings of the "getSum" function is hidden from the code calling the function. The "getSum" function is a black box as far as the main code is concerned; it just knows that it passes in a number, and it gets out the sum.

Let's create a similar function for multiplication. How about we write a function to compute factorials, 1 times 2 times 3, and so on? We'll pass in a number and then return that number's factorial. The factorial of *n* is the product of all the numbers from one to *n*, so how might we write this function?

We would start out with the function definition. We'd write "def factorial()," and then inside the parentheses, we'd put the name of the variable we want to use for the parameter that we want passed in. In this case, it's just a number, so we'll call it *n*. In the body of the function, we set the product to one, initially, and then we loop through all the numbers up to and including *n*. Notice that, like the summation, we have to set the range to $n + 1$, since we want to include *n* in the numbers we use. Each iteration of that loop will multiply the index by the product. And, finally, we return the product. Somewhere else in the code, then, we can call

the factorial function, passing in the number we want to compute the factorial of.

Now, we can actually have more than one parameter passed in if we'd like. In this case, we're going to sum up numbers in some range. We'll create a function "getRangeSum" that takes in two parameters. We'll call the first parameter *a* and the second one *b*. Notice that they're separated by a comma. When calling the function, we also have to pass in two parameters—the example shows a couple of examples. Again, the function "getSum" should look like a black box from the outside. So, if we wanted to change the interior of the function, we could do so, such as if we were using the summation formula. Notice that there's no change whatsoever to the code that's calling the function. The part of code calling the function can be completely oblivious to what's going on inside.

Or, here's another implementation, in which our current function makes use of the previous one. We can get the range sum by computing the sum from one to the ending number, and then subtracting the sum from one to just before the starting number. We can pick whichever implementation we like best when writing the "getRangeSum" function, and from the perspective of the code calling it, not a thing is changed.

That last example, by the way, might give some sense of how abstraction can help. We defined one function in terms of another function. When we write more complex programs, we'll often have a sort of function hierarchy, where one function calls some others, and each of those calls others, et cetera. So, any one function gets to ignore the stuff happening above or below it in the hierarchy.

Now, although abstraction does have many wonderful virtues, there are a few potential downsides. One downside might be that you could use a function more efficiently if you knew how it worked. So, when people need to squeeze every last drop of efficiency out of a program, they'll sometimes peek behind the veil of abstraction, but those cases are relatively rare. To a surprising degree, it's better to make use of abstraction when you can.

Let me give you a story. Early in my undergraduate years, I was working one summer as a part of a medium-energy physics experiment at a collider. Now that sounds more impressive than it actually was, but one of the cool things I got to do was sit in on group meetings and hear about what everyone was working on. There was one group of researchers that was focused on improving the rate of data acquisition in the experiment. They had written some code in a higher-level language, and I think it was C, but it ran slower than they wanted, and they became convinced that the problem was inefficiency due to the abstraction of using a higher-level programming language.

Well, for a good chunk of the summer, they were focused on rewriting some of that code at a lower level—assembly language—so that they could make use of the features of the processor and the memory, and maximize the performance of the code. Now, these were really smart people who knew what they were doing. Shortly before I left, they finished the code, and guess what? It ran slower than the higher-level code had. The high-level compiler had actually produced more optimal code than their careful attempt to remove the layers of abstraction. Now, there are several lessons that could be learned here, but the thing I want you to understand is that not only you but also every other programmer, are almost always going to be better off using abstraction to make your code conceptually simpler.

OK, there's a second, more serious, downside that can occur with abstraction, and this is a pitfall that we do need to be especially mindful of when programming. Occasionally, functions will have what are called "side effects," and yes, that's the technical term. The problem here is that in the process of doing the main thing that it's supposed to do, the thing that it's advertised to do, the function also does something else. Sometimes, the side effect is something that the person writing the function thought would be harmless and thus doesn't even advertise. Sometimes this is a bug, some action the programmer never meant the function to perform. Usually, these side effects will go unnoticed, at least for a while, but eventually, they can cause serious problems when someone doesn't realize that they're going to be there.

Let me show you one example to illustrate. Say we've created a function, "sumList," that will take in a list, and sum up all the values in the list. It then returns the sum. The way we wrote this function is to go through the list and add the previous value to the current value. So, every element of the list ends up having the sum of all the elements up to that point. And, finally, we just return the final value in that list.

Now, this routine actually works for what it was intended to do—it does return the sum of all the elements in the list. If we just call the function and output the result, it looks just fine. But, little did we know that in the process, it changed the values in the list—yikes. If we look at the list after calling, it's totally different than beforehand. That would be considered a side effect—the routine ended up doing something that it wasn't supposed to do.

Now, speaking of what functions are supposed to do, there's one very helpful practice I want to mention, and that has to do with documenting functions. It's helpful to provide documentation for functions right up near where they're first defined, and there is a standard way to do it. The documentation should say, concisely and specifically, what that function does. There's even a special syntax used to provide these comments for a function, and it's called a "docstring."

A docstring begins and ends with a triple set of quotation marks. You might remember that triple quotation marks are how we create a string that can include new lines, thus spanning multiple lines. This string should come right after the function header. In this case, we have a function named "greet" that will take in a name as a parameter, and will print, "Hello, name" for whatever name is passed in. So, we can create a docstring afterward, with three double-quotes, saying print a greeting, "Hello," name.

Docstrings are a lot like comments, in that they don't actually compute anything, but they have a second advantage. If you use the command "help" for any function, passing in the function name as a parameter, you will get information about the function, including its docstring. So, in our example of the greet function, when we have the command

"help(greet)," we'll get information about the greet function, including the docstring that describes what it does.

Now, let me remind you of some of the built-in functions that we've already been using in Python already. We've used "print" and "input," each of which takes a string that will be printed on the screen, and input also returns a value. We've used the "int," "float," and "str" commands to convert between strings and number types, and each of these are functions. We've used "open" to open files. We've used "len" to find the length of a list. We've used "range" to get a set of values to look at within our loops. There are a variety of other useful functions built in to Python, such as "slice," "list," and "tuple," and we'll introduce more as we continue through the course.

Here are a couple of examples. The "sum" function takes in a list as a parameter. It sums up all the elements of the list and returns that sum. So, if we pass in a list with elements 1, 2, 3, 4, and 5, we'll get the value 15 returned. Now, we've seen ways to write our own sum routines, and there are times that we'll have complex data that we'll still want to manage that process ourselves, but for simple cases where we just want to add up a bunch of values, the sum function works great.

Similarly, there are functions "min" and "max" that will return the minimum and maximum of a list. We can pass in a list in any order, and the functions will return a minimum and maximum. There are also functions called "methods" that are a part of objects, as we'll discuss later. You've already seen these briefly—they're the functions that come after a variable name and a period, as in the "file.close()" function. And, besides all the built-in functions, there's a vast wealth of functions that can be called from external modules and packages—really, from any Python program whatsoever.

Functions provide us one of the most direct examples of abstraction in programming. Abstraction remains probably the most powerful idea in all of computer science. Look for it; aim for it. Good programmers are intensely aware of abstraction and make use of it repeatedly in different forms. The functions that we've seen here are a very direct way to put

abstraction into practice, but functions have a whole lot more to them, in the way we handle parameters, for instance. In the next lecture, we'll explore parameters in functions, so that we can use them more effectively in all of our programs.

## MORE PYTHON PRACTICE

Let's get some practice writing a function. Try writing a function that takes in a string and a letter as parameters, and then counts how many times that letter appears in the string. Be sure you write code that lets you test your routine. Here's a hint: remember that strings can be thought of as lists of characters.

OK, here's what your code might look like. We have a function called "countChar." It takes in two parameters, the first one being the character that we're wanting to count, "ch," and the second one being the string that we want to count in, "teststring." Inside the routine, we're going to keep a count of how many times we encounter the character, so we start with zero. We'll loop over all the characters of the string with the "for" loop, and for each character, we'll check whether it's equal to the one that we're searching for or not, and we'll increment count each time we find a match. Notice that we return count at the end.

That's a pretty basic routine, but it illustrates the main aspects of a function. There are input parameters passed in, and a value is returned, and, just as importantly, we've separated a single conceptual idea—counting characters in a string—into its own section of code.

10

# Parameter Passing, Scope, and Mutable Data

Functions are some of the most powerful tools in programming, but to use them as widely and fully as possible, you need to understand the details of how data is handled within a function—which is what you will learn in this lecture. You will also learn about the key ideas of when a parameter or variable is "in scope," how to work with data types that are mutable, and what it means for parameters to have default values.

## [ SCOPE OF VARIABLES ]

- When we say that a variable is "in scope," it means that at that point in the program, it is defined and usable. Scope is what helps keep straight what we are referring to when we use a name for something. It is an important way that we make use of abstraction. The bigger our programs become, the more important scope becomes.

- Remember that functions help us conceptually simplify our task. When we are defining a function, we don't have to worry about how all that stuff outside is working. We just know that we can have some stuff coming in as parameters, and we can end up returning something. But the stuff in the middle is just there for our function; it isn't meant to affect all that stuff outside. Scope helps us keep stuff where it belongs.

- There are two "sides" every time we make use of a function: the function definition, where our code describes what the function will do, and the function call, where a function is used.

- The following is a simple function that returns the maximum of two values. Python already includes a function named "max" that does exactly this. It takes in two values and returns the one that is larger.

```
def maxdemo(val1, val2):
    if (val1 > val2):
        return val1
    else:
        return val2
a = 1
b = 2
c = maxdemo(a,b)
```

- But let's write our own max function because that will provide a good way to illustrate the tricky aspects of functions that are hidden from view in the preloaded functions.

- A function is defined by the header and the body. The header starts with the keyword "def," followed by the function name, followed by a list of parameters in parentheses and a colon. Then, we indent the body, and we can exit the body by returning some value.

- When we run our program, the computer comes along and sees the function definition. It doesn't do anything with it at that point, other than remember that it's a function with this name and these parameters, and then later if that function is called, it's going to remember that definition and use it.

- Next, we have what's called the **main program**. All of our code together is called our program, but we'll call the "main program" the stuff that actually gets executed first—that isn't part of a function definition. As our program gets processed, the first line that gets executed is a line from the main program.

- To understand how functions are really working, let's open the black box and look at how memory is getting handled. The computer skips over the function at first, just remembering that it was previously defined, and executes the first line of the main program. The line *a* = 1 creates a variable, named *a*, and assigns it the value 1.

- Then, "b = 2" creates the variable *b* and assigns it the value 2.

- In the next line, we have a function call. We call the function "max," and we pass in two parameters: *a* and *b*. When we call the function, there is a whole new area of memory set aside for that function to work in. This is called the **function activation record**.

- In that area of memory, we will first have variables created for the parameters. In this case, we have "val1" and "val2" variables created in the function activation record. As far as that area of memory is concerned, these variables are named according to the parameters in the function, not according to the names they originally had. These parameters inside the function are sometimes named the local parameters, to distinguish them from the parameters on the caller's side—that is, the ones that are listed in the "main" program.

- So, when the line "c = maxdemo (a,b)" is called, the values of the parameters on the caller's side are copied into the new memory locations, the local parameters. In this case, the value of *a*, which is 1, is copied into val1, and the value of *b*, which is 2, is copied into val2.

- The function then runs in its own little memory area, and when something is finally returned, that is sent back out to the "main" program memory area.

- After that, the memory for that function is all freed up. The activation record is destroyed, leaving that memory free to use again in the future. And our program goes on to the next line of code from the main part of the program.

› Let's look at how names of variables are handled in a function. The following is another simple function, named "testscore." The function computes a test score as a percentage, given the number of correct answers and the total number of questions.

```
def testscore(numcorrect, total):
    numcorrect += 5;
    temp_value = numcorrect / total
    return (temp_value * 100)
a = 12
b = 20
c = testscore(a,b)
```

› The testscore function takes in two parameters: the number correct and the total number. Let's say that everyone did well on a pretest, so you're giving everyone credit on the test for 5 extra questions. So, the function first adds 5 to the number of correct answers, then it divides by the total, and returns that answer, multiplied by 100.

› From the main program, we have variables *a* and *b*, and then when the function call is reached, we get a function activation record, where we have the parameter values copied into the parameters. So, *a* is copied into the local variable "numcorrect" and *b* is copied into the local variable "total."

```
def testscore(numcorrect, total):
    numcorrect += 5;
    temp_value = numcorrect / total
    return (temp_value * 100)
a = 12
b = 20
c = testscore(a,b)
```

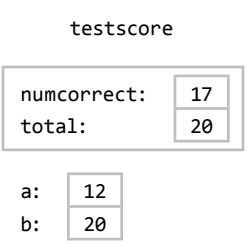
testscore

| | |
|---|---|
| numcorrect: | 12 |
| total: | 20 |

| | |
|---|---|
| a: | 12 |
| b: | 20 |

› What happens when we get to the next line, where the "numcorrect" value is increased by 5? The variable "numcorrect" in the function activation record is increased by 5, so in this case, it becomes 17.

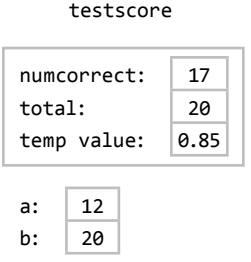
```
def testscore(numcorrect, total):
    numcorrect += 5;
    temp_value = numcorrect / total
    return (temp_value * 100)
a = 12
b = 20
c = testscore(a,b)
```




› Notice that we do not change the value of the variable *a*. The value of *a* was copied into the memory location for "numcorrect," and when we change "numcorrect," we are changing only the new memory location. As far as the computer is concerned, it no longer cares that "numcorrect" got its value from *a*; that value could have come from anywhere. It's now just "numcorrect," so whatever happens there doesn't affect the input parameters.

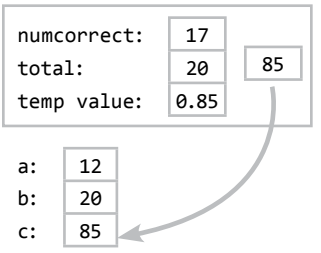
› The next line of code in the function creates a new variable, "temp_value," which gets the ratio of "numcorrect" and "total." This is a new variable, so it has memory set aside for it. The memory that is set aside is set aside within the function activation record. We create a variable named "temp_value" there, and we assign the value to that variable.

```
def testscore(numcorrect, total):
    numcorrect += 5;
    temp_value = numcorrect / total
    return (temp_value * 100)
a = 12
b = 20
c = testscore(a,b)
```




› Finally, we return a value back to the main program. We first compute a new value by multiplying "temp_value" by 100 in the function itself. This calculation doesn't get assigned to any variable, so it's just sitting temporarily in some unnamed memory. Then, that value gets passed back to the main program as the result of the function. In this case, the value is 85, so we assign 85 to the variable *c*.

```
def testscore(numcorrect, total):
    numcorrect += 5;
    temp_value = numcorrect / total
    return (temp_value * 100)
a = 12
b = 20
c = testscore(a,b)
```




- Then, because we're done with the function call, the function activation record is destroyed. We go on to the next line of code.

- What should happen if we tried to pull one of the values out of the function? If we tried to access "numcorrect," one of the parameters from the function call, or maybe the "temp_value" variable, neither of these is allowed. There's no more function activation record; all that memory was freed up. There's no way to get those values, even if we wanted to. As far as the main part of the program is concerned, those things inside the function never existed—the main program has no way of using them.

- In this case, the function parameters "numcorrect" and "total" have a scope of just the function itself. And the variable "temp_value" has a scope from the point it is defined until the end of that function. We would say that any of these are "out of scope" in the main program.

- Sometimes we need to access something outside the function—for example, to modify a variable in the main program. We couldn't pass that in as a parameter, because it would just copy the value. We couldn't just use the name of that variable because it would create a new, local copy in the function activation record. So, how can we modify something that's not passed in as a parameter?

- The solution is something called **global variables**. When we declare a global variable in the function, it means that within that function, when we refer to that variable, we are referring to the exact same variable as in the main program. So, when we set the value of a variable in the function, we change its value in the main program. In general, using global declarations is discouraged.

## [ MUTABLE AND IMMUTABLE VARIABLES ]

- Languages differ in how scope works and the way that parameters are passed. In Python, when we call a function, we make a copy of the values from the main function. Then, when we change the local parameters, there's no change to the values in the calling function.

- This lack of effect on the calling function is usually referred to as "pass by value," which means that we only transmit the value of a parameter when we make the function call. In other languages, such as C++, there is also something called "pass by reference," which means that the local parameter is exactly the same as the calling parameter—not just the same value, but the actual same memory location. So, if we change the local parameter, we do get a different effect in the global parameter.

- In Python, we always pass by value, although there are ways to affect the parameters outside. Global variables are one way. Another way to get an effect that seems a lot like pass by reference is to use mutable variables. Mutable variables bring us to the surprising fact that there are times that we can modify the value on the calling side, through a function.

- We can consider variables as being one of two types: **mutable** or immutable. A mutable variable roughly means that the variable is changeable when passed as a parameter, and an immutable one is one that can't change when passed as a parameter.

- Most of our basic variable types are immutable—for example, an int, a float, or a string. That means that when we pass it as a function parameter, the original value can't change. We just copy the value into the new memory location that's part of the function activation record. The immutable value stays the same on the calling side.

- However, there are also mutable data types, and one mutable type is a list. Because lists are mutable, when we pass them as a parameter, the value in the list can actually change.

› In a practical sense, we mainly need to understand whether the things we're working with are mutable or immutable.

## [ DEFAULT PARAMETERS ]

› We don't always have to specify each of our parameters. Basically, in the parameter list, we give a default value for the parameter that can be used if the caller doesn't specify it.

› Let's consider a function to calculate miles a car travels, given a number of gallons of gas and the miles per gallon it gets. Notice that for miles per gallon (mpg), we are setting a default value. We follow the parameter name with an equal sign and then the value that parameter should take if it's not specified. This gives us a few different ways to call the function.

```
def calc_miles(gallons, mpg=20.0):
    return gallons*mpg
print( calc_miles(10.0, 15.0) )
print( calc_miles(10.0) )

OUTPUT:
150.0
200.0
```

› In the first case, we can call it just like always, where we specify both parameters. In this case, we have 10 gallons and are getting 15 miles per gallon, so we have 150 miles. There's no difference in the way we call the function in that case, and it didn't matter that we had the default value.

› However, we also have the option to leave off the parameter that has a default value. In the second case, we specify just one parameter, which will be the first one, gallons. The other parameter, mpg, is determined from the default value, which in this case is 20. So, we get 200 miles total.

› We can extend this to any number of parameters with default values.

- When you're specifying parameters by listing the specific local parameter and an equal sign, it's called a **keyword argument**. For this kind of parameter, you specify by position first, and then give any keyword arguments after that.

- Default values can get tricky if you use them for mutable data or define them to equal a variable. It's recommended that you only use default values if the default value will be a fixed, immutable value.

## Readings

Gries, *Practical Programming*, chap. 3.

Matthes, *Python Crash Course*, chap. 8.

Sweigart, *Automate the Boring Stuff with Python*, chap. 3.

Zelle, *Python Programming*, chap. 6.

## Exercises

What would be the output of the following code?

```
1   def something1(a):
        a = 0
    b=3
    something1(b)
    print(b)
```

```
2   def something2(a):
        a[0] = 0
    b=[1,2,3]
    something2(b)
    print(b)
```

3
```
def something3(a, b=2, c=3, d=4):
    return a + b + c + d
val = something3(3, 10, d=5)
print(val)
```

4
```
def something4():
    a = 3
a = 2
something4()
print(a)
```

5
```
def something5():
    global a
    a = 3
a = 2
something5()
print(a)
```

6
```
def something6():
    a[0] = 0
a = [1, 2, 3]
something6()
print(a)
```

7
```
def something7(a, b):
    print (a, b)
a = 1
b = 2
something7(b, a)
```

Write code for the following.

8 A function that increases all the elements of a list by 1.

9 A function that multiplies anywhere from 1 to 4 parameters together, returning the product of those numbers.

TRANSCRIPT 10

# Parameter Passing, Scope, and Mutable Data

Functions are some of the most powerful tools in all of programming, in part because functions help us break our code up into manageable pieces. And the more code we write, the more we're going to want to use functions so that we don't end up with gigantically long strings of commands. But in order to use functions as widely and fully as possible—for instance, if we want to pass a list into a function—we'll want to make sure that we understand how data is handled within a function. There are several tricky but important details that can have a huge effect on how functions interact with the rest of your code.

Most of the difficulties that can arise with functions have to do with the parameters and the variables within the function. Remember, parameters are the main way that we get information into the function. They're the things we specify inside parentheses. Parameters are the values that get passed in from the function call while parameters are seen from within the function itself as variables.

So, we're going to spend some time going through the details of how functions really handle parameters and data. Along the way, we'll learn about the key ideas of when a parameter or variable is in scope, how to work with data types that are mutable, and what it means for parameters to have default values. Some of these topics that we'll be discussing here are often seen as some of the most challenging parts of programming to grasp. As you gain experience with programming, this will become more familiar and easier to deal with, but it's often a challenge at first.

When we say a variable is in scope, it means that at that point in the program, it's defined and usable. I'm sure you know two people who have the same first name. Have you ever found yourself in a situation where there's been some confusion because of that? You have to say,

“Oh, I don’t mean your brother Joe, I’m talking about Joe from work.” Normally, it’ll be clear from the situation you’re in which Joe you’re referring to, but sometimes it can get confusing. Well, this same thing can happen to us with functions. Scope is what helps us keep straight what we’re referring to when we use a name.

Now, scope is an important way that we make use of abstraction. The bigger our programs, the more important scope becomes. Remember that functions help us conceptually simplify our task. When we’re defining a function, we don’t have to worry about how all that stuff outside is working, we just know that we can have some stuff coming in as parameters, and we can end up returning something. But the stuff in the middle is just there for our function; it isn’t meant to affect all that stuff outside. Scope helps us keep stuff where it belongs.

Now, there are two sides every time that we want to make use of a function: the function definition, where our code describes what the function will do; and the function call, where a function is used. Here’s a simple function that returns the maximum of two values. Python already includes a function named “max” that does exactly this—it takes in two values and returns the one that is larger—but let’s write our own max function since that will provide a good way to illustrate the tricky aspects of functions that are hidden from view in the preloaded functions.

A function is defined by the header and the body. The header starts with the keyword “def,” followed by the function name, followed by a list of parameters in parentheses, and a colon. Then, we indent the body, and we can exit the body by returning some value. When we run our program, the computer comes along, and it sees the function definition. It doesn’t actually do anything with it at that point, other than remember, “Oh, here’s a function with this name and these parameters,” and then later on, if that function is called, it’s going to remember that definition and use it. Second, we have what’s called the main program. All of our code together is called our program, but we’ll call the main program the stuff that actually gets executed first that isn’t part of a function definition. As our program gets processed, the first line that actually gets executed is a line from the main program.

To understand how functions are really working, let's open the black box and look at how memory is getting handled. Again, the computer skips over the function at first—just remembering that it was previously defined—and executes the first line of the main program. The line *a* = 1 creates a variable, named *a*, and assigns it the value 1. And then, *b* = 2 creates the variable *b* and assigns it 2.

Now, in the next line we have a function call, and here's where it gets interesting. We call the function "max," and we pass in two parameters, *a* and *b*. When we call the function, there's a whole new area of memory set aside for that function to work in. This is called the function activation record. In that area of memory, we'll first have variables created for the parameters. In this case, we have val1 and val2 variables created in the function activation record. As far as that area of memory is concerned, these variables are named according to the parameters in the function, not according to the names they originally had. These parameters inside the function are sometimes named the local parameters, to distinguish them from the parameters on the caller's side—that is, the ones that are listed in the main program.

So, when that line, c = maxdemo(a,b) is called, the values of the parameters on the caller's side are copied into new memory locations, the local parameters. In this case, the value of *a*, which is 1, is copied into val1, and the value of *b*, which is 2, is copied into val2.

The function then runs in its own little memory area, and when something is finally returned, that's sent back out to the main program memory area. After that, the memory for that function is all freed up. The activation record is destroyed, leaving that memory free to use again in the future, and our program goes on to the next line of code from the main part of the program. It's important to understand how that memory works since it's going to play a big role in understanding how parameters and variables are dealt with in a function.

Let's look first of all at how names of variables are handled in a function. Here's another simple function, named "testscore." The function computes a test score as a percentage, given the number of correct

answers and the total number of questions. We've got our function named "testscore," and it takes in two parameters: the number correct, and the total number. Let's say everyone did well on a pretest, and so I'm giving everyone extra credit on the test for five questions. So, the function first adds 5 to the number of correct answers, then it divides by the total, and returns that answer multiplied by 100. Remember what we just saw about how memory is handled? From the main program, we have variables *a* and *b*, and then when the function call is reached, we get a function activation record, where we have the parameter values copied into the parameters. So, *a* is copied into the local variable numcorrect, and *b* is copied into the local variable total.

Now, what happens when we get to this next line, where the value of numcorrect is increased by 5? The variable numcorrect in the function activation record is increased by 5, so in this case, it becomes 17. Notice that we do not change the value of the variable *a*. The value of *a* was copied into numcorrect's memory location, and when we change numcorrect, we're changing only the new memory location. As far as the computer is concerned, it no longer cares that numcorrect got its value from *a*—that value could have come from anywhere. It's now just numcorrect, and so whatever happens there doesn't affect the input parameters.

OK, the next line of code in the function creates a new variable, temp_value, that gets the ratio of numcorrect and total. This is a brand new variable, so it has memory set aside for it. Now, this is important—the memory that is set aside is set aside within the function activation record. We create a variable named temp_value there, and we assign the value to that variable.

Finally, we return a value back to the main program. We first compute a new value by multiplying temp_value by 100 in the function itself. This calculation doesn't get assigned to any variable, so it's just sitting temporarily in some unnamed memory. Then, that value gets passed back to the main program as the result of the function. In this case, the value is 85, so we assign 85 to the variable *c*. And then, since we're all done with the function call, the function activation record is destroyed. We go on to the next line of code.

So, what should happen if we tried to pull in one of the values from the function? Say we tried to access numcorrect, one of the parameters from the function call. Or maybe we tried to access the temp_value variable? Well, neither of these are allowed. Look at the memory diagram; there's no more function activation record—all that memory was freed up. There's no way to get those values even if we wanted to. As far as the main part of the program is concerned, those things inside the function never existed. The main program has no way of using them.

In this case, the function parameters numcorrect and total have a scope of just the function itself, and the variable temp_value has a scope from the point it's defined until the end of that function. We would say that any of these—numcorrect, total, or temp_value—are out of scope in the main program. Now, scope is an important concept in understanding how functions work.

OK, let's look at how this can get a little bit trickier. This is the same function we were just dealing with; notice what I'm about to change. I've changed the name of the variable inside the function from temp_value to *a*. What do you think happens here, specifically to the value of *a* in the main program? Does it change?

Let's walk through the code and make sure that we're following it all. We go through the first few lines just like before. We create variables *a* and *b* and assign values to them. We call the function, temporarily creating a function activation record, and we copy the parameter values from the call in the main program into the local parameters. We increase the value of one of these local parameters by 5. And now we get to our next line, the one where we're assigning a value to *a*.

Let me explain what happens first, and then tell you why it's so important. When we come to this line, we're trying to assign to a variable that hasn't been created in the function activation record. So, we'll create a new variable, *a*, within the function activation record, and that's the variable that gets the new value. Notice that the original variable *a* in the main program remains untouched. We have a new variable that's defined within the function, and when we hit the next line where we compute

with *a*, we use the value of *a* from the function we're in, not the value from the calling program. When the function call ends, and the function activation record is destroyed, we still have the exact same situation as we did the first time we ran this program. Basically, it didn't make any difference that the variable inside the function was named *a* instead of temp_value.

Now, the reason why this is critically important is that it helps us abstract away parts of the program and focus our attention. We can write the function without worrying about whether the variables we use inside are somehow going to bother the stuff outside. If we've got a large program, we can't remember every instance of every variable that we might have created everywhere else in the program. To keep ourselves sane, we need to be able to consider only tiny chunks of code at a time.

So, let's look at a couple more places that scope plays a role. Let's go back to our program that we've had before, where we use the term temp_value in the "testscore" function. We're going to make one change: instead of increasing the value of numcorrect by 5, we'll increase its value by *a*. What do you think happens in this case?

OK, let's look at the memory here. We get to the function call just like before. Now, when we get to this next line, notice that we're referring to this variable, *a*. We don't have *a* defined within the function activation record, so in this case, we actually go back to the most recently defined version of *a*, which is in the main routine. So, numcorrect gets increased by 12, so that it has the value 24. Notice that the variable *a* was still in scope inside the function for reading, but if we tried to assign something to *a*, like we saw earlier, we'd be creating a whole new variable.

Now, this is not really a good way to write code. When you look at the code for this function all by itself, it's not possible to really figure out what it's doing—you don't know what *a* is, or where it came from. The function doesn't stand on its own. Remember that the whole reason we write functions, the whole reason we use abstraction, is to reduce the conceptual complexity. Well, here we haven't really reduced complexity that much, since we still have something outside the function that affects

it, and it's not contained in our parameter list. The right way to approach this problem would be to pass *a* in as a parameter, and it would be even better to choose a name other than *a*.

Now, sometimes you really do need to access something outside the function—maybe you want to modify a variable in the main program, for instance. You couldn't pass that in as a parameter since it would just copy the value; you couldn't just use the name of that variable since it would create a new local copy in the function activation record. So, how can you actually modify something that's not passed in as a parameter? The solution is something called global variables.

First, from what we've learned, what would the following code print? It would print zero. The variable fuellevel inside of the function would create a new local variable called fuellevel. So, when the "initialize" function sets fuellevel to 100, it doesn't affect fuellevel in the main program. But, if we really wanted a routine that did this, that initialized the fuellevel to some particular value, wouldn't it be nice to have a function that would let us do that?

Well, we can do this by declaring fuellevel to be a global variable within the function. When we declare a global variable in the function, it means that within that function, when we refer to that variable, we're referring to the exact same variable as in the main program. So, in the code you see here, when we set the value of fuellevel to 100, we actually do change its value in the main program.

Now, using global declarations is generally discouraged. When you use globals like this, you end up with a function that's affecting things outside it, but there's nothing obvious in the call that tells you that it's doing so. However, there are times, like when you want to initialize values, or you want to read in data and set up variables, that using global variables is very useful, and it can actually aid in abstraction.

Now, languages differ in how scope works and the way that parameters are passed. In Python, when we call a function, we make a copy of the values from the main function. Then, when we change the local

parameters, there's no change to the values in the calling function. Well, this lack of effect on the calling function is usually referred to as pass by value. It means that we only transmit the value of a parameter when we make the function call. In other languages, such as C++, there's also something called pass by reference. Pass by reference means that the local parameter is exactly the same as the calling parameter—not just the same value, but the actual same memory location. So, if we change the local parameter, in that case, we do get a different effect on the global parameter. In Python, we always pass by value, although there are some ways to affect the parameters outside. Global variables, like we just saw, are one way. Another way to get an effect that seems a whole lot like pass by reference is to use what's called mutable variables.

Mutable variables, which I'll explain now, bring us to the surprising fact that there are times that we can modify the value on the calling side, through a function. We can consider variables as being one of two types, mutable or immutable. As you might guess, a mutable variable roughly means that the variable is changeable when passed as a parameter, and an immutable one is one that can't change when passed as a parameter.

Most of our basic variable types are immutable. If we have an int, or a float, or a string, that's going to be immutable. That means that when we pass it as a function parameter, the original value can't change—we just copy the value into the new memory location that's part of the function activation record. The immutable value stays the same on the calling side, and that's what we've seen in our examples so far. However, there are also mutable data types, and one mutable data type we've already seen is a list. Because lists are mutable, when we pass them as a parameter, the value in the list can actually change. Let's see how this works.

First, imagine we have a function, "adder," that takes two parameters and adds the second one to the first one. It doesn't even return a value, it just takes the first parameter and adds on the second one, then it's done. So, what will be printed here? Well, the integers don't change when we call the function, so printing num1 still gives 3. Remember that we make a copy of the values when we call the function, and those are

what's used in the function itself. If we used floats, we'd get the same results; and the same thing if we used strings. Notice that in all those cases nothing changes to the parameters on the calling side—they're immutable data types.

Now, let's see what happens if we try a mutable value instead. We'll pass in as parameters a couple of lists. What will happen? In this case, the mutable variable means that we can actually change the value within the function. So, in this case, the end result is that the list has actually changed. Our list went from just 1, 2 to 1, 2, 3, 4.

In a practical sense, we mainly need to understand whether the things we're working with are mutable or immutable. If it's immutable, it means the function can't change it, but if it's mutable, the function can actually change the value: ints, floats, and strings are immutable, and lists are mutable. As we learn about more types of data, we'll have to understand if they're mutable or immutable. The objects that we'll introduce later on are mutable, for instance, and if we ever pass something mutable to a function as a parameter, we need to be aware that it might change. We can't assume that the parameters that we pass in will remain unchanged by a function call unless they're immutable.

OK, so let me explain a bit more about how all of this is actually working. I did say that Python was passing by value, so how could we actually change something? When we have something mutable, it might seem like we're making the local parameter exactly the same as the calling parameter, but that's not really the case. Instead, we really need to understand what's going on in memory for all of this data. Let me show you a slight variation on what we just saw. We're going to change the function so that it assigns a new list to val1. It's not just adding something on, like in the previous case, it's actually assigning something different. What do you think will happen?

OK, let's see. Whoa, what on earth is happening? I thought we just learned that lists were mutable, and here I am trying to change something, and it does not actually change—I got the same list I started out with. The function didn't do anything.

To understand this, we need to understand that a mutable data type is really a reference to data values. That reference, which is called a pointer in some other languages, is what's really being copied. When you have one of these mutable data types, this reference is automatically de-referenced when you're dealing with the type, so it seems as though you're just dealing with the data that it refers to.

Let's look at a list, which can help everything make more sense. Let's say that we have a list named list1, and it's got a couple of elements in it. The actual variable, list1—that is, the box in memory that is named list1—does not actually have all the elements of the list in it. Instead, the box holds where the list is going to start in memory. It says, "Here's where the list of stuff will be." The actual elements of the list will be in a different part of memory, and the box—the variable itself—is just saying where this stuff will be held.

Now, if we come along, and we create a new variable, list2, and we set it equal to list1, what it does is get a copy of what's in list1's box. But, what's in that box is just the info saying, "This is where our list is; it starts here." So, list2 is going to be pointing to the same list as list1 is; it's not just a copy of the list elements, it's the exact same elements.

Notice what happens now if I make a change to list2 by appending on one more item—it's going to actually change the list itself. We didn't create a brand new list, we just added onto the existing list. So, since both list1 and list2 are using that exact same list, this changes both lists. If we were to print out list1 at this point, we'd see the values 3, 5, 7.

Now, what if we did something different, like assigned a whole new list to list2, like you see here? Notice now that we're not just changing the list—we're not taking what's already there and making a change. Instead, we're assigning a whole new value to list2. So, that's changing what's in the actual box belonging to list2. List2 is now going to have a whole new value; it has a totally different list that it points to. If we print out list1 at this point, it's going to be the same as before—list1 didn't change when we assigned a new value to list2.

So, turning back to our parameter passing, let's review why even these mutable things are not really any different than the immutable ones. If we look back at our previous example, when we call the function, we copy the reference to the lists into the equivalent area in the function activation record. But, they're still referring to the same actual list of elements as in the calling parameters. So, when we make a change, like adding val2 onto val1, we get a change that not only affects val1 in the function but also affects the list as seen by the calling side. After the function finishes, those changes it's made will persist.

On the other hand, if we go back to our other example, where we set val1 to a new list, we get different behavior. We start out the same way when we call the function, and the parameters are passed in. When we assign a new list to val1, we actually end up with a new list created, with no change to the old one. So, when we leave the routine and print out the calling variable, we see that it was unchanged by the function.

I realize that this can be a lot to absorb at once. But, once you understand exactly why certain things are mutable and others are immutable, it'll be easy to identify them as mutable or immutable, and understand exactly how they'll be affected by a function. It'll make sense why we say that we're always passing by value into a function—we're always making a copy of the value. It's just that, sometimes, that copy of a value is just a reference to something else in memory, and we can still change that other thing that we're referring to.

Let's look quickly at one more example to make sure that we've got this down. What would be output from the following code? Well, list *a* gets changed by the function since the function modifies what was in the list. List *b* is not changed by the function since the function tries to assign a completely new value to it. Whenever we see data types that are mutable, it'll help to remember that they're behaving like this underneath.

All right, we've spent a lot of time in this lecture talking about parameter passing and how all of that is done. Since parameter passing is central to how functions are used, it's critical to understand how this parameter passing is done. There's one more aspect of parameter passing that I

want to discuss, though, and that's default parameters. We don't always have to specify each of our parameters. Basically, in the parameter list, we give a default value for the parameter that can be used if the caller doesn't specify it.

Let's consider a function to calculate miles a car travels given a number of gallons of gas and the miles per gallon it gets. Notice that for mpg, we are setting a default value. We follow the parameter name with an equal sign and then the value that parameter should take if it's not specified. This gives us a couple of different ways to call the function. First, we can call it just like always, where we specify both parameters: in this case, we have 10 gallons and are getting 15 miles per gallon, so we have 150 miles. There's no difference in the way the call in that function works in that case, and it didn't matter that we had the default value. However, we also have the option to leave off the parameter that has a default value. In the case you see here, we specify just one parameter, which will be the first one: gallons. The other parameter, mpg, is determined from the default value, which in this case is 20. So, we get 200 miles total.

We can extend this to any number of parameters with default values. Here's a "madlibs" function with four parameters, all with default values. We specify an animal, adjective, city, and food, and we get back some crazy sentence incorporating all of them. We could call this function with any number of parameters, from 0 to 4. Notice that if we specify one parameter, it's always the leftmost, or animal in this case. If we specify two parameters, it's always the two leftmost, or animal and adjective in this case, and so forth. So, when you're specifying parameters, you have to realize that they're getting mapped in order from left to right.

But suppose you want to specify some parameter that's not the leftmost one? Well, you can specify the parameter you want in the calling function. Let's look back at our "madlib" function again. For example, here we set the value of food to be pizza. The other variables—animal, adjective, and city—will all still get their default values. When you're specifying parameters by listing the specific local parameter and an equal sign, like in this example, it's called a keyword argument. For this kind of parameter, you specify by position first, and then give any

keyword arguments after that. So, for this example, we're specifying animal by listing it first, without giving the keyword. It's the leftmost argument, so it will get mapped to the first argument: animal. Then we specify food and city using keyword arguments. Adjective gets its value from the default.

So, in the end, in this case, the parameter values are cat for animal, green for adjective, Denver for city, and steak for food. So, this example shows a little of everything. We have a parameter value set by position—that's animal, we have parameters specified by keyword—that's food and city, and we have a parameter value set by default—that's adjective. As a word of warning, default values can get tricky if you use them for mutable data, or define them to equal a variable. I'd recommend that you only use default values if the default will be a fixed, immutable value.

We've seen here that there is a variety of ways to handle data in functions, from global variables, to mutable and immutable parameters, to default parameters. Understanding this will be helpful as functions take on an increasingly large part of our programming repertoire.

My key piece of practical advice is that you should try to keep things as simple as you can to still do what you need to. So, when you write a function and think you need lots of parameters with default values or mutable data and are using values from outside the function so that scope is a big issue, first stop and ask yourself, "Is this really necessary?" Often, you'll be able to come up with some other options that are much simpler and easier to use.

All right, at this point, you've really crossed a threshold, and you can be excited about how far you've already come. You've now seen all the fundamental components of programming that people have used for decades, and that still form the core of most of the software all around us. So, in the next lecture, we're going to look at how to take a more systematic approach to debugging our programs.

11

# Error Types, Systematic Debugging, Exceptions

Whenever you try to write your own programs, you're going to encounter the nemesis of all computer programmers: bugs. Just like a criminal in a detective novel, bugs can cause trouble when you least expect them, hide for a long time, and be tough to track down and eliminate. Detectives eventually track down and capture criminals, not only through systematic, persistent effort, but also with the help of forensic tools. Likewise, in programming, systematic and persistent effort to find and eliminate bugs is greatly enhanced by the use of tools you will learn about in this lecture.

## [ BUGS AND ERRORS ]

- A **bug** is just a mistake made by a programmer. It's an error, fault, or defect. Sometimes there are different connotations among these terms, but they're all basically the same thing. It could be that something was typed wrong. It could be that the programmer didn't think about some interaction between two parts of a program or got interrupted and forgot to come back to finish something. Or, maybe it was just a bad design from the beginning.

- Even the very best programmers have bugs, although it's true that the number of bugs you create will decrease a little as you gain experience. But what makes some of these programmers great is that when they do have bugs, they can find and eliminate them quickly.

- The easiest bugs to find are usually **syntax errors**, which happen when you have mistyped something in the program. Most syntax errors are going to be relatively easy to find, because the interpreter or compiler

for any high-level language isn't going to let you go forward. Python's interpreter-style compiler will give you a syntax error message and stop the program from executing if it finds a problem like these.

- There are other errors you might not find until the program is actually running. These are called **runtime errors**. Many runtime errors will involve someone giving input incompatible with what's required. Maybe you are asking for a month, expecting the person to enter a number, and he or she types in "January." This would cause your conversion from a string to an integer to fail.

- A third category of errors, and the ones that are most difficult to find, are **logic errors**, in which the computer will run the statements just fine, but the output will be incorrect. Remember that computers just do what they're told—they follow the instructions exactly—and don't know that they didn't do what was wanted. These can take many different forms.

- Some logic errors are like syntax or runtime errors, except that the interpreter lets them through. You might forget the way something was spelled or how you capitalized, but the computer does not catch the mistake. Instead, the new spelling can end up creating a new variable with a different name. The interpreter doesn't know what you meant to say, only what you actually did say, so you have to be careful. You'll have written code that is perfectly valid; it's just not doing what you wanted it to. That can make it difficult when reading over the code to notice the mistake you've made.

- Even more problematic, though, is when you have a logic error in your thinking and you actually meant to have a line like the one you wrote but shouldn't have. In other words, you really thought you wanted that less-than sign, for example, but it wasn't the right way to solve the problem. These are some of the most difficult errors to track down, because eventually you have to realize that the problem is in your thinking, not in the code.

## SYSTEMATIC TESTING FOR BUGS

› Fortunately, there are ways of dealing with all of these logic errors. Let's look at some code from a program that reads in data from a file and stores it in a list for later analysis. Usually, we do not see the error right away—we have to discover it.

```
for line in infile:
    #STUFF DELETED HERE
    m, d, y = date.split('/')
    month = int(m)
    day = int(d)
    year = int(y)
    #Put data into list
    datalist.append([day, month, year, lowtemp, hightemp,
      rainfall])
#STUFF DELETED HERE
#Find historical data for date
gooddata = []
for singleday in datalist:
    if (singleday[0] == month) and (singleday[1] == day):
        gooddata.append([singleday[2], singleday[3],
          singleday[4], singleday[5]])
```

› The general approach for debugging has three stages. First, we need to thoroughly test our code. Testing can tell us that there's an error somewhere. Second, we need to isolate the error. We want to find the particular conditions that cause the bug or error to occur and then hone in on the particular place where the error is occurring. Finally, once we've found the bug, we need to fix it, which might be a complex task on its own, depending on the bug.

› Let's look at how this will work on a previous example that contained weather data. Remember that the first step in debugging is to thoroughly test. We'll run the program, enter a date—for example, April 6—and get some result.

```
Enter the name of the data file: DataFile1
For the date you care about, enter the month: 4
For the date you care about, enter the day: 6
There were 14 days
The lowest temperature on record was 67
The highest temperature on record was 99
The average low has been 71.71428571428571
The average high has been 92.14285714285714
```

- The result looks reasonable at first glance.

- Let's try a different day, such as April 30.

```
Enter the name of the data file: DataFile1
For the date you care about, enter the month: 4
For the date you care about, enter the day: 30
Traceback (most recent call last):
    File "C:/Users/John/PycharmProjects/TCDataAnalysis/
      DataAnalysis.py", line 51, in <module>
    avglow = sumofmin / numgooddates
ZeroDivisionError: division by zero
Process finished with exit code 1
```

- We have a major error here—a runtime error—one that's causing the program to crash. There's clearly a bug of some sort, and it looks like it's a divide-by-zero problem. We'll need to move on to the second stage: isolating the bug.

- We were lucky enough to stumble across a problem. That's what beginners often do, if they even test at all—just do a few random things and see what happens. This kind of ad hoc testing is much better than no testing, but it's not efficient. We can write our tests more systematically.

- We can build up what is sometimes called a **test suite**, which is a set of tests that we will run against our code to make sure that it's working right. The code should be tested on the test suite any time there's an addition or change to the code. In an ideal setting, you'd even write the test suite

before writing code. Realistically, though, programmers develop their tests along with their code in most cases. As your code grows, you can grow your test suite, too.

- The most important tests to run are the extreme cases—what are called **edge cases** or "corner cases." In our program, we need to think about the "extreme" dates. The first and last day of the year are extreme. It wouldn't hurt to check the first and last days of some other months.

- Then, we should check a few cases in the middle, just to be safe. We want to make sure that we handle more than just the edge cases. But it's rarely helpful to test lots of these.

- Finally, we need to check any special cases. For dates, there's an obvious special case: leap day (February 29).

- So, our test set should have, at a minimum, January 1, December 31, some day in the middle, and February 29. And, to be on the safe side, we should check a few other dates, too.

- If we run on this set of test data, we run across the same bug we found earlier. January 1 comes out okay, but testing December 31 gives us the crash we found before.

- Our next step is to isolate the bug. We have to look for clues as to what is causing the problem, eliminate things that we check are working correctly, and gradually focus on the one problem. One of the oldest methods for debugging—one that's still used in many circumstances—is just to insert a lot of print statements in the code.

```
#Perform analysis
minsofar = 120
maxsofar = -100
numgooddates = 0
sumofmin=0
sumofmax=0
raindays = 0
```

```
for singleday in gooddata:
    numgooddates += 1
    sumofmin += singleday[1]
    sumofmax += singleday[2]
    if singleday[1] < minsofar:
        minsofar = singleday[1]
    if singleday[2] > maxsofar:
        maxsofar = singleday[2]
    if singleday[3] > 0:
        raindays += 1
print(sumofmin)
print(numgooddates)
avglow = sumofmin / numgooddates
avghigh = sumofmax / numgooddates
rainpercent = raindays / numgooddates * 100
```

- In this case, we had a runtime error at the line where we computed avglow. So, we'd insert a couple of print statements right before that to print out the values that went into the calculation. If we ran the program, we'd find that both values were coming out to zero. That would give us some information, and we could use that as a clue to what was going wrong and insert more print statements to narrow in on the problem.

- As in this example, inserting print statements is a useful approach to see what is happening along the way. It's especially useful if we want to print out one thing from a long loop. Plus, it's something that you can try in almost any programming language to start honing in on a bug.

- After you've run the print statements, you can leave them in the program and comment them out, by putting a hashtag in front of them or enclosing them in triple quotation marks. Commenting them out will keep them around in case you want to run them again later.

- However, a more systematic way to isolate bugs is to make use of a **debugger**. A debugger is not a magic wand that we wave to remove bugs automatically. Still, a debugger is a tool that will help us examine our code in detail so that we can isolate a bug.

- There are several debuggers out there, and each of them works a little bit differently. But they all provide the same basic functionality. There is a debugger integrated into PyCharm. One of the great things about using an integrated development environment, such as PyCharm, is that the debugger is right there waiting for you to use it.

- Using step-by-step analysis, we can discover that the code is comparing the first element to month and the second element to the day, but the way the data is stored, the first element is the day and the second is the month. The reason we were getting no matches for December 31; we were trying to find a match for the 12th day of the 31st month.

- Once we've isolated the bug—we know exactly what the problem is—we turn to fixing the bug. The thing you should not do is just change this line of code to get rid of the bug and go on. Instead, you need to think about where this bug originated. Was it in this line, where we compare day and month to the wrong elements, or was it when we first built the list and decided to put day first and month second?

- Maybe there are other places in our code where we assumed that month came before day. Before we make any change to the code, we need to think about how the change we made is possibly going to affect other parts of the program.

- In this case, this issue is relatively isolated. We don't use the "datalist" for anything else, and there's not another check like this one. So, we could just modify this one line within the for loop. We just swap around day and month in the if statement, and that should fix the error.

```
for line in infile:
    #STUFF DELETED HERE
    m, d, y = date.split('/')
    month = int(m)
    day = int(d)
    year = int(y)
    #Put data into list
```

```
    datalist.append([day, month, year, lowtemp, hightemp,
      rainfall])
#STUFF DELETED HERE
#Find historical data for date
gooddata = []
for singleday in datalist:
    if (singleday[0] == day) and (singleday[1] == month):
        gooddata.append([singleday[2], singleday[3],
          singleday[4], singleday[5]])
```

› Or, we could change the way the "datalist" is constructed to begin with. We would just build up "datalist" with month first and then day.

```
for line in infile:
    #STUFF DELETED HERE
    m, d, y = date.split('/')
    month = int(m)
    day = int(d)
    year = int(y)
    #Put data into list
    datalist.append([month, day, year, lowtemp, hightemp,
      rainfall])
#STUFF DELETED HERE
#Find historical data for date
gooddata = []
for singleday in datalist:
    if (singleday[0] == month) and (singleday[1] == day):
        gooddata.append([singleday[2], singleday[3],
          singleday[4], singleday[5]])
```

› Either way, the next thing we should do is determine if we actually fixed the bug by rerunning all of our tests. We want to make sure that our "fix" didn't break something that was already working and that it fixed the problem it was supposed to. So, in this case, we'll run the code for our test suite: January 1, December 31, February 29, and some other day in the middle. And now it seems to work, telling us that we did seem to fix the error.

# [ EXCEPTIONS ]

› Runtime errors come up when a program is running, typically due to an unexpected input of some kind. Python, like many other languages, has a special way that it can deal with runtime errors. This is through exceptions, which are a way of handling the special error conditions that can occur when a program is running.

› For example, trying to open a file that doesn't exist, converting a string to an integer when the string turns out to be a word instead of a number, or dividing by zero will usually cause a program to crash, printing out an error. Exceptions are a way of taking these problems and handling them in some way so that the program doesn't have to crash. For example, if opening a file to read it in fails, we can ask the user for a new filename.

› Exceptions are handled in Python through "try-except blocks," which are ways of containing code that might create the exception. You start with the statement "try," followed by a colon. Then, indented is all the code for which you want to possibly check for an exception. Following that is an except statement, identifying which error you want to deal with, and finally, indented again, is the code to run in case you do run into that error.

```
try:
    #Commands to try out
except <name of exception>:
    #how to handle that exception
```

› There is a list of built-in exceptions for Python, arranged in a hierarchy, at https://docs.python.org/2/library/exceptions.html. There are a large number of different exceptions defined and a mechanism for letting users define new ones. To see a list of the standard exceptions, look up a Python language reference. A few useful ones are TypeError, OSError, and ZeroDivisionError.

› Avoid using exceptions for cases that can and should be handled at the point the problem is detected. Conversely, use exceptions only when the problem can't be handled at the point of detection.

## Readings

Gries, *Practical Programming*, chap. 15.

Sweigart, *Automate the Boring Stuff with Python*, chap. 10.

## Exercises

1 Imagine that you have written a piece of code that is supposed to return a ticket price given an age. Those under age 3 are free, other children from 3 to 12 are $5, and all others are considered adults and cost $10. When you test your code, what are the ages that would be good to use in your tests?

2 Assume that you have written the following code to find the middle element from a 3-element list.

```
def findmiddle(a):
    if ((a[0] >= a[1]) and (a[1] >= a[2])) or ((a[0] <= a[1])
      and (a[1] <= a[2])):
        return a[1]
    elif ((a[0] >= a[2]) and (a[2] >= a[1])) or ((a[0] <=
      a[2]) and (a[2] <= a[1])):
        return a[2]
    else:
        return a[0]
```

Notice that if a list is passed in that is not of length at least 3, the code will give an error.

a) Modify the function so that it will raise an exception if the list is not valid.

b) Then, show how you could call the function, printing a message if there was an exception.

TRANSCRIPT 11

# Error Types, Systematic Debugging, Exceptions

Whenever you try to write your own programs, you're going to encounter the nemesis of all computer programmers—bugs. Just like some criminal in a detective novel, bugs can cause trouble when you least expect them, they can hide out for a long time, and they can be tough to track down and eliminate. Detectives eventually track down and capture criminals, not only through systematic, persistent effort; but also, increasingly, with the help of some forensic tools. Likewise, in programming, systematic and persistent effort to find and eliminate bugs is greatly enhanced by the use of tools, which we'll meet in this lecture—things like test suites and debuggers.

The term bug has a long history, going back to Thomas Edison. When Edison was working on an improvement to the telegraph, he ran across a design problem that he fixed by building what he called a bug trap. And he continued to use the term bug to talk about problems that he was encountering in his designs, and the term took on wider use. One of the most famous examples in computer science came from Grace Hopper, working on one of the earliest computers—the Mark II—back in 1947. The Mark II was an electromechanical computer—there were actual physical relays that would move to make connections. And in one case, when the computer wasn't functioning correctly, the source of the problem was a real bug—a dead moth—caught within the relay. The Mark II team started using the term bug to refer to problems that they encountered. And the term bug has been a common term in computer science ever since.

Now, let's be clear about exactly what a bug is. A bug is just a mistake made by a programmer. It's an error, fault, or defect. Sometimes there are different connotations among those different terms, but they're all basically the same thing. It could be that something was typed in wrong.

It could be that the programmer didn't think about some interaction between two parts of a program, or got interrupted and forgot to come back to finish something. Or maybe it was just a bad design from the beginning. Even the best programmers have bugs, although it's true that the number of bugs you create will go down a little bit as you gain experience. But what makes some of these programmers great is that when they do have bugs, they can find and eliminate them quickly. Obviously, it's better to just not make mistakes in the first place, but it's even more important to be able to find and correct your mistakes after you inevitably do make them.

The easiest bugs to find are usually syntax errors. Syntax errors happen when you've mistyped something in the program. See if you can spot the error here.

While is missing the letter e—that's a syntax error. Here, the programmer forgot how a function's defined, and didn't pass the parameters correctly. Now, most syntax errors are going to be relatively easy to find because the interpreter or the compiler for any high-level language isn't going to let you go forward. Python's interpreter-style compiler will give you a syntax error message and stop the program from executing if it finds a problem like those. These are the types of errors that you could find just from examining the syntax of the program before it's even run.

Now, other errors that you might not find until the program is actually running are called runtime errors. Many runtime errors will involve someone giving input that's incompatible with what's required. Maybe you have code where you ask the user for a file to read in. And what happens if the user gives you a file that doesn't exist? Or maybe you're asking for a month, expecting the person to enter a number, and they actually type in January. Well, this would cause your conversion from a string to an integer to fail.

A third category of error, and the ones most difficult to find, are logic errors. Here, the computer will run the statements just fine, but the output will be incorrect. Remember that computers are dumb, and they just do what they're told; they follow the instructions exactly, and they

don't know that they didn't do what you wanted them to. There can be lots of different forms of logic errors.

Some logic errors are like syntax or runtime errors, except that the interpreter lets them through. You might forget the way something was spelled or how you capitalized, but the computer does not catch the mistake. Instead, the new spelling can end up creating a new variable with a different name. The interpreter doesn't know what you meant to say—only what you actually did say—so you have to be careful. You'll have written code that's perfectly valid; it's just not doing what you wanted it to. And that can make it difficult when reading over the code to even notice the mistake that you've made.

You might not remember the order of parameters in a function and maybe reverse the month and day—you meant to say February 8, and instead you said August 2. You might accidentally switch a greater than to a less than, or you write equals when you meant to write plus equals, or you put in a minus where you meant to have a plus, or you forgot to indent. Maybe you meant to hard code that your favorite color was yellow, but you accidentally wrote that it was blue. All these errors are logically valid statements, and the program runs perfectly fine with them. So, an interpreter won't catch them for you, and they won't usually cause a program to crash. They just mean that you get a wrong answer.

Even more problematic is when you have a logic error in your thinking, and you actually meant to have a line like that, but you shouldn't have. In other words, you really thought that you wanted that less-than sign, or that equals assignment or that subtraction, but it wasn't the right way to solve the problem. These are some of the most difficult errors to track down because eventually you have to realize that the problem is in your thinking, not in the code. Fortunately, there are ways of dealing with all these logic errors.

Let's look at some code from a program that we've seen before, which reads in data from a file and stores it in a list for later analysis. But, I've made one change to it that introduces an error. I wonder if you can see what it is? Take a look, but if you don't find it, that's totally fine.

Now, what usually happens is we do not see the error right away. You have to discover it.

Here's the general approach I recommend for debugging. They are basically three stages to follow. First, we need to thoroughly test our code. We've seen testing before, but now we're going to see how to do it in a much more thorough and precise manner. Testing can tell us that there is an error, somewhere. Second, we need to isolate the error. We want to find the particular conditions that caused the bug or error to occur, and then hone in on that particular place where the error is occurring. And finally, once we've found the bug, we need to fix it—which might actually be a complex task on its own, depending on the bug.

Let's look at how this'll work on the example we saw earlier. Remember that our first step in debugging is to thoroughly test. We'll run the program, enter a date—like April 6—and get some result. We see the result, and it looks reasonable at first glance. Well, maybe the high temperature averaging 92 seems a little high for April, but at least, we've got something that's not totally unreasonable.

Let's try a different day, like April 30. Woah! We've got a major error here—a runtime error, one that's causing our program to crash. There's clearly a bug of some sort, and it looks like it's a divide by zero problem. We'll need to move on to the second stage, actually isolating the bug.

Now, we were lucky enough to stumble across a problem. That's what beginners often do if they even test at all—just sort of do a few random things and see what happens. This kind of ad hoc testing is much, much better than no testing at all, but it's not efficient. We can write our tests more systematically. We can build up what's sometimes called a test suite—a set of tests that we'll run against our code to make sure it's working right. The code should be tested on the test suite any time there's an addition or a change to the code. In an ideal setting, you'd even write the test suite before writing code. But realistically, programmers develop their tests along with their code in most cases. As your code grows, you can grow your test suite, also. Usually, the most important tests to run are the extreme cases—what are called

edge cases or corner cases. Think of a checkerboard—the squares in the corners or on the edges are different from squares in the middle. They don't have the same number of neighbors, for instance.

In our particular program, this means that we need to think about the extreme dates. The first and last day of the year are extreme. And it wouldn't hurt to check the first and last days of some other months. Then, we should check a few cases in the middle, just to be safe. We want to make sure that we handle more than just the edge cases, but it's rarely helpful to test lots of these—a few of them is plenty. And finally, we need to check any special cases. For dates, there's an obvious special case. OK, you can include Christmas and your birthday, but I'm talking about leap year—we need to check February 29. So, our test set should have, at minimum, January 1, December 31, some day in the middle of the year, and February 29. And, to be on the safe side, we should check a few other dates, also.

If we run on this set of test data, sure enough, we run across the same bug we found earlier. January 1 comes out OK, but testing December 31 gives us the crash that we found before. Our next step is to isolate the bug. We have to look for clues as to what's causing the problem, eliminate the things that we check that are working correctly, and gradually focus in onto the one problem.

One of the oldest methods for debugging, and one that's still used in many circumstances, is just to insert a lot of print statements into the code. So, in this case, we had a runtime error at the line where we computed avglow or average low. So, we'd insert a couple of print statements right before that to print out the values that went into the calculation. If we ran the program, we'd find that both values were coming out to zero. And that would give us some information, and we could use that as a clue as to what was going wrong and insert more print statements to narrow in on the problem further.

As in this example, inserting print statements is a useful approach to see what's happening along the way. It's especially useful if we want to print out one thing from a long loop. Plus, it's something you can try

in almost any programming language. To start honing in on a bug after you've run the print statements, you can leave them in the program, but comment them out. You can do this by putting a hashtag in front of them or enclosing them in triple quotation marks. Commenting them out will keep them around in case you want to run them again later.

However, a more systematic way to isolate bugs is to make use of a debugger. Despite its name, a debugger is not a magic wand that we wave to remove bugs automatically. It's not like running an antivirus software that does everything for you—I mean, that'd be cool; I wish we had that. But a debugger is still a tool that'll help us examine our code in detail so that we can isolate a bug.

Now, there are several debuggers out there. Each of them works a little bit differently, but they all provide the same basic functionality. I'm going to show you how to use the debugger integrated into PyCharm. One of the great things about using an integrated development environment, like PyCharm, is that the debugger is right there waiting for you to use it.

Like many programming tools, debuggers can include lots of features, and you could spend hours going over all of them. I'm going to focus here on a few of the important ones that'll be most helpful.

You know that green arrow in the upper right corner that we've been clicking to run our programs? Right next to it, we see a little green bug. This starts the debugger. When we click on that bug, the debugger will start, very much like running the program did. It'll first ask for the data file to use, and then we'll enter in the test case that we know makes it fail—December 31. When the program crashes, notice that the line where the code crashed is highlighted. The line has the division, and we're getting a division by zero, so this makes sense. Now we have the line of code where the program's crashing, but this is important—this line of code is not actually the problem. This line of code is just where the problem became apparent. Usually, when you have a bug in your program, the bug is actually in one place, and it doesn't become apparent until much later.

Down in the window where we see the output—what's called the Console window—we also see a tab labeled Debug. If we click on the debug tab, we see three new subwindows called Frames, Variables, and Watches. We click in the Variables window, where we see all of the variables that are active in the program, along with their value on the current line. For us, the current line is the place we crashed. Our division on the line where we crashed was trying to divide sumofmin by numgooddates. If we scroll through that variable list, both sumofmin and numgooddates are zero. So, we now have our first clue as to the problem. We seem to have a sum of zero and have found no dates matching the date we entered. That's weird; we should have plenty of dates that match December 31 in the set. So, we have a first clue here—it seems we didn't find any good dates.

Now, we need to determine why we didn't find any good dates. It's OK to make a hypothesis and check it. Let's suppose we don't have a good date because we didn't read in the month and day correctly. In that case, the day and month variables should be incorrect. If we scroll through the variable list, we can see that day is set to 31 and month to 12, so that's not the problem after all. Maybe instead, the problem is that our data from the file is missing. We can look at that variable, which is called datalist. Well, in datalist, we see a bunch of information, so we certainly have something there—it's not missing data. These easy checks aren't finding anything.

Since we have the clue about there not being any good dates and the number of good dates should be the number of dates in the gooddata list, that probably means that the gooddata list is empty. So, sure enough, if we examine the list of variables in the debugger, we'll see that gooddata list is empty. Let's examine that part of the code.

Now, I want to show you another useful feature of debuggers—the breakpoint. A breakpoint is a way of telling the computer "Run the code up to this point, but then stop or break." It's like we stopped the program waiting on user input; though, here, the user isn't involved. We can set a breakpoint by clicking right before the line of code in the editor. Notice the red dot that appears? That indicates that there's a breakpoint at that

line. Breakpoints are a great way to just stop a program partway through and examine what values everything has. So, in this case, we'll set a breakpoint right after creating the gooddata array—right on the for loop. We know that sometimes creating gooddata works since it worked for January 1. But, sometimes it doesn't, since it didn't work for December 31.

We should start the program again, and enter our information—the filename and 12/31. Notice that now the program comes to a stop where we set the breakpoint. That line of code is highlighted in the screen, and we can look and examine variables. Scrolling through the list, we see that our variables all seem to have the values we'd expect. So, it looks like everything up to this point is probably OK. But we know that after this loop, things aren't OK—we never get any entries in gooddata. So, we want to see what happens as we go through the loop. Why are we not getting any dates put into the gooddata list?

We can now look at another tool we have available to us in the debugger, and that's the step over command. If you look in the debugger window, you'll see a set of arrows. The step over arrow will basically take you to the next line of code. It executes whatever is on the current line, and then stops on the next line. If we click that button, we can look at the variable list and see there's now a variable called singleday, and it has data for the first day. Now, the if statement that we're on is going to compare the month and day to the first two elements of singleday. In this case, those elements are both 1—that is, it's January 1—so obviously, there's not a match. If we click step over, we'll go back up to our for statement. Besides step over, we also have a step into button. Step into is used when there's a function call. Instead of just doing the whole line of code, your next step is into the function that's called. It's a great way of checking whether the function code is behaving like you expected.

A good debugger will have additional ways to show you where you are in the program, and what's going on. In PyCharm, the Frames window shows the sequence of functions that have been called. If those functions are nested, like one frame inside another, it shows you how deep you are in the program. If a function calls a function that calls a

function that calls a function, then this window would show you that whole sequence of function calls.

The Watches window lets you set up some variables to be watched. This is more systematic than just inserting a lot of print statements. You can add variables to the watch window, and their value will be updated there as you step through the code—it just saves you the time of scrolling through the variable list over and over. In this case, let's add singleday since that's one we're looking at. If you're tired of going line-by-line, you can always hit the play button—the green arrow in the debugger window—and it'll keep running until it hits another breakpoint.

OK, but for us, we're still trying to figure out this part of code, so we'll check how this loop is working some more. We click on step over, and sure enough, singleday is updated. It now shows as being a list—2, 1, 2000, et cetera. Let's see our if statement at this point. We're comparing the month to the first element, and the day to the second element. Whoa; hold on! Something seems a little wrong here. We went to our second day, and it was the first element in the list that changed. This is the element that we're comparing to month, so it looks like we went from January 1 to February 1. Maybe the data really did go from January 1 to February 1, but that doesn't seem likely, so we've got another clue. We don't have a definitive answer yet, but we're getting close—I can feel that we're closing in on that bug.

Let's take a few more steps through to see how things progress. If I keep clicking on step over, we'll see that singleday keeps updating on each iteration of the loop, and that it's the first element that keeps going up by one. We can keep doing this until we see that the singleday has a first element of 13. Now, we know there's no 13$^{th}$ month, so this must be a problem. And, now we finally have some proof that we've got our bug trapped. Our code is comparing the first element to month, and the second element to day. But the way the data is stored, the first element is the day, and the second is the month. No wonder we were getting no matches for December 31; we were trying to find a match for the 12$^{th}$ day of the 31$^{st}$ month. So at this point, we've isolated our bug. The debugger was a key part of this.

With our bug now isolated, we know exactly what the problem is. We now turn to fixing the bug. The thing you should not do is just change this one line of code to get rid of the bug and go on. Instead, you need to think about where this bug really originated. Was it in this line, where we compare day and month to the wrong elements? Or was it when we first built the list and decided to put day first and month second? Or maybe there are other places in our code where we assumed that month came before day. Before we make any change to the code, we need to think about how the change we make is possibly going to affect the other parts of the program.

In this case, the issue is relatively isolated. We don't really use the datalist for anything else, and there's not another check like this one. So, we could just modify this one line within the for loop. We just swap around day and month in the if statement, and that should fix the error. Or, we could change the way the datalist is constructed to begin with. We could just build up datalist with month first, then day.

Either way, what's the next thing we should do—celebrate that the bug is fixed? Of course not. We first need to determine if we actually fixed the bug. The way we do that is by re-running all of our tests. We want to make sure that our fix didn't break something that was already working, and that it fixed the problem that it was supposed to. So, in this case, we'll run the code for our little test suite—January 1, and December 31, and February 29, and some other day in the middle. And now it seems to work, telling us that we really did seem to fix the error.

So, our process here was to come up with a complete set of test cases, run these tests, and then for any error we found, isolate it and fix it, and then test that fix. Not all errors will show up as crashes. In fact, most errors will simply be erroneous values produced. The process for dealing with them is the same.

Now that we've got a way of dealing with logic errors and syntax errors let's turn to runtime errors. These are the things that come up when a program is running, typically due to unexpected input of some sort. Notice that although the example we just ran through was something

we saw at runtime, we're not calling it a runtime error. The end effect is not that different, but what we had was not just a failure to deal with an unexpected case. We actually had a wrong concept in our code, where we assumed data was entered one way when it was actually entered another. We thought the data was being stored as month then day, instead of day then month, and that's why we call it a logic error. However, the main difference between logic errors and runtime errors—and maybe the only reason to care about the difference—is the way we treat these errors.

Python, like many other languages, has a special way that it can deal with runtime errors. This is through exceptions. Exceptions are a way of handling the special error conditions that can occur when a program is running. For example, trying to open a file that doesn't exist, or convert a string to an integer when the string turns out to be a word instead of a number or trying to divide by zero will usually cause a program to crash, printing out an error. Exceptions are a way of taking these problems and handling them in some way so that the program doesn't just have to crash. For instance, if opening a file to read it in fails, we can ask the user for a new filename.

Exceptions are handled in Python through what are called try-except blocks. A try-except block is basically a way of containing code that might create the exception. You start with the statement try, followed by a colon, then indented is all the code for which you possibly want to check for an exception. Following that is an except statement, identifying which error you want to deal with. And finally, indented again, is the code to run in case you do run into that error.

Let's see a particular example. In this case, we try to open a file that a user entered. If there's an error in opening that file, we have an except statement that catches the exception. In this case, it prints a message and opens some different file. Now, we had to know that OSError was the name of the exception we were looking for, and that means we needed to know the possible errors.

Here's a list of built-in exceptions for Python, arranged in a hierarchy. There are a large number of different exceptions defined, and a mechanism for letting users define new ones. To see a list of the standard exceptions, like here, look up a Python language reference. But to give you an idea, I've listed a few useful ones here: TypeError, OSError, and ZeroDivisionError.

A segment of code could potentially raise more than one type of exception. We can deal with this by using multiple except blocks. For example, if we're writing a function that takes in a couple of numbers and divides one by the other, we could run into issues in a couple of places. We could have a problem if someone passes in a string or something weird; this would create a TypeError. Or we could end up with a ZeroDivisionError, or maybe there's some other error. We can list each category of error in its own block. And for the other errors, we can have a single except block that's used to catch all of them.

If needed, we can have two other types of blocks. The first of these is an else block. The else block comes after the except statements and specifies code to run if there were not any exceptions. Only if there were no exceptions raised in the code in a try block does this get executed. There can also be a finally block, which comes at the very end, after any except or else blocks. The finally block will get executed after everything else, whether there were exceptions or not.

Most of the time, exceptions get raised when a regular system error occurs. But as a programmer, we can raise an exception ourselves. What this means is that we basically raise a flag saying, "Hey, such and such out-of-the-ordinary thing happened here." The command you use to do that is called raise. For example, if you have a date function and want to ensure that the month lies between 1 and 12, you could check for this and raise a TypeError exception if the month is of an incorrect type. Or, say you had a function that was designed to look for a pattern of length 5 within a sequence. If you received a pattern of a different length, it would make sense to raise an exception.

Now, let me give a word of advice. Be careful when using exceptions; they can be powerful ways to handle some of the runtime errors that would otherwise cause your program to crash. But they can easily be abused to do things that are better done with other code. Exceptions should be for exceptional situations. For example, if you're checking whether the month is between 1 and 12, it's often just as easy to check for those cases with an if-then statement and ask the user to re-enter data. And this use of standard code is much more normal looking than an exception.

So, avoid using exceptions for cases that can and should be handled at the point the problem's detected. Conversely, use exceptions only when the problem really can't be handled at the point of detection. For example, exceptions are about the only way you can deal with the pattern-matching example where the wrong length pattern is given. And if an exception does happen, you want to handle it—that's what the try-except blocks are for.

Let me tell you about a really expensive case where an exception was not caught, and the impact of that. One of the most expensive software bugs ever was due to an exception in the inertial guidance system of the European Space Agency's Ariane 5 rocket. To quote the official report from 1996, "The exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value."

What happened, basically, was their software tried to convert a float to an int, but it wasn't able to because the float was too large. The number stored in the 64-bit floating point was too large to fit in the 16-bit integer. So, the software, which was written in the language Ada, generated an Operand Error exception—which just means there was an error in something being operated on. However, they didn't have anything to catch the exception and handle it. As a result, the guidance system just failed completely. The rocket and its payload of four satellites were all lost, at a cost of $500 million.

When we debug, we're looking for three types of errors. Syntax errors are usually found by the compiler, and logic errors can be eliminated

with the help of our debugger tool and steady, systematic testing and examination. Runtime errors generate exceptions, and these are things we can catch and deal with in our program using the try-except blocks. The more time you spend time working through a debugger or writing all your test cases, the better you'll become at debugging—and thus at coding. Use of test suites and systematic debugging also make it much easier for other programmers to be interested in trying out any code you might write or inviting you to contribute towards other projects.

And speaking of program sharing, in the next lecture, we're going to learn about importing modules and packages that can dramatically improve what we're able to do in our own programs. I'll look forward to seeing you then.

12

# Python Standard Library, Modules, Packages

"Batteries included" is a slogan associated with the Python programming language because of the many powerful functions that are included with every installation. In this lecture, you will learn how to access functions that have been pre-bundled with Python's standard library in a form known as modules. Modules, and bundles of modules known as packages, give even beginning programmers access to enormous power when writing their programs. You will also learn about the many thousands of third-party modules that are available for download.

## [ THE PYTHON STANDARD LIBRARY ]

- Very early on, computers were basically unique devices, and if you wanted to write code for the computer, you had to write it for that one computer. But, as computers became more uniform, you could write programs that would run across all of them. And as programming languages became more standardized, one person could write code that other programmers could use.

- Modules, also known as libraries, provide a nice way of packaging up functions from one program and making them more widely available for use in other programs.

- A Python module is actually just an ordinary Python program, but it's organized in a way that supports abstraction, meaning that we don't have to worry about the inner workings of the module to use it. To preview this in very simple terms, basically all we have to do is use an "import" command to load one program for use in another program.

- Importing a program as a module can save us enormous amounts of time and effort. In fact, without modules, it's difficult to imagine that the entire software industry could have developed nearly as quickly or fully as it has.

- There are a many different types of modules with all kinds of different uses. Some of them aid us in basic programming functionality. For example, a math library, such as the math module from the standard library, or the downloadable NumPy module, gives us access to many other mathematical functions. We get access to all of these functions by the command "import math," or "import numpy," and then can reference things like pi and the sine function.

```
import math
print(math.sin(math.pi/2))
```

- Some modules let us communicate over networks. There are libraries, such as ssl, that will let us set up network connections. Others, such as webbrowser, can open up a browser window. The following code will cause a browser window to open to The Great Courses website.

```
import webbrowser
webbrowser.open("http://www.thegreatcourses.com")
```

- Some modules will provide a graphical display. For example, we might draw some graphics to the screen with the turtle library. This code draws a line 100 pixels long.

```
import turtle
turtle.forward(100)
```

- Other modules might help you get input from a computer. Tkinter provides a link to the commonly used Tcl/Tk library that's used for making graphical user interfaces.

- Other modules, such as shutil, will let us do things on the computer itself, such as copy a file. The following code copies file1.txt to file2.txt.

```
import shutil
shutil.copy("File1.txt", "File2.txt")
```

- And that's just the beginning. There are hundreds of modules distributed with every installation of Python as part of the Python standard library. And there are thousands more available for download from PyPI or independent websites. If you want to see how many are already installed, try typing "help("modules")."

- Many of these modules do things you probably wouldn't know how to do on your own, and that's the point: Thanks to these modules, all the details of complex functions have been abstracted away, making them much easier for you to use.

- Every Python program is saved in a file that ends in a ".py" extension. Even if you write code in an integrated development environment, such as PyCharm, the code you write is automatically saved in a ".py" file, and that file is what is run.

- The typical use for a module is not to hold code to run right away when it is imported in. Instead, a module will usually define a set of functions we can use later on in our program. A module can also define variables. And it can define classes.

- So, this is how modules work: You essentially have other files that contain a bunch of function definitions. Then, you have an "import" command that, in effect, loads that file into your program.

- Basically, every Python program can be regarded as a module in waiting, because every ".py" file can potentially be imported for use as a module. So, in that sense, there are probably millions of potential "modules" floating around out there.

- But while technically any Python can be regarded as a module, realistically what raises a program to the level of a module is that the program provides a set of functions, classes, and constants that all work together to accomplish a common goal. A good module will also provide a "clean" interface to the user, revealing only as much about how it works as is needed for the user to use the module effectively.

## [ MODULES IN ACTIVE USE ]

- When it comes to modules in active use, there are two main groups of modules to consider: the Python standard ibrary modules and the third-party publicly distributed modules.

- When you install Python, a set of modules is also installed on your computer by default—the Python standard library modules. You still have to import these modules to use them, just like any other module. However, you can rely on these modules being there for you to use, and if you write a Python program, you can be sure that someone else running that version of Python will be able to run your program, too.

- Information about modules in the Python standard library can be seen in many places online, including the documentation at Python.org: https://docs.python.org/3/library/. You'll see a whole list of modules there, and more about each module is just another click away.

- For example, the standard library's module called "math" has a list of about 45 functions that are included. There is a square root function, and there are functions for computing greatest common divisor or cosine. In addition, some values are defined, such as the value of pi.

- To write a program using these commands, we'd just import the math module. Then, we can use "math.cos" to compute the cosine and "math.pi" to get pi.

- If we were going to be doing a lot of math, we might bring in the whole math module by writing "from math import *." We can then write our math calculations even more directly, without having to put "math." in front of each one.

- When there's something you want to do, especially if that something is outside of what could be considered "standard" programming, you should check around to see if there's a module that can do that for you. Determining which modules are important and which ones aren't depends on what you're trying to program.

- The Python standard library is quite useful on its own, and its modules are probably the most important and widely used set—that's why they're part of the standard. But there's a much bigger source of modules out there; these are the third-party modules.

- Because creating a module is really just creating a Python file, basically everyone can write modules. People can then put these modules on the web, and others can download them and use them. And people have put many useful modules online. There are thousands of Python modules that do all kinds of interesting things.

## [ PACKAGES ]

- One thing that helps you find what you want is that modules themselves are often bundled together into what Python calls **packages**. A package is a collection of modules.

- A package can bring in many modules, and we can access those modules by adding an extra period and then the name of the subpackage or module. The rest of the import works just like before.

- There are many popular packages out there. NumPy and SciPy are packages used for a lot of mathematical and scientific computing. Matplotlib provides graphing and plotting capabilities. ZeroMQ is a

package that provides messaging capabilities, and Twisted is one that provides networking. Beautiful Soup helps process HTML files, and Requests provides a way of getting data files over the web. And all of these are just scratching the surface.

- To find a good Python module for something we want to do, there are a few options. One would be to browse through the Python Package Index (PyPI): https://pypi.python.org/pypi. The PyPI is an "official" index of Python modules that other people have released. These are not automatically included with your Python installation, and the Python modules that are collected there are not necessarily good, or reviewed, or rated.

- Almost anyone can create a module and upload it there for others to use. There are several tens of thousands of modules uploaded to the PyPI, and the purpose of the index is to make sure that there's a central place people can go for the modules they want.

- A second option is to just do an internet search for the topic you want a module for and then add "Python module" to the end.

- Once we've figured out which external package or module we want, we have to download it. Details about how to download may vary, depending on the package or module. Some packages will have their own website, where you can just click on a link to download and install a whole package very easily.

- Python also has a recommended tool for installing packages, called pip. Assuming that you installed a recent version of Python, pip will have been installed automatically for you, so it will already be on your computer.

- To use pip, you will need to go to the command line in your computer. The command line is an interface in your operating system that you might not be that familiar with from standard usage. In Windows, you can get to the command line by running the program "CMD." On a Mac, you want to run the "Terminal" program. If you don't know where it is, you should be able to find it in the Applications and then Utilities folder.

- The command line will let you type in commands to the operating system directly. If you installed Python so that it could be run from anywhere, you can type the next command anywhere. Otherwise, you'll need to go to the directory that contains Python.

- You can install a Python package using pip by typing the line "python –m pip install <package name>." That should find and install the package you specify, along with any packages it needs. Once that is done, you'll be able to access that package from Python, just like any of the standard library packages.

- Once you've installed a package on your computer, using it is just the same as the standard Python library. You just import whichever packages or modules you want to use and go from there.

- For most packages, you can get a list of commands that a package provides from within Python. After importing the module, you can use the "dir" command, passing in the module name.

```
import math
print(dir(math))
```

- This list will show you the names of the functions provided. It's not necessarily a whole lot of help to see the function names without knowing what parameters they take or what they do, but it's a start, and it can be useful to verify that you didn't accidentally overwrite a function name or something.

- If the developers of the module were good about using docstrings, you should be able to write "help()," with the function name in parentheses, to find out more about the function, too. More helpful, however, is to look at the online documentation for that package to see how to use it.

## Readings

Gries, *Practical Programming*, chap. 6.

Sweigart, *Automate the Boring Stuff with Python*, chaps. 7–18.

## Exercises

**1** From the Python standard library, find the module you could import to do each of the following.

a) Read zipped files and compress and uncompress files in a zip format.
b) Work with numbers as fractions.
c) Send email. Note: Email is sent using the SMTP protocol.
d) Work with URLs (the addresses of web pages).

**2** What would be the code you would write to make a new directory named "DataDir" off of the current one? Note: You can do this using the "os" module, which is part of the Python standard library. You will probably need to look at the "os" module documentation to find the appropriate command.

TRANSCRIPT 12

# Python Standard Library, Modules, Packages

"Batteries included" is a slogan associated with the Python programming language. It refers to the many powerful functions that are included with every installation. In this lecture, we'll see how to access functions that have been pre-bundled with Python's standard library in a form known as modules. Modules and bundles of modules known as packages give even beginning programmers access to enormous power when writing their programs. We'll also look at the use of many thousands of third-party modules available for download.

Things haven't always been this easy. Very early on, computers were basically unique devices. And if you wanted to write code for the computer, you had to write it for that one computer. But as computers became more uniform, you could write programs that would run across all of them. And as programming languages became more standardized, one person could write code that other programmers could use. Modules, known as libraries, provide a nice way of packaging up functions from one program and making them more widely available for use in other programs.

Now, a Python module is not something mysterious or new. It's actually just an ordinary Python program, like any of those we've been writing, but it's organized in a way that supports abstraction—meaning that we don't have to worry about the inner workings of the module in order to use it. To preview this in very simple terms, basically, all we have to do is use an import command to load one program for use in another program. Importing a program as a module can save us enormous amounts of time and effort. In fact, without modules, it's hard to imagine that the entire software industry could have developed nearly as fast or fully as it has.

There are a lot of different types of modules out there, with all sorts of different uses. Some of them aid us in what I'd call basic programming functionality. For example, a math library—like the math module from the standard library or the downloadable NumPy module—gives us access to a lot of other mathematical functions. We get access to all these functions by the command import math, or import numpy, and then we can reference things like pi or the sine function.

Some let us communicate over networks. There are libraries, such as SSL, that will let us set up network connections. Others, such as WebBrowser, can open up a browser window. This code will open a browser window to pop open to The Great Courses website. Some modules will provide a graphical display. For example, we might draw some graphics to the screen with the turtle library. This code draws a line 100 pixels long. Other modules might help you get input from a computer. Tkinter provides a link to the commonly used Tcl/Tk library that's used for making graphical user interfaces. Other modules, such as shutil will let us do things on the computer itself, like copy a file. This code copies file1.txt to file2.txt.

And that's just the beginning. There are hundreds of modules distributed with every installation of Python as part of the Python standard library, and there are thousands more available for download from PyPI or independent websites. If you want to just see how many are already installed, try typing "help;" and then in parentheses, "modules."

Many of these modules do things you probably wouldn't know how to do on your own, and that's the point. Thanks to these modules, all the details of the complex functions have been abstracted away, making them much easier for you to use. Every Python program is saved in a file that ends in a .py extension. Even if you write code in an integrated development environment, or IDE—such as PyCharm—the code you write is automatically saved in a .py file, and that file is what's run.

I've got a program here called ModuleDemo that I've written in PyCharm. It's a short little program that just reads in a string and then prints it back out, and it does so forever. If I run it, I can type in test, and

it prints test. OK, now let's look at the directory that it's in, and let's run that Python file on our computer directly, without going to the PyCharm IDE. If I navigate to the directory where my PyCharm projects are stored, I see the file ModuleDemo.py. Since I have Python installed on my computer, I can just execute this file directly. Notice that it pops up a window, and it works just like what we saw before in the PyCharm IDE—I type something in, and it spits it back out. This file, ModuleDemo.py, is a standalone set of Python code; I didn't need to be in PyCharm to run it. The file itself is just a file, consisting of text, with all the lines of Python code that we've written.

The idea of modules is that we can take Python code like that—like this segment of code that we wrote in ModuleDemo—and import it for use as part of a different Python file. So, first, let's make a copy of the old Python module we had, and we'll call the new one ModuleA.py. ModuleA.py is just those three lines of code that we saw.

OK, now back in the PyCharm IDE, let's show how we can bring that into another Python program as part of our main file. We'll delete the code in ModuleDemo. Remember that we have ModuleA still sitting out there. We'll instead use a single line of code, saying import ModuleA. That's it. That's our entire program, saying import ModuleA. Now, what the import command does is it takes the file listed, and it essentially just inserts it into your file. So, what do you think will happen when we run this program? Well, it runs just like the code we saw before. It's still that same little program that asks us to type something in and then spits it back out.

The typical use for a module is not to hold code to run right away when it is imported in. Instead, a module will usually define a set of functions that we can use later on in our program. A module can also define variables, and it can define classes as we'll see when we get to object-oriented programs later in the course. So, this is how modules work: You essentially have other files that contain a bunch of function definitions, then you have an import command that, in effect, loads that file into your program.

Let's say that we need a math function that will compute the sum of the squares of all numbers from 1 to *n*. Let's first modify ModuleA.py, just like we saw. I'm going to work in PyCharm since it's easy to keep multiple files open, but remember that these files work just fine even if they're run directly from outside PyCharm. We'll get rid of that earlier code we had in ModuleA, and instead write this new function. We take in a number, *n*, and then we compute a sum by going through all the numbers from 1 to *n*, and adding on the square of that number. Remember that the range command does not go until the last number, so we need to set the range from 1 to *n* plus 1. The name of our new function is SumOfSquares, and this is saved in ModuleA.py.

OK, now let's turn back to our main program. We've got that import command that lets us load in a particular library or a module—in this case, ModuleA. So, let's say that we want to use the function that we just wrote. We can then try calling the function we just wrote SumOfSquares. So what happens? It's an error. Did I misspell it or something? No. What's going on? I defined the function; I imported the file, but now I can't call the function. Well, the problem here is that the import command wants a little more information, and that's actually for our own good. First, let me show how you would actually access that new function that we just wrote. Instead of calling the function SumOfSquares, we instead call ModuleA.SumOfSquares. Notice that we first have to list the name of the module that contains the function we're calling. When we run this, sure enough, it runs just fine. SumOfSquares of 3 gives us 1 squared plus 2 squared plus 3 squared, or 14. OK, but why do we need to put the name of the module first when we call a function from that module? How does this help us?

Well, imagine that there are two people who create modules. I create my super-awesome-module, and I name it John. And Joe creates his not-nearly-as-good module and names it Joe. But, what if both of us include some function within the same name, like ReadData, and you want to use both of our modules? Well, we can still use both of these functions. We import both the module John and the module Joe. Then, we can call John.ReadData to use the function I wrote, and Joe.ReadData to use the function that Joe wrote.

Requiring the name of the module makes it possible for our program to distinguish reliably between functions that might have the same name—imported from two different modules. The people writing modules don't know what other modules might be imported into the same program, so there's a possibility of repeating a name. Given how much of programming depends on modules, requiring the module name when we call a function is actually a good thing.

There are times that we might get tired of typing an int module name, and it's not really necessary. For example, we might have a clear understanding of what we're bringing in from various modules, and we know there's not going to be some naming conflict. So, fortunately, for these cases, there's a way to bring in functions from modules so that we don't have to type the module name each time. Here's what it looks like, for the example we saw before. Instead of just the import command, we now have a from import command. We start with from, then give the module name, then say import and finally give the name of the function that we're importing. After that, we can use that function name with no difficulty—we don't need to list the module it came from beforehand. So, in this case, we can call SumOfSquares directly.

Now, one thing to keep in mind is that a module can contain lots of functions. Let's add a second function to ModuleA. This new one can be SumOfCubes. It'll work just like SumOfSquares but will add up the cubes of numbers.

Back in our main program, if we import ModuleA, we can access both functions inside of it. However, if we use the import from formulation, we need to import each function individually. It works to import each function individually on a separate line, or you can import multiple functions from one module in a single line of code, just by separating them with a comma. So, I can write "from ModuleA import SumOfSqaures, SumOfCubes."

If you want to import all the functions from a module, you can do that, too. Instead of listing each function individually, you just put in an asterisk. Now, there's a big danger in doing this, and I wouldn't

recommend it unless you're very familiar with all the functions that are included. Usually, this means that you're making heavy use of that particular module. The danger of importing all the functions is that you might unknowingly import a function that overrides one that you've already defined. Generally, you should think about what functions you'll need and import only those functions.

As we've seen, basically every python program can be regarded as a module in waiting, since every .py file can potentially be imported for use as a module. So, in that sense, there are probably millions of potential modules floating around out there. But while technically, any Python program can be regarded as a module, realistically, what raises a program to the level of a module is that the program provides a set of functions, classes, and constants that all work together to accomplish a common goal. A good module will also provide a clean interface to the user, revealing only as much about how it works as is needed for the user to use it effectively. So, let's turn to modules in active use, where there are two main groups of modules to consider. These are the Python standard library modules, and then the third-party, publicly-distributed modules.

Let's start with the Python standard library. When you install Python, a set of modules is also installed on your computer by default. You still have to import these modules to use them just like any other module; however, you can rely on these modules being there for you to use. And if you write a Python program, you can be sure that someone else running that version of Python will be able to run your program, also.

Information about modules in the Python standard library can be seen in a lot of places online, including the documentation at Python.org. You'll see a whole list of modules there, and more about each module is just a click away. For example, the standard library's module called math has a list of about 45 functions that are included. For example, there is a square root function, sqrt. There are functions for computing greatest common divisor or cosine. And you can also see there are some values defined, such as the value of pi. To write a program using these commands, we'd just import the math module. Then we can use

math.cos to compute the cosine, and math.pi to get pi. So, in this case, if we run the program, we'll print out the cosine of pi over 4. And that comes out to 0.707, which is also half the square root of 2.

If we were going to be doing a lot of math, we might just go ahead and bring in the whole math module in by writing "from math import *."

We can then write our math calculations even more directly, without having to put math dot in front of each one. So, take a little time to browse through the standard library modules yourself. You might be surprised at some of the nice functionality that's provided there.

For instance, let's suppose we want to open up a web browser to a particular page. How can we do that? If we look around the standard library, we see there's a module called WebBrowser, and within that is a simple command, open. So, it turns out that doing that is just a couple of lines of code—one to import the module, and one to actually run the command. And that's just one of many functions available in the modules of the Python standard library; there are many other modules with a variety of commands available. Many of these make some powerful commands available with a line or two of code.

For example, import the calendar module, and you can format full calendars as you like and calculate with dates. The code you see here prints out the calendar for January of 2020. Try it for yourself. Or import the time module, and you have access to a variety of time functions. For example, the ctime function you see here will get the current date and time. Again, try this for yourself. The statistics module gives you functions that you can use to compute basic statistics. The code here shows how you can use this to compute the median from a list of numbers. There are functions in the shutil module that let you mess around with files. The code you see here will copy the file temperature.dat to the file weather.txt.

We could spend days doing nothing but going over all the capabilities of various modules. But what I want you to understand is that when there's something that you're wanting to do, especially if that something is

outside of what I'd call standard programming, you should check around and see if there's a module that can do it for you. I find myself relying on various math libraries when I program. Another person might never use math modules but might want to do all sorts of file and operating system commands. And a third person might mainly use Internet data. So which modules are important and which ones aren't is really going to depend on what you're trying to program.

At this point, I want you to stop and try this yourself. It's probably easiest to start by trying out one of the short programs that I just showed. See if you can type in one of those—the WebBrowser one, or the calendar, or time would all be good places to start. With just a couple of lines of code, you get a result that's probably more complicated than anything you could have developed all by yourself. And then try to modify one of these programs, or look up one of the other commands that that library provides, and see if you can get it to do something else.

The Python standard library is quite useful on its own, and its modules are probably the most important and widely used set—that's why they're part of the standard. But there's a much bigger source of modules out there, and these are the third-party modules I mentioned earlier. Since creating a module is really just creating a Python file, basically, everyone can write modules. People can then put these modules out on the web, and others can come along and download them and use them. And people have put a lot of useful modules out there online. There are thousands of Python modules that can do all sorts of interesting stuff. One thing that helps you find what you want is that modules themselves are often bundled together in what Python calls packages, and a package is a collection of modules.

To see how this works, say you want to create your own custom package called MyFavoritePrograms. Well, a package is basically just a directory that can have several .py files within it. So, we'll create a directory called MyFavoritePrograms. Each module in your package would be an individual file with a .py extension. Let's copy our ModuleA.py file that we were working with earlier—the one that contains SumOfSquares and SumOfCubes—into this directory. Then, let's create a new file, ModuleB.

py. You can make that anything, but let's make it a copy of ModuleA.py. This is just for illustration purposes.

The main additional thing that you need, in order for your package directory to be recognized as a package, is for that directory to have a specific file inside of it, called __init__.py. This file lets Python know that the directory is actually a package. It actually can be an empty file—it just has to exist in that directory. So, we'll create a file named __init__.py, and we'll put it in the MyFavoritePrograms directory.

We could also have a subdirectory in the MyFavoritePrograms directory. That subdirectory could also contain an __init__.py file, in which case it'd be a subpackage off of the main package. So, we'll create a new subdirectory, called FunPrograms off of MyFavoritePrograms. In that subdirectory, we'll put another one of those init py files, as well as a ModuleC.py file. For illustration, let's have a ModuleC.py file, again, just be a copy of ModuleA.py. So, ModuleA, ModuleB, and ModuleC actually all contain the same code.

Now, if we wanted to actually use this package, we can import an individual module. We need to actually import an individual module from the package in order to use it. So, we could write: "import MyFavoritePrograms.ModuleA," and then we could make use of functions in Module A. Or we could write "from MyFavoritePrograms import ModuleB," and then we could make use of functions in ModuleB. Or we could write "import MyFavoritePrograms.FunPrograms.ModuleC," and then we could make use of functions in ModuleC. The point here is that a package can bring in a lot of modules, and we can access those modules by adding an extra period and then the name of the subpackage or module. The rest of the import works just like before.

There are many popular packages out there. NumPy and SciPy are packages used for a lot of mathematical and scientific computing. Matplotlib provides graphing and plotting capabilities. ZeroMQ is a package that provides messaging capabilities. And Twisted is one that provides networking. BeautifulSoap helps process HTML files. And Requests provides a way of getting data files over the web. And these

are just scratching the surface. So, how can we find a good Python module for something that we want to do?

Well, there's a couple of options. One would be to browse through the Python Package Index. The Python Package Index, or PyPI for short—and I'll sometimes say, "PyPI"—is an official index of Python modules that other people have released. These are not automatically included with your Python installation, and the Python modules that are collected there are not necessarily good, or reviewed, or rated. Almost anyone can create a module and upload it there for others to use, and there are several tens of thousands of modules uploaded to the Python Package Index. And the purpose of the index is just to make sure that there's one central place that people can go for the modules they want.

You can search on PyPI Ranking if you want to see which modules are the most popular. Details about how to download may vary, depending on the package or module. Some packages will have their own website, where you can just click on a link to download and install a whole package very easily.

Python also has a recommended tool for installing packages called pip. Assuming you installed a recent version of Python, pip will have been installed automatically for you, so that it's already on your computer. To use pip, you'll need to go to the command line in your computer. Command line is an interface in your operating system that you might not be that familiar with from standard usage. In Windows, you can get to the command line by running the program cmd. On a Mac, you'll want to run the Terminal program. If you don't know where it is, you should be able to find it under the Applications, then Utilities folder.

The command line will let you type in commands to the operating system directly. If you installed Python so that it could be run from anywhere, you can type the next command anywhere. Otherwise, you'll need to go to the directory that contains Python. You can install a Python package using pip by typing the line "python –m pip install," and then the package name. That should find and install the package you specify, along with any packages that it needs. Once that's done,

you'll be able to access that package from Python, just like any of the standard library packages. Even if that sounds complicated, once you do it once or twice you'll realize, "Oh, that's actually really simple." In fact, it's pretty amazing how easily you can pull in a complicated package.

Once you've installed a package on your computer, using it is the same as the standard library. You just import whichever packages or modules you're wanting to use and go from there. For most packages you can actually get a list of commands that a package provides from within Python.

After importing the module, you can use the dir command—D-I-R—passing in the module name. And this list will show you the names of all the functions provided. It's not necessarily a whole lot of help to see the function names without knowing what parameters they take, or what they do, but it's a start, and it can be useful to verify that you didn't accidentally overwrite a function name or something. If the developers of the module were good about using docstrings, you should be able to write help parentheses and then the function name inside the parentheses to find out more about that function. More helpful, however, is to look at the online documentation for that package in order to see how to use it.

Let's give a quick example of how we can use a package as an integral part of a program. We'll be seeing more examples as we go through the course. We'll use a module that lets us do some basic operations on directories and files. Let's say that we want to rename files in a directory. For example, when my family goes on vacation, we often take lots of pictures. When we get these pictures off of the camera, we have hundreds of files that have names that don't tell you much, other than having a consecutive number of when they were taken. Now, let's say that we want to take all the files with some extension in a directory and rename them with some new name, followed by a number. For example, maybe we want to take all the JPEG files and call them SummerVacation1.jpg, SummerVacation2.jpg, and so on.

We'll make use of the OS module, which lets us manipulate files in our operating system—the OS stands for operating system. Let's look at some code. We'll start by importing the OS module. We ask the user for the directory in which to rename. Now, the default directory for all the OS commands is the directory that the Python file is in. So, we start out by using the chdir command to change our working directory to the one the user specified The next command that we use from the OS module is listdir. That will return a list of all the files and directories in that directory.

Next, we go through each of the files in that list we just got. For each of them, we'll check to see if it ends in .jpg. That's done using the endsin function for our string. And that's not a command we've seen before; it's just going to return true if the string ends in the text passed in as a parameter. If that's true—that is, if we found a string ending in .jpg—we want to form a new string. We'll form a new string consisting of SummerVacation, plus a picture number, plus the .jpg extension. Finally, we'll use one last OS module command to rename the file. The rename function takes in the old file name and the new file name.

You could modify this code to ask the user to give you the name to stick at the front, instead of just writing SummerVacation in the code. Whenever we specify a specific value like SummerVacation instead of a variable, it's called hard coding the value. Programmers always face these kinds of choices. Hard coding is more convenient in the short term. It's easier to just write in a fixed value, rather than set up a variable, and get that variable's data read in or whatever.

By contrast, providing more flexibility to users—sometimes called softcoding—is, frankly, more time-consuming since it requires not only deciding what kind and degree of flexibility to provide, but also writing and debugging this more general code with enough test cases to make sure that you've actually provided what's needed. Still, in this case, we had SummerVacation hard coded into the code, and it wasn't hard to see that it limited the usefulness of our program. So, just by switching to allow the user to set the label for the files, it makes our code much more flexible.

So, now our little program is one you could easily reuse to rename pictures in various directories for yourself, family, or friends. The key is that the OS module has allowed us to handle a lot of those basic directory operations easily—another example of the power that modules can provide.

Modules and packages don't do your programming for you, but they can provide a huge boost to your overall programming productivity. Using the standard library or one of the third-party modules that seem to be well-supported will make it much less likely that you'll run into problems, and it'll be easier for your code to remain usable into the future.

For every module, there's going to be a bit of a learning curve—understanding what functions the module provides, how the different functions interact with each other, and the details of how each one works. No one person is going to master all the modules, whose combined code is many times larger than all the code needed to build Python itself. The key is to identify the particular tasks that you're wanting to do, and then find one or more modules that'll help you.

We'll be making use of modules in many of our future programming examples. In our next lecture, we'll be using a module to help us develop a game. It'll be fun, so I look forward to seeing you then.

## MORE PYTHON PRACTICE

Now let's use a module in a program for playing a number guessing game. We want the computer to pick a number from 1–100, and let the user guess the number—finding out if they're too high or too low. We'll count the number of tries and then keep on repeating.

OK, first, we'll want the computer to select a number, and this means picking some random number. There are some rather complex ways of generating numbers that seem random, and we could look these up and try to implement them, but we might as well use a module for generating random numbers. One of the standard library modules is random and

within it is randint. The randint function takes in two numbers, and it generates a number somewhere between the first and the second, including both. So, we can generate random numbers repeatedly.

So, how might you write your program to let the computer pick a number from 1–100 and then let the user keep guessing—telling the user if they're too high or too low?

Let's look at this implementation. We used that randint function to generate our random number for the user to guess. We then loop repeatedly, so that the game will keep playing until we break out of the program. On any one game, we first generate a new random number using randint. We then set the count of guesses to zero and say that we haven't guessed it yet. We'll then continue to loop until the user guesses. First, we ask for a guess, and we increment the number of guesses. Then, we compare that to the secret number that the computer generated. We let the user know if it's too high, too low, or if they guessed it correctly; and, if so, how many guesses it took. We set the variable guessed to True in that case—meaning that we won't do that inner loop any more—and the game will repeat after that. Try this out yourself and have fun with the game.

# 13 Game Design with Functions

In this lecture, you will learn how to develop a game that is similar to many popular computer games. You will learn how functions directly support a top-down design approach, and you will use stub functions to help you rough in the structure of the program along the way. The game will have the guts for a grid-based matching game, in which you have a bunch of objects arranged in a grid and you try to move things around to match up similar items, at which point the matched-up items disappear.

## [ THE BASICS OF THE GAME ]

- The game we'll develop will have a two-dimensional grid of different objects. In the game, we'll have five objects: the letters Q, R, S, T, and U. It will also have the same familiar game mechanics where objects disappear once we get a certain number of the same object in a row or column.

- On each turn, you get to choose to rearrange the pieces somehow. Different games have different types of moves allowed. We're going to assume that the only move we can make is to swap a piece with an adjacent piece.

- When a move is made, some objects are removed from the grid according to patterns that are made. In our case, we'll remove any cases with three or more of the same object adjacent in the same row or same column. Usually, this is what the player gets points for. Then, the remaining objects rearrange, typically by falling down to fill in the gaps just removed.

› We'll want to fill in gaps at the top with random new objects. The game continues like this until the user meets a goal. For this game, the goal will be to get a predefined number of points.

› This is a somewhat complex piece of software; it's certainly not the kind of thing we want to just sit down and start writing.

## [ DEVELOPING THE PROGRAM ]

› We will design this program using a top-down approach. At the broadest level, we have three basic steps: First, we set up everything, initializing the game. Second, we go into a loop. The condition for the loop makes sure that it's not time to end the game. Finally, within the loop, we go through one round of the game.

› In code, we can, and should, take each of these steps and put in a comment describing what needs to be done and in what order. In this case, we have three comments: one for the initialization, one for the loop, and one for taking a turn.

```
#Initialize
#While game not over
    #Do a round of the game
```

› Each of these general tasks is something that can, and usually should, be encapsulated into a function. To illustrate, every time we have one of these tasks, we'll turn it into a function call. This is the main idea of **procedural programming**, where functions are created to handle all the main tasks.

› The following is what the current code looks like if we introduce the functions. Notice that each of the original lines has turned into one function call. Those actual functions are defined, but don't do anything, because we haven't gotten to that level yet.

```
def Initialize():
    #Initialize game
def ContinueGame():
    #Return false if game should end, true if game is not over
def DoRound():
    #Perform one round of the game
#Initialize game
Initialize()
#While game not over
while ContinueGame():
    #Do a round of the game
    DoRound()
```

- If we are using top-down design in practice, we would define some of the lower levels first, before writing any of this code. In particular, you'll notice that all of our functions have empty parameter lists. That's because we don't understand yet what information we need at those lower levels, so we don't know what information needs to be passed in. Despite that, this code is the "main" program for us.

- The term we use to refer to the little functions that are placeholders for something that should be much bigger is a **stub**, which is a function that doesn't really do what it's intended to do but is just enough that everything around it runs.

- Because we've written some code, the next thing we should do is test it. To test in this case, we want to see if everything is getting called in order. The goal is to have something stable that has been tested.

- Let's look at one of the functions that hasn't been defined yet: initialization. In initialization, we need to set up the grid itself—that is, we need to get all the pieces placed into their starting positions on the grid. We also have to set the user's score to zero because we're just starting the game. And we probably want to initialize other variables, such as one that will help us keep track of which round of the game we are on.

- For one round of the game, we have three basic steps: We have to get the move from the user, update the game based on that move, and then display the new grid to the user.

- This brings us to our "continue the game" check. It turns out that this is going to be a pretty simple check for our game—we just want to see if the user has reached the goal score yet, or not. So, we'll have a conditional that checks whether the score exceeds some maximum, or not, and return true or false for that routine.

- This routine is so simple that each of those commands is basically a line or two of code. We can implement this routine as follows.

```
def ContinueGame(current_score, goal_score = 100):
    #Return false if game should end, true if game is not over
    if (current_score >= goal_score):
        return False
    else:
        return True
```

- We have an if statement that compares current score and goal score and a return of either true or false.

- In the main part of the code, we will make a change to the call to ContinueGame. We set up the score and the goal, and we call ContinueGame with that score and goal passed in as parameters.

- At this point, we should test everything.

- We need to decide how the grid itself will be represented. This is what we would refer to as a data structure. For this game, we can have a pretty straightforward data structure. Basically, we want our grid to have a set of rows and columns, and in each of those rows and columns, we have one object. In this case, an object is just a letter.

- Lists let us store rows and columns nicely. In fact, a grid representation is a list of lists.

› Let's say that our board is an 8-by-8 grid, like a checkerboard. We will thus need a list of 8 rows, each with 8 elements. We will actually set these elements to what is needed for the game in the initialization routine, but to begin with, we will make a list of all these elements.

```
board = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
```

› The initialization routine is going to have three different parts: initializing the grid itself, initializing the score, and initializing the turn. Later, we might find some other things that we wanted initialized, so we'll have to add them here, too.

› Initializing the grid is a complicated process, and we'll do that in a separate function. For the score and the turn, we will want to set the score to 0 and the turn to 1. These score and turn variables are trickier to set. These are immutable values, so we can't pass them in and change them in the function. What we can do, though, is to make sure, within the function, that we declare them as global variables. This will let us initialize them to their appropriate values.

```
def InitializeGrid(board):
    #Initialize Grid by reading in from file
    print("Initializing grid")
def Initialize(board):
    #Initialize game
    #Initialize grid
    InitializeGrid(board)
    #Initialize score
    global score
    score = 0
    #Initialize turn number
    global turn
    turn = 1
```

```
#State main variables
score = 0
turn = 0
goalscore = 100
board = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
```

› Notice that because this is a list, it is mutable, and thus we are passing it as a parameter to initialize it. There are different ways we could initialize, but let's assign random objects to each grid cell.

› To assign random objects, we need to use the random module, which is part of the Python standard library. We will import the choice function, which will randomly choose one element from a list. Then, to initialize our grid, we will loop through all 8 rows and all 8 columns and set the element to a random value. We will assume that the possible objects are the letters *Q* through *U*, but we could change those values to anything we want.

```
from random import choice
def InitializeGrid(board):
    #Initialize Grid by reading in from file
    for i in range(8):
        for j in range(8):
            board[i][j] = choice(['Q', 'R', 'S', 'T', 'U'])
```

› That's the initialization stage. We can now turn our design to the game round itself. There are four basic parts to a turn: presenting the state of the game, then getting the user's move, then determining the result of that move, and finally incrementing the turn number.

› The top-down approach means that we can create a separate function for each of these main steps. We just call these in order from our "DoRound" routine.

```
def DrawBoard(board):
    #Display the board to the screen
    print("Drawing Board")
def GetMove():
    #Get the move from the user
    print("Getting move")
    return "b1u"
def Update(board, move):
    #Update hte board according to move
    print("Updating board")
def DoRound(board):
    #Perform one round of the game
    #Display current board
    DrawBoard(board)
    #Get move
    move = GetMove()
    #Update board
    Update(board, move)
    #Update turn number
    global turn
    turn += 1
```

› The next unfinished portion of our routine is presenting the board. We're just printing to the screen at this point, so we just need to output each of the grid values, in an orderly format. We'll draw horizontal and vertical lines to separate the individual elements.

```
def DrawBoard(board):
    #Display the board to the screen
    linetodraw=""
    #Draw some blank lines first
    print("\n\n\n")
    print(" ---------------------------------")
    #Now draw rows from 8 down to 1
```

```
    for i in range(7,-1,-1):
        #Draw each row
        linetodraw=""
        for j in range(8):
            linetodraw += " | " + board[i][j]
        linetodraw+= " |"
        print(linetodraw)
        print(" ---------------------------------")
```

› That's our display routine. Let's now address our "get move" routine. For now, we'll just ask the user for a move and return that move. Notice that we haven't said what form a move should take, so for now, the "move" is just a string.

```
def GetMove():
#Get the move from the user
    move = input("Enter move: ")
    return move
```

› Next, we'll turn to the actual turn mechanics, which is embodied in the "update" routine. The turn mechanics are the main thing that define the game. They embody the rules about how the game progresses according to a move. In this case, there are a few parts, each of which will require us to do something.

› First, we'll need to update the board according to our move. In this case, that means swapping one object with an adjacent one. Then, we'll need to repeatedly eliminate pieces and update the board until there's nothing more to be eliminated. We'll have to go through and remove any pieces that are three in a row or three in a column. This will leave some empty spaces, and everything else will need to fall down. Finally, any blank spaces at the top will get filled in with new random objects.

› Putting this into code is straightforward, because each action gets turned into a new function. We stub out these functions, and then we will address each of those functions individually.

```
def SwapPieces(board, move):
    #Swap pieces on board according to move
    print("Swapping Pieces")
def RemovePieces(board):
    #Remove 3-in-a-row and 3-in-a-column pieces
    print("Removing Pieces")
    return False
def DropPieces(board):
    #Drop pieces to fill in blanks
    print("Dropping Pieces")
def FillBlanks(board):
    #Fill blanks with random pieces
    print ("Filling Blanks")
def Update(board, move):
    #Update the board according to move
    SwapPieces(board, move)
    pieces_eliminated = True
    while pieces_eliminated:
        pieces_eliminated = RemovePieces(board)
        DropPieces(board)
        FillBlanks(board)
```

- To determine the swapping, we need to convert a "move" into an actual position, and its adjacent position. To do this, we'll need to determine how to express a move. This decision will affect how we express a move when a person types it in.

- In order to express a position, we'll use a system similar to that used in chess. The columns will be numbered using a lowercase letter from *a* through *h*, and the rows will be numbered using a number from 1 to 8. So, we can express a particular position by a letter-number combination.

- Our move must also say what direction we are swapping. To do that, we'll put a single letter after the space to say whether it is swapping up, down, left, or right (*u*, *d*, *l*, and *r*, respectively). Also, there are some invalid moves.

```
def ConvertLetterToCol(Col):
    if Col == 'a':
        return 0
    elif Col == 'b':
        return 1
    elif Col == 'c':
        return 2
    elif Col == 'd':
        return 3
    elif Col == 'e':
        return 4
    elif Col == 'f':
        return 5
    elif Col == 'g':
        return 6
    elif Col == 'h':
        return 7
    else:
        #not a valid column!
        return -1
def SwapPieces(board, move):
    #Swap pieces on board according to move
    #Get original position
    origrow = int(move[1])-1
    origcol = ConvertLetterToCol(move[0])
    #Get adjacent position
    if move[2] == 'u':
        newrow = origrow + 1
        newcol = origcol
    elif move[2] == 'd':
        newrow = origrow - 1
        newcol = origcol
    elif move[2] == 'l':
        newrow = origrow
        newcol = origcol - 1
```

```
    elif move[2] == 'r':
        newrow = origrow
        newcol = origcol + 1
    #Swap objects in two positions
    temp = board[origrow][origcol]
    board[origrow][origcol] = board[newrow][newcol]
    board[newrow][newcol] = temp
```

› The next routine that's just a stub and needs to be filled in is "RemovePieces." We need to make sure that we remove any three in a row or three in a column, and we could have both cases.

› We'll create a new 8-by-8 board that we will use to keep track of whether a piece should be removed or not. We'll then update this board by looking at the rows and columns to find three of the same object and mark those spaces if they need to be removed. After we've found all the pieces to be removed, we'll go back and remove them. As we're removing them, we'll increase the score. Finally, we'll return "True" or "False," depending on whether the pieces were removed or not.

```
def RemovePieces(board):
    #Remove 3-in-a-row and 3-in-a-column pieces
    #Create board to store remove-or-not
    remove = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
    #Go through rows
    for i in range(8):
        for j in range(6):
            if (board[i][j] == board[i][j+1]) and (board[i][j] ==
              board[i][j+2]):
                #three in a row are the same!
                remove[i][j] = 1;
                remove[i][j+1] = 1;
                remove[i][j+2] = 1;
    #Go through columns
```

```
    for j in range(8):
        for i in range(6):
            if (board[i][j] == board[i+1][j]) and (board[i][j] ==
              board[i+2][j]):
                #three in a row are the same!
                remove[i][j] = 1;
                remove[i+1][j] = 1;
                remove[i+2][j] = 1;
    #Eliminate those marked
    global score
    removed_any = False
    for i in range(8):
        for j in range(8):
            if remove[i][j] == 1
                board[i][j] = 0
                score += 1
                removed_any = True
    return removed_any
```

› The next stub routine to fill in is for dropping pieces. To do this, we'll go to each column and make a list of remaining pieces from bottom to top. We'll then fill in the column with those pieces, putting zeros in at the top.

```
def DropPieces(board):
    #Drop pieces to fill in blanks
    for j in range(8):
        #make list of pieces in the column
        listofpieces = []
        for i in range(8):
            if board[i][j] != 0:
                listofpieces.append(board[i][j])
        #copy that list into colulmn
        for i in range(len(listofpieces)) :
            board[i][j] = listofpieces[i]
        #fill in remainder of column with 0s
        for i in range(len(listofpieces), 8):
            board[i][j] = 0
```

› We have just one more stub function to fill in: filling in any blank spaces with new pieces. In this case, we'll just run through all spaces, and if there's a zero, we'll replace it with a random new piece.

```
def FillBlanks(board):
    #Fill blanks with random pieces
    for i in range(8):
        for j in range(8):
            if (board[i][j] == 0):
                board[i][j] = choice(['Q', 'R', 'S', 'T', 'U'])
```

› We finally have the whole program finished. We can play it now. And when we do, we probably see some things that we could improve. We can use an iterative improvement process to gradually add on these additional features.

## Reading

Matthes, *Python Crash Course*, chaps. 12–14.

## Exercises

Imagine that you wanted to create a tic-tac-toe game on the computer. Assume that the board spaces are numbered 1 through 9, with the top row numbered 1, 2, 3; the middle row numbered 4, 5, 6; and the bottom row numbered 7, 8, 9.

Show the code you would write for the following pieces of the program.

**1** Define an initial empty tic-tac-toe board, using a character "'.'" to represent an empty square.

2 A function that takes in a board, a position (a number 1 through 9), and a character "'X'" or "'O'" and updates the board to have that value in the appropriate position.

3 A function that takes in a board and examines the first row. If all elements are "'X,'" or all are "'O,'" then that character is returned. Otherwise, "'.'" is returned.

TRANSCRIPT 13

# Game Design with Functions

One of the attractions of computer science, for as long as I can remember, has been games. Video games have evolved a lot from the early days of Pong in the 1970s, into a wide-ranging industry that produces games for desktop computers, video game consoles, and smartphones. Believe it or not, the game industry is now bigger than Hollywood and bigger than the music industry. I find that pretty amazing.

Now, just like making a movie, creating a popular game these days takes a lot of people filling a lot of different roles, including artwork, sound, level design, and production. But, the core of the game is the still the software that controls the game mechanics—what actually happens in the game. And, this part of the game is developed by programmers.

We're going to show how to develop a game similar to a lot of popular casual games. In fact, you've probably played games like this before. We'll see how functions directly support a top-down design approach, and we'll use stub functions to help us rough in the structure of our program along the way.

The game we're going to develop will have the guts for a grid-based matching game. Now, these are games where you have a bunch of objects arranged in a grid, and you try to move things around to match up similar items—at which point the matched-up items disappear. There are lots of variations on these games. Objects on the grid might be pieces of candy—like in Candy Crush, or jewels—like in Diamond Mine, or they could be geometric shapes.

For this lecture, our game will use individual letters for our objects, and we'll see the grid as a text grid printed out to the screen.

Here's the game that we'll develop. We'll have a 2D grid of different objects. In our game, we'll have five objects—the letters Q, R, S, T,

and U. And, we'll have the familiar game mechanics, where objects disappear once we get a certain number of the same object in a row or column.

On each turn, you get to choose to rearrange those pieces somehow. Now, different games have different types of moves allowed. We're going to assume that the only move we can make is to swap a piece with an adjacent piece.

When a move is made, some objects are removed from the grid according to patterns that are made. For our game, we'll remove any cases where three or more of the same object are adjacent in the same row or the same column. Usually, this is what the player gets points for. Then, the remaining objects rearrange, typically by falling down to fill in the gaps that just got removed.

We'll want to fill in gaps at the top with random new objects. The game continues like this until the user meets a goal. For us, the goal will be to get a predefined number of points.

Now, this is going to be a somewhat complex piece of software—it's certainly not the sort of thing that we want to just sit right down and start writing.

So, let's see how we can design this program using a top-down approach, modified for teaching purposes. Let's think about what has to happen in the broadest terms first. At this broadest level, we have three basic steps. First, we set up everything—initializing the game. Second, we go into a loop. The condition for the loop makes sure that it's not time to end the game. Finally, within the loop, we'll go through one round of the game. So, that gives us three more aspects that we need to define. Notice that at the top level, we've defined the main tasks to be done, but not the details of how to do them.

Now, full top-down design would mean that we keep going, defining more and more levels before we actually write the code. But, I want to go ahead and show you right now how some of this might look. Let's

consider that top level first. In code, we can, and should, take each of those steps and put in a comment describing what needs to be done—in what order. In this case, we have three comments. We have one for the initialization, one for the loop, and one for taking a turn. Now, I want to make a key point here.

Each of these general tasks is something that can and usually should be encapsulated into a function. To illustrate here, every time we have one of these tasks, we'll turn it into a function call. Now, this is the main idea of procedural programming, where functions are created to handle all the main tasks. Here's what the current code looks like if we introduce the functions. Notice that each of our original lines has turned into one function call. Those actual functions are defined but don't do anything, since we haven't really gotten to that level yet.

Now again, if we're using top-down design in practice, we would define some of the lower levels first, before writing any of this code. In particular, you'll notice that all of our functions have empty parameter lists. That's because we don't yet understand what information we need at these lower levels, so we don't know what to pass in and out via parameter.

Despite that, this code that you see here is indeed the main program for us. Since we've written some code, what's the next thing we should do with it? Test it.

To test in this case, we really want to see if everything is getting called in order. So, we'll add a few simple print statements in each of the functions. And, for one of the functions, we do need to return True or False, so we'll have to add that line, even though the function's not really doing anything. So, now let's run this. When we run, we do indeed get an initial line that shows that we're in the initialization function. Then, we enter an infinite loop in which we keep seeing that we're checking whether or not to continue and then performing a round of the game. OK, so far so good. We should probably also check to make sure that it's working if that one function returns False. So, we change that line of code, run again, and now we just get two lines—one saying that we're

initializing and one that we're checking whether to continue. And, we never actually do a round since we know that check returns False.

By the way, the term we use to refer to these little functions that are placeholders for something that should be much bigger is stubs. A stub is just a function that doesn't really do what it's intended to do, but it does just enough that everything around it runs. So, if it needs to return a True or a False, it'll do so, but not according to any real computation. When we're developing a program and have a bunch of function calls to make, we say we're stubbing out the program when we write stub functions for each.

OK. Great. We've now looked at our code, and we have something stable that's been tested. Remember how I talked in an earlier lecture about building code in a pyramid fashion one layer at a time? Well, we just implemented our foundation. It's time to add some more layers onto the pyramid.

Now, let me step back for half a second and make sure that we're on the same page. I'm going to be showing you how this code is developed. As I do so, we'll see a good amount of code—about 200 lines overall. When you see code, I want you to make sure that you understand what you see before going on. So please, any time that we're looking at code, make sure you understand what it's doing before going on further.

OK, so now we want to turn back to our design process. Let's look at one of these functions that hasn't been defined yet—say initialization. In initialization, we need to set up the grid itself. That is, we need to get all the pieces placed into their starting positions on the grid. We also have to set the user's score to zero since we're just starting the game. And, we probably want to initialize other variables, such as one that will help us keep track of which round of the game we're on.

Now, for one round of the game, we have three basic steps. We first have to get the move from the user. Then, we have to update the game based on that move, and finally, we have to display the new grid to the user.

This brings us to our continue the game check. It turns out that this is going to be a pretty simple check for our game. We just want to see if the user has reached the goal score yet, or not. So, we'll have a conditional that checks whether the score exceeds some maximum, or not, and return True or False for that routine.

Now, this routine is so simple that each of those commands is basically a line or two of code. So, let's go ahead and implement this routine. Here's what that code will look like. We have an if statement that compares the current score and the goal score, and will return either True or False. Now, do you see anything about this function that's not so great? Well, the thing that strikes me is that we've got these variables—current_score and goal_score—but we never really define them anywhere. They're mystery values defined somewhere outside of the function, and we just have to hope that they're set correctly for when we read them.

So, a better option will be for us to pass in these values as parameters. So, here we list these values as parameters to our function. This is good because it means that the functions calling ContinueGame will know that they need to pass in the important values that ContinueGame needs to operate. Also, notice that we've used a default value for the goal score, so if the calling routine just wants to pass in the current score, it can do so. Now, we need to make sure that we have the right data on the calling side and that we're passing it in.

So, in the main part of the code, we'll make a change to the call to ContinueGame. We set up the score and the goal, and we call ContinueGame with that score and goal passed in as parameters.

Now, you might be wondering—why did we create a whole function to determine ContinueGame? Couldn't we have just written all of this into the while loop where we now call ContinueGame? The answer is yes. We could have done this all in one line, but that's mainly because we have such a simple condition to determine whether or not to keep going. Here's what that would look like.

There are two related reasons why we made ContinueGame a function, rather than just putting the comparison directly into the while loop. First, it followed our top-down design approach. In our design, at first, we just knew that we needed a check on whether to continue. Only as we went farther down in the design did we determine what that check should be. Since we're separating different concepts using function calls, using a function makes more sense.

Second, using a function call leaves us the option of having a much more complicated function to determine ContinueGame. For instance, maybe the user has to not only get a certain score but also has to remove all the objects of a certain type from the grid. We could encapsulate that check into the ContinueGame routine. As far as the main program is concerned, it just needs to make sure that it provides ContinueGame with the parameters it asks for, and it doesn't need to worry about how ContinueGame itself is implemented. This top-down design has given us the flexibility to make much bigger changes later on.

Now, we should test everything, just like I've shown you before. For the sake of time, I'm going to skip showing you all the testing done along the way, and I'll show you correct code each time. But realize that when you actually write this code, it's extremely unlikely that you're going to have all of this work right, right off the bat.

OK, that was only one small part of this whole program. Before we go much farther, we first ought to think about what some of the data we'll want to store is. We need to decide how the grid itself will be represented. This is what we would refer to as a data structure, and ways of defining good data structures is a whole topic of its own that we'll come back to in a future lecture. For now, we can have a pretty straightforward data structure. Basically, we want our grid to have a set of rows and columns, and in each of those rows and columns, we have one object. In our case, an object is just a letter.

Now, can you think of where you've seen something that would let us store rows and columns nicely? How might you store something like this? If you remember, we've talked about lists in the past. In fact, we've

seen a grid representation as a list of lists. Let's look at how this would appear in code.

Let's say that our board is an 8 by 8 grid, like a checkerboard. We'll thus need a list of 8 rows, each with 8 elements. Now, we'll actually set these elements to what's needed for the game in the initialization routine, but to begin with, we'll make a list of all these elements. This is just going to be a list of 8 lists, each of length 8. Now, there're other ways that we could have created these lists, for instance, by creating loops in which we append lists, but I think writing it like this is just as easy, and it makes the structure clear.

OK, now let's turn back to our top-down design and look at some of our other routines. Let's take the initialization routine again. It's going to have three different parts—initializing the grid itself, initializing the score, and initializing the turn. Later on, we might find some other things that we wanted initialized, and so we'll have to add them here, also.

Initializing the grid is a complicated process, and we'll do that in a separate function—Initialize grid. For the score and the turn, we'll want to set the score to zero and the turn to 1. Now, the score and turn variables are going to be trickier to set. These are immutable values, so we can't pass them in and change them in the function. What we can do, though, is we can make sure, within the function, that we declare them as global variables. This will let us initialize them to their appropriate values.

Now, remember globals are often discouraged, but there's not an easy way to get around using them here. Later on, we'll learn about objects, and how they can be mutable, and thus let us initialize data without using globals.

OK, let's consider now how we might handle the grid initialization itself. Notice that since this is a list, it's mutable, and thus we're passing it as a parameter to initialize it. They're different ways we could initialize. For instance, we might have a file that specifies an initial configuration. But, for us, let's just assume that we'll assign random objects to each grid cell.

To assign random objects, we'll need to use the random module, which is part of the Python standard library. We'll import the choice function, which will randomly choose one element from a list. Then, to initialize our grid, we'll loop through all 8 rows and all 8 columns, and set the element to a random value. We will assume the possible objects are the letters *q* through *u*, but we could change those values to anything we wanted. I chose *q* through *u* since those letters appear pretty different from each other visually.

So, that's the initialization stage. We can now turn our design to the game round itself. So let's think about what needs to happen during a turn. There are four basic parts: presenting the state of the game, getting the user's move, determining the result of that move, and finally incrementing the turn number.

Again, the top-down approach means we can create a separate function for each of these main steps. We'll just call these in order from our DoRound routine. One of these, incrementing the turn number, requires just a couple of lines of code, and so we'll actually write that out, rather than creating a whole separate function. And that's it for DoRound—we're done with that routine.

OK, now we turn to the next unfinished portion of our routine—presenting the board. We're just printing to the screen at this point, so we just need to output each of the grid values in a nice orderly format. We'll draw horizontal and vertical lines to separate the individual elements. So, we start by drawing a few blank lines to separate our display from any that might have appeared right above it, and then we'll draw a horizontal line.

After that, we're going to draw stuff for each of the rows. We'll form a single string for drawing each row. Since we want the highest row number drawn first, we'll have a range that's decreasing. Notice that we want to start with row 7, and work our way down to row 0, which means our range will go 7 to –1 by steps of –1. Remember that the last element of the range is the stopping point and doesn't get processed.

OK, for each row we'll have one string that we start out using the empty string. For each column, we'll output a vertical line, followed by the letter designating the object that we have. When we've done this for the entire row, we print out one vertical line to end the string; we print it, and then we print another line of horizontal dashes.

OK, that's our display routine. Let's look now at our GetMove routine. For now, we'll just ask the user for a move and return that move. That's super simple. Notice that we haven't really said what form a move should take, so for now, the move is just a string. We're going to have to work out what form that string should take, shortly.

OK. Let's stop quickly to look at all of our code and check out our progress. We have separate routines for all the various levels we've discussed—from initializing, to checking whether to continue, to doing one round—along with all the routines that those would call. We have one stub function remaining, and that's the Update routine. Of course, that's actually a really important routine. If we run the code at this point, we see a display of the board and can enter a move, but nothing's actually happening.

Now, thinking back to our pyramid model of software construction, we've laid out several layers of the pyramid, and we can test it and make sure that it's all working correctly so far. But, we still have a ways to go. Next, we'll turn to the actual turn mechanics, which is embodied in the Update routine.

The turn mechanics are the main thing that define the game. They embody the rules about how the game progresses according to a move. In this case, there are a few parts, each of which will require us to do something.

First, we'll need to update the board according to our move. In this case, that means swapping one object with an adjacent one. Then, we'll need to repeatedly eliminate pieces and update the board, until there's nothing more to be eliminated. We'll have to go through and remove any pieces that are 3-in-a-row or 3-in-a-column. This'll leave some

empty spaces, and everything above will need to fall down. Finally, any blank spaces at the top will get filled in with new random objects.

Again, putting this into code is straightforward, as each action gets turned into a new function. We stub out these functions, and then we'll address each of those functions individually.

To determine the swapping, we need to convert a move into an actual position, and its adjacent position. So to do this, we need to determine how to express a move. Notice that this decision will affect how we expressed a move when a person typed it in.

In order to express a position, we'll use a system similar to that used in Chess. The columns will be numbered using a lower-case letter from a through h. And, the rows will be numbered using a number from 1 to 8. So, we can express a particular position by a letter-number combination. For example, e3 would be one space, and f3 would be the space just to the right.

Now, our moves must also say what direction we're swapping. To do that, we'll put a single letter after the space to say whether it's swapping up, down, left, or right. We'll use *u*, *d*, *l*, and *r* to designate up, down, left, and right. Notice, then, that if we were to say e3r, that's the same as saying f3l. Also, they're going to be some invalid moves. For instance, nothing on row 1 could swap down, and nothing in column h could swap right.

Here's how this code would look. We need to get the row and column number, each of which is between 0 and 7, from the first two parts of the move. For the row, we'll take the second element of the move, convert it to an integer, and subtract 1. We have to subtract one since our indices run 0– 7 instead of 1–8. For the column, we need to take the first element of the move and convert the letter a through h to a number. We'll use a series of if-then-else statements to do this, and because it's kind of messy, we'll pull that into its own function.

Next, we'll need to get the row and column we're swapping to. That'll be based on the third element of the move. We'll determine the new row

and column from the original row and column, depending on whether it's *u*, *d*, *l*, or *r*. For example, if the swap was up, then the new row is one higher, but the column remains the same. We can make similar rules for *d*, *l*, and *r*.

Finally, we need to actually swap the objects in the two positions The typical way we swap two objects is to copy one to a temporary position and then move the other object into the original position and then use that temporary to fill in the original. Swapping like this is very common.

So, now we have our swap routine fully finished. We've taken in a move, and we've swapped pieces based on that move. We next need to turn to another routine that's just a stub and fill it in. The next one we have is RemovePieces.

Again, following top-down design, let's think about how RemovePieces should work. Now, this is tricky. We need to make sure that we remove any three-in-a-row or three-in-a-column, and we could have both cases. What we'll do is we'll create a new 8 by 8 board that we'll use to keep track of whether a piece should be removed or not. We'll then update this board by looking at the rows and columns to find three of the same object and mark those spaces if they need to be removed. After we've found all the pieces to be removed, we'll go back and remove them. Again, notice that we should mark them first, then remove them. If we remove them when we first find them, then we might miss some additional cases. As we're removing them, we'll increase the score. And, finally, we'll return True or False depending on whether the pieces were removed or not.

Putting this into code takes several lines, but it's pretty straightforward. We first create a board, Remove, filled with 0—where 0 will indicate not to remove, and 1 will indicate we should remove. We then loop through all 8 rows. In each row, we'll go through the first 6 columns and compare that element with the next two columns over. If they match, we'll mark those elements to be removed. We do the same for the columns. Finally, we go through the entire board, and if there's an element marked to

be removed—we change that element to a 0, increment our score, and note that we removed something. Finally, we return the removed value.

The next stub routine to fill in is for dropping pieces. To do this, we'll go to each column and make a list of remaining pieces from bottom to top. We'll then fill in the column with those pieces—putting 0s in at the top.

Again, the actual code for this is relatively straightforward. We'll handle each column separately. In each column, we first make a list of the elements that are non-zero. Then, we go through that list, copying it into the column. Notice that our loop is over the size of that new list. Finally, we fill in the remaining spots in that column with 0s. Notice that our loop range, in this case, starts from the length of the list of remaining pieces and goes to the column length, 8.

We have just one more stub function to fill in. That's filling in any blank spaces with new pieces. In this case, we'll just run through all spaces, and if there's a zero, we'll replace it with a random new piece. So, what should the code for this look like?

This is pretty straightforward code, again. We loop through all the spaces on the board. If we have an if statement to check whether the space is empty, that is, if it has a value of zero. Finally, if it does, we generate a random piece in its place.

Hooray! We finally have the whole program finished. We can go ahead and play it now. And, when we do, we probably see some things that would be nice to improve. For instance, it would be nice to know the current score. We can use an iterative improvement process to gradually add on these additional features. For instance, we can print out the score when we print out the board. And, it's nicer to have the rows and columns labeled. These are relatively simple improvements that make the game easier to play.

We could also give instructions more clearly when getting the move. It would help to give some instructions about how to enter a move. It's a very simple improvement—just adding some print statements.

Maybe we should check to make sure the user enters a valid move. Now, where do you think we could put a change like that? Well, since we're trying to get a valid move, we could make a change to the GetMove function. So, we probably ought to think about how we would redesign this function.

Our function already gives instructions—that's the print statements—and it already gets a move. So the new thing we need to add is a loop to continue asking for a move until the user enters a valid one. We'll need a loop, and within that loop, we'll need to ask for another move. The condition for the loop will be that the current move is not a valid one. Now, that check will probably require a routine of its own. So, let's modify our GetMove function and add some comments. We divide it up into three parts—giving instructions, getting a move, and then looping until the move is valid. We'll also create a new stub function for IsValid to test whether a move is valid. Notice that since IsValid will return True at the moment, this code doesn't actually behave any differently than what we had before.

OK, now, how would you write the IsValid function? Remember, before you just write code—stop and think about what it means for a move to be valid, the categories of things that could go wrong, and then about the various checks that you would need to put in to verify that. Here are three things we could check to see if it's a valid move. First, make sure that they entered a 3-character move. If not, the move's invalid, so we return False.

Next, make sure that the individual parts of the move are valid—that they entered a valid row, a valid column and a valid direction. So, we make sure that the column is between a and h, that the row is between 1 and 8, and that the direction is either *u*, *d*, *l*, or *r*.

And third, we can check that the swap direction is valid for the given row and column. If the piece is in the first column, we can't swap left. And if it's in the last column, we can't swap right. The same thing goes for the top and bottom rows. We've now got three nice checks to make sure that the user doesn't enter invalid moves that could cause the program to crash.

There are other ways that we could improve on the program. Maybe we want to do something more advanced, like give an additional type of move. For example, maybe we want to allow the user to have a bomb that destroys lots of pieces or a laser that removes an entire row or an entire column. Maybe removing certain combinations of pieces gives an additional bonus or a new type of piece. Maybe we change from having five types of regular pieces to having more.

Or, maybe we want to change when the game ends, for example, giving the user a certain number of turns or even building in a loop that lets the user continue playing if their points, after a certain number of turns, exceeds a predefined score.

The point is, now that we have this working version, we could augment it to add any of the features we'd like. Let me encourage you to try this out on your own. Think of some feature you would like to see added, then look for a way to add that feature yourself.

OK. We've written a program almost 250 lines long. About a third of our lines are comments and blank lines, but remember that these are very important to helping us develop and understand our code.

Grid-based game mechanics give us a way of demonstrating how top-down design and the use of functions go hand-in-hand. Top-down design provides a conceptual separation between different levels of design, so that one level doesn't have to worry about those just above or just below it. Functions like DoRound, DropPieces, RemovePieces, and so on ensure that we can put this abstraction into practice. By using stubs, we're able to design, write, and test our code in stages without having to generate every piece of it at once. Then, as we fill in the stubs with the real code, the whole program gradually comes together.

In the next lecture, we'll talk about a different form of software design. One that still makes heavy use of functions, but it's kind of the opposite of top-down—we call it bottom-up. And, instead of games, we'll consider some other fun stuff—graphics and robots. I'm looking forward to seeing you then.

14

# Bottom-Up Design, Turtle Graphics, Robotics

In bottom-up programming and software design, you start with pieces of code you already understand how to use and use those to build upward toward more complex projects. Bottom-up design tends to promote the reuse of ideas and working code from the lower levels, which should yield savings in the amount of work it takes to develop. Bottom-up design works especially well when we already understand our building blocks and when there is no clear or obvious top-down plan for how to build something better. As you will learn, one area of technology where bottom-up software design works well is robotics.

## [ TURTLE GRAPHICS ]

› To illustrate **bottom-up** programming for things like robots, we're going to use a simple module that will let us simulate a robot motion. The name of this is the turtle module, and it's one of the modules installed automatically with the Python standard library. The turtle module lets us create what are called **turtle graphics**, which are relatively simple line drawings but can be lots of fun on their own.

› A simple turtle graphics program in Python looks like the following. It's a program to create a square spiral. From the "turtle" module, we import the "forward" and "left" commands, and then for every *i* within a given range is movement forward and then a left turn. The little shape that moves around the screen and traces out a path as it moves is called a turtle.

```
from turtle import forward, left
for i in range(1,100):
    forward(2*i)
    left(90)
input()
```

- Even though this isn't a real robot moving around, we can treat the turtle like a robot. It'll have some of the same basic commands that a real robot would have.

- For our example, the turtle will have only six basic commands: move forward or backward, turn left or right, and raise or lower a pen it carries. When the pen is down, wherever it moves is traced out as a graphic on the screen. When the pen is up, it moves without tracing an image.

- These commands, and many others, are all part of the "turtle" package. The "forward" and "backward" commands take in a parameter that says how far to move in pixels, where a pixel is just one dot on the screen. Images that you see are made up of a bunch of pixels arranged in a large grid.

- The commands "left" and "right" cause the turtle to turn in place, either to the left or to the right, with the number of degrees to turn passed in as a parameter. The two controls for the pen are simply "pendown" and "penup."

- The turtle will start in the center of the screen, facing to the right, with the pen down.

- These are the most basic commands. For a real robot, you'll often have something similar—a few basic motion commands—that you will have to put together to do something more complicated.

- For example, let's say that we want to draw a square. We can imagine what we need the turtle to do: go forward for a while, turn 90 degrees (counterclockwise), go forward the same amount, and so on, until the square is completed.

› The following is what this will look like in code.

```
from turtle import forward, backward, left, right, penup, pendown
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
input()
```

› First, notice that we can use a "from turtle import *" command to get all six commands from the turtle module. We'll make the square 100 units long on each side. So, drawing the square means that we move forward 100 units, turn 90 degrees, etc., until we've drawn all four sides. We are going counterclockwise, so we turn left.

› When we run the program, the turtle goes around and draws all four sides, creating a square. The turtle (the triangle-looking thing) is back at the center of the screen, although now it's facing down instead of to the right.

› Here's where bottom-up design comes into play. We just created a sequence of code that will create a square. We can package those commands together to create a new routine called "drawSquare." We simply define a function, called "drawSquare," and put the code we just wrote into the body of the function. Then, when we call the "drawSquare" function, we get the same behavior as before.

```
from turtle import forward, backward, left, right, penup, pendown
def drawSquare():
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
```

```
    left(90)
    forward(100)
drawSquare()
input()
```

- This is an example of bottom-up design. We took some simple things that we already knew how to do—in this case, moving forward and turning left—and we put those together to create something more complicated—in this case, making a square.

- If we call the "drawSquare" function a second time, it creates a second square, just below the first one. Remember that when we finished drawing our square, the turtle was pointing down, instead of to the right. So, when we called "drawSquare" a second time, it drew another square. For both the first and second square, the square was drawn to the front left of the direction the turtle was originally facing.

- There are several ways we can improve the square program. And we can also create other shapes, such as a triangle or rectangle. And if you explore some of the other turtle commands listed in the library, you can get other features.

## [ ROBOT PROGRAM ]

- One great, if perhaps surprising, way to think about the turtle library is as a good proxy for robot motion. So, we're going to look at how we could control a robot to have it explore a room—the same way a robot vacuum cleaner might, for example. We'll assume that we have the basic turtle commands—forward, backward, turn left, etc.—and will build from the bottom up from those basic commands to define the robot's paths to cover a whole room.

- In addition to movement, most modern robots also have sensors. Sensors can help detect if there's a potential problem or some other event. For example, mobile robots will often have sensors to detect how close a wall is or if they've bumped into something.

- Our turtle is obviously not a real robot, and it does not have sensors. However, we can just define a sensor function that will act like a sensor for our on-screen turtle.

- This sensor we define can tell us if we're too close to an obstacle. If we call "sensor," it should return "True" if we're too close to an obstacle or "False" if we're not. So, the sensor we define is very similar to proximity sensors that you could find on a real robot.

- Our code will start by importing two modules. First, we're going to be using turtle pretty heavily, so we'll import all the turtle functions, indicated with an asterisk. Second, we're going to want some random functions, so we'll import the random module, too. Next, we'll set up variables to say that the room we're operating in is a simple square, going from -250 to 250 in both *x* and *y*. And we'll assume that we should say we're too close to the edge if we're within a proximity of 10.

```
from turtle import *
import random
xmax = 250
xmin = -250
ymax = 250
ymin = -250
proximity = 10
def sensor():
    if xmax - position()[0] < proximity:
        #Too close to right wall
        return True
    if position()[0] - xmin < proximity:
        #Too clsoe to left wall
        return True
    if ymax - position()[1] < proximity:
        #Too close to top wall
        return True
    if position()[1] - ymin < proximity:
        #Too clsoe to bottom wall
        return True
```

```
        #Not too close to any
        return False
```

- The sensor function itself will use the position command from the turtle library to compute how far away the turtle is from each of the walls. If the turtle is within proximity of any of the four walls, the sensor function returns "True," indicating that the sensor had triggered. Otherwise, it returns "False."

- Robot vacuums typically have just a few basic types of motion. It can travel in an ever-increasing spiral, and in fact it usually starts out in a spiral. It travels in a straight line, in some seemingly random direction. It also moves parallel to a wall that it is close to.

- We're going to try to build up these patterns for our turtle. For all of them, we only want to continue the pattern until the sensor triggers a proximity warning, at which point we have to do something else.

- Let's start by thinking about how we'd build the easiest of these—traveling in a straight line in a random direction. How would we use our "forward" and our "left" or "right" commands to pick a random direction, and then head in that direction, until the sensor triggered? Remember that we have the random module available to us, too.

- The following is one way to describe this. We define a function named "straightline." Notice that we've included a docstring, stating what the function does. The first action is to pick a random direction to go.

```
def straightline():
    '''Move in a random direction until sensor is triggered'''
    #Pick a random direction
    left(random.randrange(0,360))
    #Keep going forward until a wall is hit
    while not sensor():
        forward(1)
```

- The turtle will turn left by some random amount between 0 and 360 degrees. We use "randrange" from the random module to pick the number of degrees and pass this to the "left" function. The second part of the function just continues in a straight line until the sensor returns "True." Notice that we only move forward one unit at a time before we check the sensor again.

- If we run this code, we see that the turtle heads off in some random direction, until it hits the "edge" of the square room that it's in.

```
def straightline():
    '''Move in a random direction until sensor is triggered'''
    #Pick a random direction
    left(random.randrange(0,360))
    #Keep going forward until a wall is hit
    while not sensor():
        forward(1)
    straightline()
```

- Next, let's define a spiral function. We could use the spiral that we defined earlier, but that's a square spiral—it would be better to have something more circular. Mathematically, this is going to be trickier.

- The following is a spiral function we could use. We've defined a parameter, called "gap," that will tell us how tight the spiral should be. We set a default value in case we don't want to actually specify it, though. The way the spiral works is that at any one time, we pretend we're on a circle of some radius, and we move one unit along that circle's circumference. Then, we increase the radius so that it increases slightly with every step. We keep doing this until our sensor function says it's time to stop.

```
def spiral(gap = 20):
    ''Move in a spiral with spacing gap'''
    #Determine starting radius of spiral based on the gap
    current_radius = gap
```

```
    while not sensor():
        #Determine how much of the circumference 1 unit is
        circumference = 2 * 3.14159*current_radius
        fraction = 1/circumference
        #Move as if in a circle of that radius
        left(fraction*360)
        forward(1)
        #Change radius so that we will be out by 2*proximity
          after 360 degrees
        current_radius += gap*fraction
```

› The code shows how the math works.

› If we run this code, we see that the turtle indeed is going to spiral out.

```
def spiral(gap = 20):
    '''Move in a spiral with spacing gap'''
    #Determine starting radius of spiral based on the gap
    current_radius = gap
    while not sensor():
        #Determine how much of the circumference 1 unit is
        circumference = 2 * 3.14159*current_radius
        fraction = 1/circumference
        #Move as if in a circle of that radius
        left(fraction*360)
        forward(1)
        #Change radius so that we will be out by 2*proximity
          after 360 degrees
        current_radius += gap*fraction
spiral()
```

› How might we build a pattern for wall-following? In this case, we'll want to find which of the four walls is closest and then set our direction to be parallel to that wall. Note that we'll want to use the turtle function named "setheading," which allows us to give a direction: 0 is to the right, 90 is up, 180 is left, and 270 is down.

› The following is one way to define a function we can call "followwall."

```
def followwall():
    '''Move turtle parallel to nearest wall for amount
      distance'''
    #find nearest wall and turn parallel to it
    min = xmax - position()[0]
    setheading(90)
    if position()[0] - xmin < min:
        min = position()[0] - xmin
        setheading(270)
    if ymax - position()[1] < min:
        min = ymax - position()[1]
        setheading(180)
    if position()[1] - ymin < min:
        setheading(0)
    #Keep going until hitting another wall
    while not sensor():
        forward (1)
```

› At this point, we were able to build up three different motion patterns: random straight line, spiral, or wall-following. Let's build up from here to create a new routine, "backupspiral," which will move us backward for some amount and then spiral outward.

› The following is the code.

```
def backupspiral(backup = 100, gap = 20):
    '''First move backward by amount backup, then in a spiral
      with spacing gap'''
    #first back up by backup amount
    while not sensor() and backup > 0:
        backward(1)
        backup -= 1
    #Determine starting radius of spiral based on the gap
    spiral(gap)
```

› Now we have several different motion patterns. We can put these together to build up a plan to explore a room. Imagine again that the turtle is a robot vacuum that's going to just keep going around cleaning up. We want something that will keep picking one of these motion patterns at random and using it to explore the room. The following is one way to implement this.

```
speed(0)
#Start with a spiral
spiral(40)
while (True):
    #First back up so no longer colliding
    backward(1)
    #Pick one of the three behaviors at random
    which_function = random.choice(['a', 'b', 'c'])
    if which_function == 'a':
        straightline()
    if which_function == 'b':
        backupspiral(random.randrange(100,200), random.
          randrange(10,50))
    if which_function == 'c':
        followwall(random.randrange(100,500))
```

› If we run this code, we start out in a spiral. One we've spiraled all the way out to where we come in proximity with a wall, we start taking random motions, according to one of our three patterns.

```
speed(0)
#Start with a spiral
spiral(40)
while (True):
    #First back up so no longer colliding
    backward(1)
    #Pick one of the three behaviors at random
    which_function = random.choice(['a', 'b', 'c'])
    if which_function == 'a':
        straightline()
```

```
if which_function == 'b':
    backupspiral(random.randrange(100,200), random.
      randrange(10,50))
if which_function == 'c':
    followwall(random.randrange(100,500))
```

## Reading

Zelle, *Python Programming*, chap. 9.

## Exercise

Write a function, "drawA," using turtle commands to draw the letter *A*.

Hint: Make the sides of the *A* at a 60-degree angle to the horizontal. This will make the shape of the *A* an equilateral triangle, which may be easier to draw.

Suggestion: Make the turtle finish in its original orientation, shifted over slightly from the last point on the *A*.

14 TRANSCRIPT

# Bottom-Up Design, Turtle Graphics, Robotics

Once you understand how to write functions and how to use modules, you're ready for a style of programming and software design called bottom-up. In bottom-up, you start with pieces of code that you already understand how to use, and you use those to build up toward more complex projects. As you might guess, bottom-up design is kind of the opposite of top-down design, where we started with an overall goal and broke it up into its different parts.

To see how bottom-up works, think back to the analogy we used for top-down design: planning a dinner. The top-down approach to this problem is to say, "I'm planning a dinner, so I'll need an appetizer, a main dish, and a dessert," and so on. However, another way to approach cooking is to say, "OK, I have some flour, sugar, and so on, and I know how to mix ingredients into a batter, and I also know how to bake, and I know how to make frosting. So, if I combine all of those together, I can create a delicious dessert." Only later might you be in a position to say, "Well, I know how to make a dessert, and also an appetizer and a main course. If I combine all of these, I can put together a great meal."

Now, bottom-up design has one big advantage—you're starting from common patterns that you already know how to use, and that lets you reuse ideas. In the dinner example, you might be able to use the fact that you know how to cook pasta to create a pasta salad appetizer, or a spaghetti and meatballs main course, or even make something like noodle pudding for desert. So, your ability to cook pasta could potentially get reused many times, across all three courses.

When we're writing code, bottom-up design means that we can take sections of code that we know work, and we can combine them together to get something that has new functionality. Bottom-up design also tends

to promote the reuse of ideas and working code from lower levels, and that should yield savings in the amount of work it takes to develop.

Bottom-up design works especially well when we already understand our building blocks, and when there's no clear or obvious top-down plan for how to build something better. Early voice-controlled digital assistants on smartphones are an example. I'm talking about things like Siri, the pleasant voice on an iPhone, that you talk to in order to find out things like how late the oil change place is open, or where you can get a good sandwich for lunch.

Now, putting together a voice-controlled digital assistant took a lot of parts, but most of those parts already existed before Apple made Siri. There was already voice recognition software. There was already the ability to run a search. There was already software that could convert text into speech. Now, I don't mean to minimize the work that it took to make a great product; there was a whole lot of effort and ingenuity that was needed to bring all those pieces together. But that's the point—by starting with some great pieces that already work, a bottom-up approach can create something that's even better.

Another area of technology where bottom-up software design works well is robotics. Now, to illustrate bottom-up programming for things like robots, we're going to use a simple module that'll let us simulate a robot motion. The name of this is the turtle module, and it's one of the modules installed automatically with the Python standard library. The turtle module lets us create what are called turtle graphics, which are relatively simple line drawings, but can be lots of fun on their own. The first turtle programs, dating all the way back to the mid-1960s, were actually based on a physical object that moved around like a robot, but turtle graphics has been adapted for use in various programming languages such as Logo, a teaching language, and Python.

So, let's see what a simple turtle graphics program looks like in Python. Here's a program to create a square spiral. It's just a few lines of code, but you can get an idea of what's going on. From the turtle module, we import the forward and left commands, then for every *i* within a given range, we

move forward and then turn left. The little shape that moves around the screen and traces out a path as it moves is what we call the turtle.

Now, even though this isn't a real robot moving around, we can treat the turtle like a robot. It'll have some of the same basic commands that a real robot would have. For example, the turtle will only have six basic commands: move forward or backward, turn left or right, and raise or lower a pen that it carries. When the pen is down, then wherever it moves is going to trace out a graphic on the screen, and when the pen is up, it moves without tracing an image on the screen.

These commands, and many others, are all part of the turtle package. The forward and backward commands take in a parameter that says how far to move in pixels, where a pixel is just one dot on the screen. Images that you see are made up of a bunch of pixels, arranged in a grid. The commands "left" and "right" cause the turtle to turn in place, either to the left or to the right, with the number of degrees passed in as a parameter. The two controls for the pen are simply "pendown" and "penup." The turtle will start in the center of the screen, facing to the right, with the pen down.

Now, these are our most basic commands. For a real robot, you'll often have something similar, a few basic motion commands that you'll have to put together to do something more complicated. So, let's see how we can put these together to do something more complicated. Let's say that we want to draw a square. We can imagine what we need the turtle to do: we want it to go forward for a while, turn 90°, go forward the same amount, and so on until the square is completed. And, by the way, I'm going to be going counterclockwise in my examples because computer graphics has traditionally used counterclockwise orientations. The reason for that, which you don't need to understand for this lecture, is so that cross products between convex edges, defined by the right-hand rule, will generally point up.

Let's see what this will look like in code. First, notice that we can use a "from turtle import" command to get all six commands from the turtle module. We'll make the square 100 units long on each side. So,

drawing the square means that we move forward 100 units, turn 90°, et cetera, until we've drawn all 4 sides. We're going counterclockwise, so we turn left.

Notice that I put a line, "input," at the end. This is just to keep the program running until I press a key. If you don't put something like this in, the graphics screen will draw, but then the program will finish, and it'll immediately disappear. And so, by waiting for input, it'll keep the image on the screen just until I type something in or stop the program.

All right, so now when I run the program, I see that the turtle goes around and draws all four sides. We've created a square. The turtle—that's that little triangle-looking thing—is back at the center of the screen, though notice that now it's facing down instead of to the right.

Here's where bottom-up design comes into play. We just created a sequence of code that'll create a square. Now, we can package those commands together and create a new routine called "drawSquare." We simply define a function called "drawSquare," and we put the code that we just wrote into the body of the function. Then, whenever we call the "drawSquare" function, we get the same behavior as before.

This is an example of bottom-up design. We took some simple things that we already knew how to do—in this case, moving forward and turning left—and we put those pieces together to create something more complicated—in this case, making a square.

What happens if we call the "drawSquare" function a second time? Well, it creates a second square, just below the first one. Remember that when we finished drawing our square, the turtle was pointing down, instead of to the right. So, when we called "drawSquare" a second time, it drew another square. For both the first and second square, the square was drawn to the front left of the direction that the turtle was originally facing.

Let's look at a couple of ways we can improve the square program. First, maybe we don't want the turtle to face a different direction at the end than it did at the beginning. As you develop more code, you'll see that

it's often good to finish something by returning to as near a state as you can to where you started out. So, if we have a function that should draw a square, it would be nice to just draw the square and have the turtle right where it was when we began. So, to draw this square to the screen and return to our original state, we'll add in one more left-hand turn at the end of the function. If we run the code now, it'll draw the square, and the turtle ends up pointing to the right, just like it started. So, calling "drawSquare" again will just trace right over the previous square.

OK, how about another improvement? I picked a square size of 100 in this case, but what if you wanted a different size? How might you modify the code to easily allow that? Well, we can modify the "drawSquare" routine to take in a parameter that we'll name "size." Then, when we draw each edge, we'll use our size parameter to draw it using whatever value we've specified in the line where we called "drawSquare." Notice that we can set a default value for size of 100, in case a user doesn't want to specify it. However, the user, in this case, did specify a size of 50.

You can imagine doing something similar to create other shapes. How about a triangle? Try making an equilateral triangle, where the length of a side is passed in. Now, as a hint, think about how much you would need to turn each time as you're drawing. Well, we can define "drawTriangle" to have sides the same length as before. So, the code is basically the same as to draw a square, but we're going to be turning 120° each time, and obviously we're only going to draw three edges.

Now, how about a rectangle? How could you create a routine to draw a rectangle when both the horizontal length and the vertical height are needed? Here, instead of one parameter for size, we can take in two parameters: one for length, one for height. Then we draw, just like for a square, but we make sure to draw using length for the horizontal motions, and height for the vertical motions.

Each of those shapes can be encapsulated into its own function, and then we can build up from there. Say we wanted to build something like a house picture, just a square with a triangle on top—that'd be a good start. We could create a "drawHouse" function that first draws a square,

and then moves the turtle up to the top of the square, and then draws a triangle. See if you can write something like this. Remember, the shape will always be drawn assuming the turtle is pointing to the right, so if you move the turtle, you want to make sure it's pointing in the right direction before you draw.

OK. Notice that our "drawHouse" function includes "drawSquare" and "drawTriangle," with commands for transitioning between them in the middle. Here, we draw the square first. Then, we reposition the turtle by turning 90° left, going to the top of the square, and turning 90° right. Then, we draw the top, by calling "drawTriangle." And then, finally, we reposition back to the start, by turning right 90°, moving to the bottom of the square, and turning back left 90°. Now, have some fun with this; you can get the turtle system to draw all sorts of shapes for you. And, if you explore some of the other turtle commands listed in the library, you'll see you can get other features like filled-in areas.

Let's look at another example of how we can use bottom-up design and turtle graphics to build up a program. This time, our end goal will be a program where we read in a number from a user, and we convert it into groups of five, where we have four vertical lines crossed by a single line that's diagonal. Now, these are called tally marks, or slash marks, or hash marks—the name varies. So, someone entering the number 13 would have two groups of five, followed by three more.

OK, using bottom-up design, let's see if we can put together the commands we need to represent numbers with tally marks. A single tally mark is simple—just a vertical line. To make a vertical line from the turtle's starting position, we'll turn left 90° and then move forward. I'll make the tally mark 20 pixels tall. Now, after making the tally, I need to return back to my original position. So, in this case, I'll just go retrace my steps by going backward and then turn to the right. Let's run this and test it. Sure enough, it creates a vertical tally mark.

OK, now, let's say that we want to create a second tally mark. We need to move over to the right a few spaces and then draw another tally. When we move right, we don't want to actually draw a line, we just want

to move over. So, what we'll do is we'll tell the turtle to pick up the pen, then move five spaces to the right, and then put the pen back down. After that, we can draw another tally mark. Again, we have some new functionality. In this case, we're shifting over to the right, and so we can incorporate that into a function of its own called "shiftRight." Since we're building bottom-up, another option is to incorporate the right shift into the tally mark drawing routine. In this case, the turtle automatically moves over once the tally mark is drawn.

Let's see what other functionality we can put together. Normally, as we're tallying, every five tally marks will be marked with a diagonal slash through the previous four. So, let's see how we can create that diagonal slash. After our last tally, we've already moved over one space, so this is where we'll put the end of the slash. Let's think about how we would want the slash to look. If we don't want the slash to go all the way from the bottom, we need to go up a little bit first. So, we'll turn left, pick up our pen, and go up by a little bit—say, 3 pixels. Then, we'll put our pen down so that we can draw our slash backward through the previous four slashes.

Let's think here a little more about how to draw that diagonal line. If the slashes are 20 units tall, and we want to start and stop 3 units from the ends, we want a slash mark that's 14 units tall in the vertical direction. We're also starting 5 units from the right-hand tally mark, and so we want to go 5 units past the leftmost tally mark. The tally marks themselves are a total of 15 units apart, so we're going to go 25 units in the horizontal direction.

But, instead of writing a function to calculate the distance and angle, which can introduce some rounding errors, we could instead just look at our current position in the graphic, using the "pos" function. Position returns the *x, y* value of the turtle, so we can get the *x* value by taking position-sub-0, or the *y* value by taking position-sub-1. Then, the "goto" function will move us precisely to the new location on the coordinate system, which we can reach by moving –25, 14 away from the current position. Finally, we'll return to our starting location by undoing those motions, going back to the starting position, and then moving back to

the bottom of the screen. And, last but not least, we'll shift over to the right again. In fact, to leave a gap for the next set of tallies, we'll do a double shift over. Now, if we test this out, drawing four tally marks and a slash, we see that the code performs just like we'd hope.

Now, we can build up from here a little bit more. It seems useful to create a single function to draw five tallies, so we'll put that into a function of its own. Once we can draw a group of five tally marks, we can repeat that as many times as needed. We'll create a function, "drawTallies," that takes any number in as a parameter, *n*. First, we'll be drawing as groups of five tallies over and over, until we can't draw any more. Sounds like a loop, right? So, one way that we can do this is to have a "while" loop that continues as long as the number that we pass in is, at least, five. Each iteration of the loop, we'll draw five tallies and reduce *n* by 5. When that loop finishes, *n* is 4 or less. So, we have some single tally marks still to draw at the end, and we can create another loop to draw all the single tally marks, reducing *n* by one each time, and this will continue as long as *n* is greater than or equal to one.

We can ask the user for the total number of tallies to draw. Then, we just call "drawTallies" for that number. If we run this code, we see that it works just fine. For example, let's type in 23. We see that we get four groups of five, plus three more tallies.

Notice what we've done here. We took very basic commands, and we used them to build up more complex functionality. We started with the very simple turtle commands—forward, backward, left, right, penup, and pendown—and we used those commands to get some slightly more complex operations: making tally marks, shifting over, and making a diagonal slash. Then, we combined those commands together to make something even more complicated, a function that lets us input any number and see the tallies drawn for it.

This is typical for bottom-up design. If you think about it, bottom-up design isn't necessarily that different from top-down design. Both can give your code a very well-organized structure, and both have different levels of detail. In the end, the code might not look all that different

from code developed using a top-down approach, but the mindset in developing it is somewhat different. Bottom-up also makes it easier to see how certain functionality can be reused. For instance, in the example we just saw, we used the "shiftRight" command in a couple of different places—after drawing an individual tally mark, and again, twice, after drawing a slash mark. Turtle graphics can be a whole lot of fun, so I'd encourage you to try playing with it some more. The turtle module contains 175 different commands that you can use.

Again, one great, if perhaps surprising, way to think about the turtle library is as a good proxy for robot motion. So, for the rest of the lecture, we're going to look at how we could control a robot to have it explore a room, the same way a robot vacuum cleaner might. We'll assume that we have the basic turtle commands—forward, backward, turn left, et cetera—and we'll build bottom-up from those to define the robot's paths to cover a whole room. In addition to movement, most robots these days also have sensors. Sensors can help detect if there's a potential problem or some other event. For example, mobile robots will often have sensors to detect how close a wall is, or if they've actually bumped into something.

Now, our turtle is obviously not a real robot, and so it doesn't have sensors. However, we can define a sensor function that will act like a sensor for our little on-screen turtle. This sensor that we define can tell us if we're too close to an obstacle. If we call "sensor," it should return true if we're too close to an obstacle, or false if we're not—that's all. So, the sensor that we define is very similar to proximity sensors that you could easily find on a real robot. Isn't programming cool?

So, our code is going to start by importing two modules. First, I'm going to be using turtle pretty heavily, so I'll import all the turtle functions, indicated with an asterisk. Second, I'm going to want some random functions, so I'll import the random module also. Next, I'll set up variables to say that the room we're operating in is a simple square, going from –250 to 250 in both *x* and *y*, and I'll assume that I should say I'm too close to the edge if I'm within a proximity of 10.

The sensor function itself will use the position command from the turtle library to compute how far away the turtle is from each of the walls. The sensor function, then, is pretty simple. It computes how far away the turtle is from each wall, and if it's within proximity from any of the four walls, it returns true, indicating that the sensor had triggered. Otherwise, it returns false.

OK, so we have a sensor that we can check any time we want, just by calling "sensor." If you've ever watched a robot vacuum, you might have noticed that it typically has just a few basic types of motion. It can travel in an ever-increasing spiral, and in fact, it usually starts out in a spiral. It can also travel in a straight line, basically a random direction. And, finally, it can travel right along a wall, parallel to the wall. So, we're going to try to build up these patterns for ourselves, for our turtle. For all of them, we only want to continue the pattern until the sensor triggers a proximity warning, at which point we have to do something else.

OK, so let's start by thinking about how we'd build up the easiest of these—traveling in a straight line in a random direction. How would we use our forward and our left or right commands to pick a random direction, and then head in that direction until the sensor triggered? Remember that we have the random module available to us also.

OK, here's one way that we could describe this. We define a function named "straightline." Notice that I've included a docstring, stating what the function does. The first action is to pick a random direction to go. So, the turtle will turn left by some random amount between 0 and 360°. We'll use and range from the random module to pick the number of degrees, and then we pass this to the "left" function. The second part of the function just continues in a straight line until the sensor returns true. Notice that we only move forward one unit at a time before we check the sensor again. We wouldn't want to travel too far, or we might accidentally crash. Now, this is essentially saying that we're going to keep monitoring our sensor as we move, and stop as soon as it goes off. Now, if we run this code, we'll see that the turtle heads off in some random direction until it hits the edge of this square room that we're in.

OK, that function was pretty straightforward. Now, let's define a spiral function. We could use the spiral that I defined earlier, but that's a square spiral—it would be better to have something more circular. Mathematically, this is going to be trickier. Here's a spiral function that we could use. I've defined a parameter, called gap that will tell us how tight the spiral should be. I set a default value in case we don't want to actually specify it. The way that a spiral works is that, at any one time, we going to pretend that we're on a circle of some radius, and we're going to move one unit along that circle's circumference. Then we increase the radius so that it increases slightly with every step, and we keep doing this until our sensor function says it's time to stop.

Now, the code shows how the math works. We compute the circumference, and then we figure out what fraction of the circumference one unit is. That gives us the fraction of 360° that we need to turn. So, we turn left by that amount, and we go forward one unit. We then increase the radius for the next step. We know how much to increase by taking the fraction and multiplying by the gap. So, if we run this code, we see that the turtle is indeed going to spiral out.

OK, we've built up two of the three motion patterns we've seen before. So how might we build a pattern for wall following? In this case, we'll want to find out which of the four walls is closest and then set our direction to be parallel to that wall. Note that we'll want to use the turtle function named "setheading" which allows us to give a direction—zero is to the right, 90 is up, 180 is left, and 270 is down. So, given that, and having already seen how we computed distance to the walls in the sensor function, how might we set up this wall-following function?

Here's one way to define a function that I've called "followwall." Notice that what we do is we first set up motion parallel to the right-hand wall, and store the distance to that wall in a variable: min. Then, we check the distance to each of the other walls. If they're closer than the closest one so far, we set our heading parallel to that wall, and we update minimum. So, in the end, we'll be facing parallel to the nearest wall, and if we're in a corner, it'll be parallel to one of the two walls that are nearest. Then, we just keep moving forward, just like we were in the "straightline" in a

random direction function, until the sensor tells us that we're too close to a wall.

So, at this point, we were able to build up three different motion patterns: a random straight line, a spiral, or a wall following. So, let's build up from here to create a new routine: "backupspiral." Now what this will do is it will move us backward for some amount, and then spiral outward. You can see the code here—we just try to move backward for a while before starting the spiral. Now, the reason for having this routine is that it will sometimes help us to move away from an edge before we start a spiral so that the spiral doesn't just end right after it begins. Again, we'll set some default value for the backup amount.

So, at this point, we have several different motion patterns. Imagine that you're at a ministry of silly robot motions, and you want to create something silly. See what you can come up with.

We can put these together to build up a plan to explore a room. Imagine again that the turtle is a robot vacuum that's going to just keep going around, cleaning up. We want something that'll keep picking one of these motion patterns at random and using it to explore the room. Now, think about how you might do this yourself. How would you choose to combine these pieces we have to explore the room? One thing to keep in mind—when you have a sensor triggered, you probably want to go backward one step before trying anything else, otherwise you'll start out already in proximity of the wall, and you'll probably stop immediately.

Well, this is the way I chose to implement it. I set the speed to zero. If you've looked at the turtle module documentation, zero is the way to tell the turtle, "Move as fast as you can." So, I start out by spiraling out from the starting position. When the spiral stops, it means that we've hit a sensor warning, so we're near a wall. Now, we have an infinite loop, the first part of which will bring us backward one unit—that should get us away from whichever wall we got too close to. Then, we select one of our three motions using "which_function," whose value is assigned by the "random module.choice" command. We either go in a straight line in a random direction, we back up and then do a spiral, or we follow along

a wall. Notice that if we're going in a spiral, we have two randrange commands for setting a random amount we'll back up—and I just chose for it to be between 100 and 200—and a random gap between spiral loops—and I chose that to be between 10 and 50.

So, if we run this code, we start out in a spiral. One we've spiraled all the way out, we come to proximity with a wall, and then we start taking random motions according to one of our three patterns. We could have picked a different way to combine these basic elements together—this was just the method that I chose. You can try experimenting. For example, maybe don't allow a spiral. Or see what happens if you only follow random straight lines.

Programming for turtle graphics has deep similarities with programming for control of robot motion. Both use many of the same commands, and both benefit from a bottom-up process for software design. Anytime someone is working with existing things to bring together something brand new; it's an example of bottom-up design. It's also a terrific approach for beginners to build their skills in iterative fashion, building on what you already know to put together more complex projects. So, I encourage you to practice a bottom-up approach whenever you can—it's a great way of generating new ideas.

In the next lecture, we'll see how we can go beyond the turtle to develop even fancier graphics, the type you might use in the graphical user interface of a game or in productivity software.

# 15 Event-Driven Programming

A picture is worth a thousand words. Actions speak louder than words. And the same can be true in programming and computer interfaces: What we see in a graphical user interface (GUI), and what we do inside that graphical interface, can be more important than words. In this lecture, you will explore this visual, action-oriented style of programming: how to write a graphical user interface and how to use event-driven programming, a style of programming that responds to mouse clicks and other events within the graphical interface. To do this, you will be introduced to a package called pyglet.

## [ PYGLET ]

- Pyglet is a Python package created to help support development of games and other audiovisual environments. It provides functions that let you create windows, display images and graphics, play videos and music, get input from the mouse, etc.

- There are a few other game-development packages people also use—pygame is one that's well known—and there are other modules that do individual things that pyglet does, but pyglet packages these functions together nicely and is easy to install. Just go to the pyglet site (bitbucket.org/pyglet/pyglet/wiki/Home), or you can find pyglet through a web search.

- If you're using pip to install modules such as pyglet, you should be able to install pretty easily. Remember that you can go to the command line, to the directory where Python is installed, and type "python –m pip install pyglet" and it should install for you.

› Once you have pyglet downloaded, be sure to run the command: "import pyglet." Only if you don't have pyglet installed will you see a response, so any error message probably means that pyglet hasn't yet installed properly.

## [ EVENT-DRIVEN PROGRAMMING ]

› One form of graphics that is familiar to everyone on a computer these days, in practice if not by name, is the **graphical user interface** (GUI, or "gooey"). The way a GUI works is entirely based on what is called **event-driven programming**.

› To understand the contrast, let's think about how we've been programming up to this point. We've been writing commands, and we expect to start at the first command and follow the commands in sequence, one after the other. Things like conditionals, loops, or function calls might make us jump to a different line of code, but we're basically going along in a definite sequence, where we always know which line of code we will execute next. We can refer to this as a sequential program.

› Event-driven programming is different. Instead of the program deciding when to ask a user for input, events outside the program determine what the program does next. An event is anything that happens where we want the program to respond. In a GUI, an event might be not only pressing a key on the keyboard, but also clicking a button on the screen, entering data into a box on the screen, moving the mouse, etc. For each of those events, the computer program needs to respond—it needs to do something—maybe just update a variable or maybe print to the screen.

› Robots often use event-driven programming, which allows them to respond to events in their environment. For a robot, events might be data that comes in from a sensor—for example, the robot detecting it's about to hit a wall. When the sensor gets this information, it needs to send it to the program to respond.

› The same kind of monitoring is always underway in the GUI of a computer. In any kind of event-driven programming, whenever the program runs,

there is an **event monitor** that runs continuously in the program. There are many other terms for the event monitor—such as a main loop, or an idle function, or a control function—but the job of the event monitor is to take in events and make sure that the appropriate function gets called in response to the event.

› There's more than one way to monitor events. Sometimes the event monitor uses what are called "interrupts." Basically, it just sits there until something interrupts it with an event. Other times, the event monitor actively "polls" the various devices—that is, it actively checks to see whether there is a keyboard event, a mouse event, etc. But you don't have to worry about whether polling or interrupts are monitoring events in your own programs—just know that the event monitor is indeed going to be getting the events.

› When the event monitor gets an event, it needs to do something to handle the event. In a GUI, if someone clicks the mouse, the event monitor should call whatever function has been designated to handle mouse clicks. These functions that get called are known as "event handlers."

› The event handler's main job is to take in events and then call what is known as a **callback function** corresponding to that event. The callback is just a function to execute in response to an event, at which time we go back to the event handler to get the next event.

› To write an event-driven program, there are a few stages. The callback functions have to be defined on their own. These are defined just like other functions. The only difference is that different callback functions have to be ready to handle the appropriate type of parameters for the type they are. For example,a callback that handles a key being pressed on the keyboard needs to be able to take in the key that was pressed as a parameter.

```
#define functions to be used as callbacks
#initialization:
    #set up any variables, data, etc.
    #register callbacks
    #start up event monitor
```

- For the main body of the program, there's usually some initialization work that's done, just like any sequential program. As part of this initialization, callbacks need to be "registered." "Registering" a callback function is how we say what function will be called for each event that we want to respond to. So, we have to write functions for each possible event. The final step of the sequential part of the program is to start up the event monitor.

- Once the event monitor is started, it keeps running indefinitely. As events occur, it keeps calling the different callback functions. The real actions of the program happen in the callback functions.

- There's not a single, universal way that event-driven frameworks work. The way the callbacks are registered, the parameters they need to take in, and the way the event handler is started will all vary depending on what event-driven framework you're using.

- In Python, the pyglet framework is great for event-driven programming. Other frameworks are structured somewhat differently.

- Let's look at a basic pyglet program. After we download pyglet, we import pyglet using the import command. Then, we can set up a window. This window command is pyglet.window.Window, and that lets us pass in parameters that define the window. In this case, we set width to 400 and height to 300 and a caption of "TestWindow." Finally, we have the command pyglet.app.run. This last command is there to start the event handler.

```
import pyglet
window = pyglet.window.Window(width=400, height=300,
  caption="TestWindow")
pyglet.app.run()
```

- When we run this, we get a window of size 400 pixels by 300 pixels, and the name on the window is "TestWindow."

- After we set up a window, we need to register some callback functions. In addition to keyboard commands, pyglet can get input from the mouse and display images.

› We can use this functionality to develop the grid-based game we developed in Lecture 13—the one where we would move some letters around and if you had three in a row or in a column, they'd disappear. With the tools in pyglet, we can easily make a graphical version of the game.

```
from random import choice
import pyglet
window = pyglet.window.Window(width=400, height = 450,
  caption="GameWindow")
Im1 = pyglet.image.load('BlueTri.jpg')
Im2 = pyglet.image.load('PurpleStar.jpg')
Im3 = ('OrangeDiamond.jpg')
Im4 = pyglet.image.load('YellowCircle.jpg')
Im5 = pyglet.image.load('RedHex.jpg')
def InitializeGrid(board):
    #Initialize Grid by reading in from file
    for i in range(8):
        for j in range(8):
        board[i][j] = choice(['A', 'B', 'C', 'D', 'E'])
def Initialize(board):
    #Initialize game
    #Initialize grid
    InitializeGrid(board)
    #Initialize score
    global score
    score = 0
    #Initialize turn number
    global turn
    turn = 1
    #Set up graphical info
def ContinueGame(current_score, goal_score = 100):
    #Return false if game should end, true if game is not over
    if (current_score >= goal_score):
        return False
    else:
        return True
```

```
def SwapPieces(board, move):
    #Swap objects in two positions
    temp = board[move[0]][move[1]]
    board[move[0]][move[1]] = board[move[2]][move[3]]
    board[move[2]][move[3]] = temp
def RemovePieces(board):
    #Remove 3-in-a-row and 3-in-a-column pieces
    #Create board to store remove-or-not
    remove = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0,
      0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0,
      0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
    #Go through rows
    for i in range(8):
        for j inpyglet.image.load range(6):
            if (board[i][j] == board[i][j+1]) and (board[i][j] ==
              board[i][j+2]):
                #three in a row are the same!
                remove[i][j] = 1;
                remove[i][j+1] = 1;
                remove[i][j+2] = 1;
    #Go through columns
    for j in range(8):
        for i in range(6):
            if (board[i][j] == board[i+1][j]) and (board[i][j] ==
              board[i+2][j]):
                #three in a row are the same!
                remove[i][j] = 1;
                remove[i+1][j] = 1;
                remove[i+2][j] = 1;
    #Eliminate those marked
    global score
    removed_any = False
```

```
    for i in range(8):
        for j in range(8):
            if remove[i][j] == 1:
                board[i][j] = 0
                score += 1
                removed_any = True
    return removed_any
def DropPieces(board):
    #Drop pieces to fill in blanks
    for j in range(8):
        #make list of pieces in the column
        listofpieces = []
        for i in range(8):
            if board[i][j] != 0:
                listofpieces.append(board[i][j])
        #copy that list into colulmn
        for i in range(len(listofpieces)):
            board[i][j] = listofpieces[i]
        #fill in remainder of column with 0s
        for i in range(len(listofpieces), 8):
            board[i][j] = 0
def FillBlanks(board):
    #Fill blanks with random pieces
    for i in range(8):
        for j in range(8):
            if (board[i][j] == 0):
                board[i][j] = choice(['A', 'B', 'C', 'D', 'E'])
def Update(board, move):
    #Update the board according to move
    SwapPieces(board, move)
    pieces_eliminated = True
    while pieces_eliminated:
        pieces_eliminated = RemovePieces(board)
        DropPieces(board)
        FillBlanks(board)
```

```
@window.event
def on_draw():
    window.clear()
    for i in range(7,-1,-1):
        #Draw each row
        y = 50+50*i
        for j in range(8):
            #draw each piece, first getting position
            x = 50*j
            if board[i][j] == 'A':
                Im1.blit(x,y)
            elif board[i][j] == 'B':
                Im2.blit(x,y)
            elif board[i][j] == 'C':
                Im3.blit(x,y)
            elif board[i][j] == 'D':
                Im4.blit(x,y)
            elif board[i][j] == 'E':
                Im5.blit(x,y)
    label = pyglet.text.Label('Turn: '+str(turn)+'   Score:
      '+str(score), font_name='Arial', font_size=18, x=20,
      y = 10)
    label.draw()
@window.event
def on_mouse_press(x, y, button, modifiers):
    #Get the starting cell
    global startx
    global starty
    startx = x
    starty = y
@window.event
def on_mouse_release(x, y, button, modifiers):
    #Get starting and ending cell and see if they are adjacent
    startcol = startx//50
    startrow = (starty-50)//50
    endcol = x//50
    endrow = (y-50)//50
```

```
    #Check whether ending is adjacent to starting and if so,
      make move.
    if ((startcol==endcol and startrow==endrow - 1)
      or (startcol==endcol and startrow==endrow+1) or
      (startrow==endrow and startcol==endcol-1) or
      (startrow==endrow and startcol==endcol+1)):
        Update(board,[startrow,startcol,endrow,endcol])
        global turn
        turn += 1
        #See if game is over
        if not ContinueGame(score):
            print("You won in", turn, "turns!")
            exit()
#State main variables
score = 100
turn = 100
goalscore = 100
board = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
#Initialize game
Initialize(board)
pyglet.app.run()
```

› The new code, using event-driven programming, is shorter than the code for the text-based game. Creating fancier interfaces doesn't have to be a huge amount of code; you can create a lot of flexible functionality easily if you have the right library—in this case, pyglet.

› Pyglet is designed to support game development and can do lots of other stuff you might want to explore. It can generate two- and three-dimensional plots, graphs, and charts. It also can play sounds and music, display images, and handle events. Pyglet lets you use a graphics library called OpenGL, which lets you make all kinds of complex three-dimensional graphics.

## [ TKINTER ]

- Tkinter, a module that is part of the Python standard library, is useful for creating GUIs with buttons, boxes, and sliders. TK is a cross-platform toolkit that has been ported to many programming languages. Tkinter provides a binding between Python and the overall TK toolkit. The fact that it's cross-platform means that it's available for and works on most platforms.

- Although TK itself is not part of Python, it's released with Python, and the Tkinter module is built on top of it, providing an interface between Python and TK. This lets Python programmers have relatively easy access to a powerful cross-platform GUI library.

- Tkinter relies on object-oriented programming, so some of the coding might look strange on first viewing. But there is more than one way to do event-based programming, and this example will give you a sense of how an object-oriented approach to event-based programming differs, yet is also fundamentally the same.

- The following is a small program to demonstrate some really basic TK commands. It'll create two buttons: "increase" and "decrease." When you hit a button, you see a value printed out. If "increase" is hit, we double the value. If "decrease" is hit, we halve the value.

```
import tkinter
class Application(tkinter.Frame):
    def __init__(self, master=None):
        tkinter.Frame.__init__(self, master)
        self.pack()
        self.increase_button = tkinter.Button(self)
        self.increase_button["text"] = "Increase"
        self.increase_button["command"] = self.increase_value
        self.increase_button.pack(side="right")
        self.increase_button = tkinter.Button(self)
        self.increase_button["text"] = "Decrease"
```

```
        self.increase_button["command"] = self.decrease_value
        self.increase_button.pack(side="left")
    def increase_value(self):
        global mainval
        mainval *= 2
        print (mainval)
    def decrease_value(self):
        global mainval
        mainval /= 2
        print (mainval)
mainval = 1.0
root = tkinter.Tk()
app = Application(master=root)
app.mainloop()
```

- Tkinter uses event-driven programming, just like pyglet. We start by importing the tkinter module.

- Classes are a way of grouping things together in object-oriented programming. We'll define a class that inherits from tkinter ".Frame." The section of code that is indented is what is going to describe our window and how it works. Inside of here we'll set up our buttons and the callbacks that go with each one. These things that appear in the window that a user can interact with and generate events are called **widgets**. Besides buttons, TK provides all kinds of widgets—text boxes, sliders, and so on.

- When this object is initialized, it sets up the shape of the window—that's the line "self.pack()." Tkinter will pack the widgets into the window for you with some pretty simple commands. You can create sub-windows and pack those together to design the layout the way you want.

- That routine to create the widgets creates two widgets in this case: the two buttons. Each is created by four lines of code. The first button is the "increase" button. The first line of code for that section just tells TK that we're creating a button, which we refer to by the local variable increase_ button. The button is an "object," and objects will contain data called **attributes** and functions called methods.

- The second line of code for this button says that the button should display the text “increase,” and it does this by setting the “text” attribute of increase_buttton.

- The third line registers our callback by setting the “command” attribute of increase_button. It says that when the button is pressed, we should call the “increase value” function.

- The fourth line says to place the button at the right of the window. It does this by calling the “pack()” method that’s part of the increase_button.

- The increase value function will take a value named “mainval” (a global variable, in this case), multiply it by two, and print the new value to the output window.

- A second button is also created. The format is the same, but this one is labeled “Decrease,” will call the “decrease_value” function, and is at the left of the screen. The decrease_value function is just like the increase_value one, but it divides by two instead of multiplying by two.

- The last part of the code, in the main part of the code, will start the event handler. Specifically, there are lines to set up TK. Note that the class we just created is going to be the one defining our window and then starting the event manager—that’s the final line in the code. If we run this code, we see a window come up with two buttons, doing just what we said.

## Reading

Gries, *Practical Programming*, chap. 16.

## Exercises

1. Using the pyglet library, write a program that draws a window and draws an image wherever a mouse is clicked.

2. Using Tkinter, create a window with a single button. Each time the button is pressed, some phrase—such as “Hello!”—should be printed several times, once more than the previous one. So, the first press should print “Hello!” once, the second press should print “Hello!” twice, etc.

15 TRANSCRIPT

# Event-Driven Programming

You've heard phrases like, "A picture is worth a thousand words," or "Actions speak louder than words." Well, the same thing can be true in programming and in computer interfaces. What we see in a graphical user interface, and what we do inside that graphical interface, can be more important than words. We're going to explore this visual, action-oriented style of programming: how to write a graphical user interface, also known as a GUI; and how to use a style of programming that responds to mouse clicks and other events within the graphical interface, a style known as event-driven programming. To do all of this, we're going to make use of a package called pyglet.

Pyglet is a Python package created to help support development of games and other audio-visual environments. It provides functions that let you create windows, display images and graphics, play videos and music, get input from the mouse, et cetera. There are a few other game development packages that people also use—pygame is one that's well known—and there are other modules that do individual things that pyglet does, but pyglet packages these functions together nicely and it's easy to install. Just go to the pyglet site on bitbucket.org, and you can always find pyglet through a web search.

If you're using pip to install modules such as pyglet, you should be able to install pretty easily. Remember that you can go to the command line, then to the directory where Python is installed, and type "python –m pip install pyglet"—that's P-Y-G-L-E-T—and it should install for you. Once you have pyglet downloaded, be sure to run the command "import pyglet." Only if you don't actually have pyglet installed will you see a response, so any error message probably means that pyglet hasn't yet installed properly.

Now, one form of graphics familiar to everyone on a computer these days, in practice if not by name, is the graphical user interface—that's

G-U-I or GUI. And the way a GUI works is entirely based on event-driven programming. To understand the contrast, let's think about how we've been programming up to this point.

We've been writing commands, and we expect to start at the first command and follow the commands in sequence, one after the other. Things like conditionals, or loops, or function calls might make us jump to a different line of code, but we're basically still going along in a definite sequence where we always know which line of code we'll execute next, so we can refer to this as a sequential programming.

Event-driven programming is different. Instead of a program deciding when to ask a user for input, now events outside of the program determine what the program does next. An event is anything that happens where we want the program to respond. In a GUI, an event might not only be pressing a key on the keyboard, but also clicking a button on the screen, or entering data into a box on the screen, or moving the mouse. For each of those events, the computer program needs to respond; it needs to do something—maybe just update a variable, or maybe print to the screen. Robots often use event-driven programming, which allows them to respond to events in their environment. For a robot, events might be data that comes in from a sensor—for instance, the robot detecting it's about to hit a wall. When the sensor gets this information, it needs to send it to the program to respond.

The same sort of monitoring is always underway in the graphical user interface of a computer. In any kind of event-driven programming, whenever the program runs, there's an event monitor that runs continuously in the program. There are lots of other terms for the event monitor—a main loop, or an idle function, or a control function. Regardless of which term is used, the job of the event monitor is to take in events and make sure that the appropriate function gets called in response to the event.

There's more than one way to monitor events. Sometimes the event monitor uses what are called interrupts. Basically, it just sits there happily until something interrupts it with an event. Other times, the

event monitor actively polls the various devices—that is, it checks to see, “Is there a keyboard event? Is there a mouse event?” et cetera. But you don’t have to worry about whether it’s polling or interrupts that are monitoring events in your own programs—just know that the event monitor is indeed going to be getting the events.

Now, when the event monitor gets an event, it needs to do something to handle it. In a GUI, if someone clicks the mouse, the event monitor should call whatever function has been designated to handle mouse clicks. These functions that get called are known as event handlers. The event handler’s main job, its whole purpose for being there, is to take in events and then call what is known as a callback function corresponding to that event. The callback is just a function to execute in response to an event, at which time we go back to the event handler to get the next event. We sometimes say the event handler is invoking the callback function. It makes it sound almost magical.

To write an event-driven program, there are a few stages. The callback functions have to be defined on their own. These are defined just like all the other functions we’ve seen. The only difference is that different callback functions have to be ready to handle the appropriate type of parameter for the type they are. So, a callback that handles a key being pressed on the keyboard needs to be able to take in the key that was pressed as a parameter.

For the main body of the program, there’s usually some initialization work that’s done, just like any sequential program. As part of this initialization, callbacks need to be registered. Registering a callback function is how we say what function will be called for each event that we want to respond to. So, we have to write functions for each possible event. The final step of the sequential part of the program is to start up the event monitor. Once the event monitor is started, it keeps running indefinitely. As events occur, it just keeps calling the different callback functions. The real actions of the program happen in those callback functions.

Now, there’s not a single, universal way that event-driven frameworks work. The way the callbacks are registered, the parameters they

need to take in, and the way the event handler is started will all vary depending on what event-driven framework you're using. In Python, the pyglet framework is great for event-driven programming. But, realize that other frameworks will be structured somewhat differently.

So, let's look at a basic pyglet program. After we download pyglet, we import pyglet using the import command. Then, we can set up a window. This window command is: "pyglet.window.Window," and that lets us pass in parameters that define the window. In this case, I set a width to 400 and a height to 300, and a caption of "TestWindow." Finally, I have the command "pyglet.app.run." This last command is there to start the event handler. Now, when we run this, what do you think we'll see?

Well, we get a window, of size 400 pixels by 300 pixels, and the name on the window is "TestWindow." We don't really have anything else at this point, and we didn't define any callback functions, so our program doesn't actually do much of anything, but you can see that we set up a window just like that.

Let's print some text in this window. In pyglet, text is printed in what's known as a label. We'll create a label, in this case, one that just says "Howdy!" We can set some details about the label: the font style—in this case, that's Times New Roman; the size of the font, which is 18 in this case; and the place we want the font to appear—in this case, we're at $x = 50$ pixels and $y = 150$ pixels. That'll be just a little bit over in the horizontal direction and up about halfway in the vertical direction.

Now, in pyglet, $x$ and $y$ values are given in terms of the number of pixels from the lower left corner, with $x$ in the horizontal direction and $y$ in the vertical. But, that's not universally true in graphics. In fact, it's fairly common in other frameworks to start from the upper left-hand corner and have $y$ be the distance down the window. That's mainly due to a historical artifact of how graphics used to be created, but the idea has persisted. Also, sometimes $x$ and $y$ are not specified in pixels but rather in values relative to the current width and height. For example, there might be $x = 0$ on the left-hand side and $x = 1$ on the right-hand side, so you specify positions in between as a floating-point number between zero and one.

Now, after we set up a window, we need to register some callback functions. The first callback we'll need is for whenever we open or refresh, a window and the code looks kind of weird. The way we register a callback is with this: "@window event line." That says that we're about to define how to respond to a particular window event. The next line defines the "on_draw" function, and that means that this is the function that's going to be called when a draw event occurs in the window. Now, that might seem weird. What is a draw event? And does a user create a draw event?

Well, no, a user doesn't create a draw event, at least not directly. Remember that event-driven programming is a general idea, not one specific only to user input. In this case, the event that we're registering a callback for is whenever it's time to draw the window. As you might guess, that'll happen automatically when we first create a window, and it might happen at other times, too, like if we refresh a window, or if we have some other callback function that says, "We need to draw the window again."

So, let's look at the commands in the callback function itself. We have two of them. First, we have something that clears the window: "window.clear." We clear the window each time so that we have a nice blank canvas to work off of. After that, we have another command that draws the label: "label.draw." Notice that we took the variables we defined before—window and label—and we're calling a function by putting a dot after them and then the function name. You've seen this before, for example when we close a file. It's also an example of how we call functions in objects, which we'll discuss in later lectures.

So, what this code should do is create our window, and then clear it and draw the label inside of it. And indeed, that's exactly what it does. You might want to try messing around with this a little bit yourself. Try changing the font size and the position of the label. Or try changing the font itself and the words printed. Or, here's a challenge: try printing a second label to the window, maybe putting "Howdy" in the top left, and "World" in the lower right. Can you do this? Well, here's one way you might have done that. We'll create two labels, label1 and label2, each

with different text at a different position. Then, in our draw function, we'll draw both labels.

Let's see now how we might handle a different event, say a keyboard press. This is not a string that a person types in, it's the physical key being pressed down. The event is triggered when a key is pressed down, not because it's typed in. Believe it or not, you can even get another event when the key is lifted back up again.

So, we need a callback to handle keyboard presses. The pyglet function we need is known as "on_key_press." In the code here, we define the function to change the label that's shown. Before the key press, the label's text will be, "Nothing pressed so far." When there is a key press, the event handler will call the "on_key_press" callback. It'll reset the label's text to be, "You pressed a key." Now, if we run the code, we see that the window with the text, "Nothing pressed so far," comes up. As soon as we press a key, we get the new message, "You pressed a key."

Now, the parameters are of two types. The first parameter, symbol, gives the key that was pressed. So, the symbol can be all sorts of things, like a letter, or an arrow key, or the enter key, or the delete key. The second parameter is the modifier, but we're going to ignore modifiers here. Just so you know, a modifier is a key held down in combination with another key—the shift key, or control key, or function key are all examples of this. You can even have more than one modifier active when you press a symbol key.

So, let's focus on symbol keys and say that you want to print out a key that was pressed down. As I said, this is not a string, so we can't check to see if the symbol is equal to A or something. Instead, we have to compare it to some special value that indicates the key. We can put in a check to see if the symbol variable in the callback function is equal to the special value designating that the A key was pressed. That value, which is "pyglet.window.key.A," is kind of long to keep typing. So, I'm going to go ahead and import "pyglet.window.key" on its own, to make it easier to refer to. I've changed the text to be printed to show the key pressed, and I output "unknown" otherwise.

Now, we could start checking for other key presses also, like the return key—key.RETURN—or the left arrow key—key.LEFT. See if you can modify the code to note when the return key or left arrow keys are hit. Again, the code for return is key.RETURN, with return all capitalized, and the code for left arrow is key.LEFT, with left all capitalized. Here's the code. We just add a couple of "elif" statements onto the "if" statement. If we run this code, we can see that the text changes as I hit different keys. It will tell me if I hit the A key, the left arrow, the return key, or some other key it doesn't know about.

Let's see some of the other things pyglet can do. Besides keyboard commands, it can get input from the mouse. It's really similar to what we saw before with the keyboard. One callback function is "on_mouse_press," and that'll happen whenever a mouse button is clicked down. The parameters passed in are the position of the mouse, *x* and *y*, when the button was pressed down; a button value that indicates which mouse button was pressed down; and then modifiers that indicate if there's a key being held at the same time, just like for the keyboard. So, the code you see here will indicate where a mouse button was pressed down, and write that text to the screen.

There's also a simple mouse motion command, "on_mouse_motion," that gives the current *x* and *y* position, along with the change in position that's seen in the *x* and *y* direction from the last update. If we change the callback we saw earlier to "on_mouse_motion," we can print out the current position of the mouse as we move around. There's a mouse event for releasing a button—that's "on_mouse_release." And there's a mouse event for dragging the mouse—that's "on_mouse_drag." You can see here the format needed for each of those events.

Let's see one more thing that pyglet can do, and that's display images. To use images, there are two stages: reading in an image, and then displaying an image. To read an image in, we'll use the "pyglet.image.load" command. Pyglet can load several image formats, but we'll stick to .JPG, or jpeg files. If you look at the example here, I'm loading in a file called BlueTri.jpg. That's a file I created and put in the directory where my Python code is saved. It's a small 50 × 50 image of a blue triangle

on a white background, and I'm calling the loaded image file Im1—that's my shorthand for Image 1.

To display this, I use a command called "blit." Blit is short for "block image transfer," which refers to whenever two bitmaps are combined together. And that's what we're doing here—combining our jpg with the graphical window that it's going to appear inside. So, I write, "Im1.blit," and then pass in the parameters of where the lower left corner should be. If I run the code, sure enough, I get a window, and I see the image displayed right where I wanted it to be.

All right, let's put this functionality to use. Let's bring back the grid-based game that we developed in Lecture 13. This is the one where we would move some letters around, and if you had three in a row or in a column, they'd disappear. With the tools in pyglet, we can pretty easily make a graphical version of the game. Let's see how. I'll give you just a little warning—I'm going to move through this pretty quickly, so you might want to pause and think about each part that I show you instead of just rushing through it.

Here's the code from the game again. We're going to make some modifications to the way we draw the screen, and later to the way we get input. In the past, we had a "DoRound" function, and that would take care of one round: getting the user input, making the results of the move, and updating the turn. We now have to put this into an event-driven programming framework.

Drawing the board will no longer be something we call directly, but it needs to happen in response to events. We'll have an "on_draw" callback function in pyglet that should draw our current board. Likewise, we're going to get moves via the mouse. So, we don't call a "getmove" function any more, but rather we get our move by looking at where the mouse motion should be.

Let's look at the drawing function first. Instead of printing a list of characters to the screen, now we'll draw a set of shapes. So, I've created several .JPG files that can correspond to the various types of pieces. I'll

load each of these into a separate image. Each is a 50 × 50 image, so my 8 × 8 board will need to be at least 400 pixels × 400 pixels. I'll create a 400 × 450 window to make sure I have enough space to write some info at the bottom of the screen.

To draw the window itself, we'll write the "on_draw" function. We start by clearing the window. We then loop through the board and draw the corresponding image per piece. Since each image is 50 × 50, we offset in *x* and *y* by 50 for every row and column. And, when we draw each image, we pick one of the five images, depending on the piece that's there. At the bottom, we create a label in which we print out the turn and the score.

OK. Now, getting a move is going to be done through a combination of functions. We'll let the player click on one square, and then try to drag it to an adjacent cell and release, in order to make a move. To find the initial cell, we'll have to get the square in which they click the mouse, and so that will be the "on_mouse_press" callback. Basically, all we need to do at this point is store the starting point. We don't even know if it's a valid move yet until we figure out where the player releases the mouse.

Now, finding the final cell will happen when the mouse button is released, and that will be the "on_mouse_release" command. So, let's look at this code a little bit closer. First, we convert our window positions to row and column positions on our grid—this is just mathematics. Since every grid cell takes up 50 pixels, we divide by 50 to get the position in the grid. Notice that we're using the double slash instead of a single slash for division. You might remember that this is integer division, and it returns the integer part of the quotient, ignoring any remainder. Also, notice that since we left a gap at the bottom of the screen to write the score, we have to subtract 50 from the *y* value. Now, we do this for both the start position and the end position, and this gives us our row and column for each one. And we then check to see if we had a valid move. Basically, we need to either have the same row with the column different by exactly one, or the same column with the row different by one. If that's the case, then we have a valid move.

For a valid move, we call update, passing in the board and the move. And we also update the turn and check to see if we won, and thus can stop. Now, one thing that's different here is the way we specified the move. In the text-based version of the game, our move was a letter for the column and a number for the row, and then a direction for where to swap. This was only done to make things easier for the user to type in. But all that first part, where we get the original row and column and have to call "ConvertLetterToCol," and then go through all these "if-elif" statements—all of that's totally unnecessary now. Now that we have a graphical interface, there's no need for any of it. We already have the beginning and end positions. Instead, we can just pass in the row and column for the beginning and end. We do this as a single list, containing four numbers: the starting row and column, and the ending row and column.

The update function doesn't actually do anything with this move besides pass it on to the "SwapPieces" function. That's where we need to make a change. Look at the old "SwapPieces" function. Think about how to update it to make use of the way we specify the move now. Now remember, we already have the beginning and ending positions—they're just the four elements of the list that we pass in as move. So, we can delete that whole first part of the function, everything except the swap part itself. We then need to make a change to the swap part.

And, here's what that looks like. It's the same set of commands that we had before. We're going to swap by putting the first piece into a temporary variable, then moving the second piece into the first slot, then moving from the temporary variable into the second slot. The only difference from what we had in an earlier version is that we're using the indices passed in as the move parameter in order to reference the rows and columns. So, move-sub-zero and move-sub-1 are the original row and column, while move-sub-2 and move-sub-3 are the new row and column.

The only other change is that in the main part of the code, we need to end by starting the event handler. This goes in the main function. That's the "pyglet.app.run" command—the rest of the code is the same. So, if we run this, we can see all the elements, and we can use our mouse to choose which one to move where. For example, if we take this square,

and we move it over to this other one, we line up several pieces in a row, they disappear, and the board updates.

Now, compared to a professional game, this probably looks kind of clunky. We aren't focused on graphic design, and I'm not much of an artist, and there's much more that could be done to make it look pretty. We could add additional things like flashes when something's deleted. Each improvement to the look of the game is just more code.

Anyway, the key code for the game is what we already have right here. And what's really surprising is this—the new code, using event-driven programming, is actually shorter than the code for the text-based game. Remember, we were able to get rid of a lot of code to do things like convert letters to columns, and there were several routines no longer called that we can just delete. So, creating fancier interfaces doesn't have to be a huge amount of code—you can create a lot of flexible functionality easily if you have the right library. In this case, that was pyglet.

Pyglet is actually designed to support game development and can do a lot of other stuff that you might want to explore. It can generate 2-D and 3-D plots, graphs, and charts; it can play sounds and music; display images; and, of course, handle events. Pyglet lets you use a graphics library called OpenGL, and that lets you make all sorts of complex 3-D graphics. Now, 3-D graphics could easily be a whole course of its own. The point here is that you can build on pyglet, or find another module to handle events and do whatever you want to do.

Now, speaking of finding another module, let me mention a part of the Python standard library called Tkinter, which is useful for creating graphical user interfaces with buttons, boxes, and sliders. TK is a cross-platform toolkit that's been ported to many programming languages. What Tkinter does is provide a binding between Python and the overall TK toolkit. The fact that it's cross-platform means it's available for, and it works on most platforms—Windows, OS X, and Linux. And, although TK itself is not part of Python, it's released with Python, and the Tkinter module is built on top of it, providing an interface between Python and

TK. This lets Python programmers have relatively easy access to a powerful cross-platform GUI library.

Tkinter relies on object-oriented programming, so some of the coding might look a little strange on first viewing. But there's more than one way to do event-based programming, and this example will give you a sense of how an object-oriented approach to event-based programming differs, yet is still fundamentally the same.

Here's a small program to demonstrate some really basic TK commands. It'll create two buttons: increase and decrease. When you hit a button, you see a value printed out. If increase is hit, we double the value; if decrease is hit, we halve the value—simple. I'll highlight the various parts of this code, and if you're interested in finding out more about the details, you should be able to investigate more on your own. Again, remember that some of this will make a little more sense after we've talked about object-oriented programming.

Tkinter uses event-driven programming, just like pyglet. We start by importing the Tkinter module. Classes are a way of grouping things together in object-oriented programming. We'll define a class that inherits from "tkinter.Frame." This section of code, all the indented stuff, is what's going to describe our window and how it works. Inside of here, we'll set up our buttons and the callbacks that go with each one. Now, these things that appear in the window that a user can interact with and generate events with are called widgets. Besides buttons, TK provides all sorts of widgets—textboxes, sliders, and so on. When this object is initialized, it sets up the shape of the window—that's the line "self. pack." Tkinter will pack the widgets into the window for you with some pretty simple commands. You can create sub-windows and pack those together in order to design the layout the way you want.

That routine to create the widgets creates two widgets in this case—the two buttons. Each is created by four lines of code. The first button is the increase button. The first line of code for that section just tells TK that we're creating a button, which we refer to by the local variable increase_button. The button is an object, and objects will contain data

called attributes, and functions called methods, and this is what we'll discuss when we talk about object-oriented programming. The second line of code for this button says that the button should display the text "increase," and it does this by setting the text attribute of increase_button.

The third line registers our callback by setting the command attribute of increase_button. It says that when the button is pressed, we should call the "increase_value" function. Finally, the fourth line says to place the button at the right of the window. It does this by calling the pack method that's part of the increase_button. The "increase_value" function will just take a value named mainval, multiply it by two, and print the new value to the output window. Mainval is a global variable in this case.

A second button is also created. The format is the same, but this one will be labeled "decrease," and it will call the "decrease_value" function, and it's at the left of the screen. The "decrease_value" function is just like the "increase_value" one, but it divides by two instead of multiplying by two.

Finally, the last part of the code, in the main part of the code, will start the event handler. Specifically, there are lines to set up TK, note that the class we just created is going to be the one defining our window, and then starting the event manager—that's that final line of code. If we run this, we do indeed see a window come up with two buttons, doing just what we said.

Now, even though that code looked quite different from the event-driven programming in pyglet, you can see that the concepts are similar. We still have an event handler that will run until an event occurs. We have callback functions that are registered to the different events. In pyglet, the events were basic user input—pressing a button, moving a mouse, and so on. In Tkinter, the events are tied into the widgets—like buttons, sliders, and so on—so that the programmer sets things up, and then those events get called. Both modules also support graphical display, with pyglet focused more on basic graphics like images and 3-D graphics, and Tkinter focused more on creating interactive windows

with widgets. To gain greater familiarity with these approaches, I'd encourage you to play around with pyglet, Tkinter, or a similar package, and explore some ways to make programs that you've already written more graphical.

Event-driven programming is widespread, used in everything from robotic sensors, to network data communication, to graphical user interfaces, including many that appear on the web. Event-driven programming is usually what's responsible whenever you press buttons, or fill out a form with boxes where you enter data and then press a button, or move a slider, or hit enter on a text box. All of those will trigger some event in the program.

The same kind of graphics-based functionality is also present in everything from software applications on a computer to apps downloaded on a smartphone. Best of all, as we've seen, event-driven programming can make possible much more impressive graphical displays without necessarily requiring a lot more work.

In our next lecture, we're going to be using programming to create graphical displays of data, as well as explore how we can create simulations of data that may be difficult or impossible to obtain directly. Far more than most people realize, computer simulations of many kinds influence everything from public policy to our personal lives, so I'll look forward to showing you more then.

16

# Visualizing Data and Creating Simulations

One particularly useful aspect of computation is to simulate what might happen in the real world, test scenarios, and understand the range of options. Visualizing data is a key part of this. Taken together, the decisions made based on simulations have consequences measured in the trillions of dollars, are sometimes matters of life and death, and affect practically everyone on the planet. Computers are famous for handling data, but data visualization and data simulation are two areas that often go under-recognized. In this lecture, you will learn how to do both.

## [ DATA VISUALIZATIONS ]

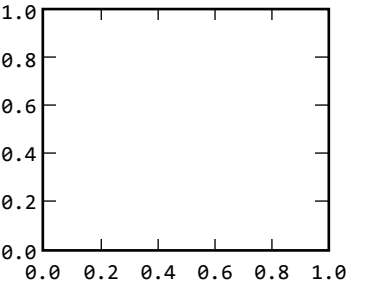
- One of the best packages to create visualizations of data is matplotlib. It has a very wide range of capabilities and is probably the most well-known and popular Python package for creating plots, graphs, and charts from data.

- The first step is to install matplotlib. One option is to use pip. If pip is installed, you should be able to go to your Python directory in the command line and type "python –m pip install matplotlib." If you aren't using pip, you can find out the details of how to install matplotlib at the matplotlib.org website. Installing matplotlib will require installing several other libraries, too.

- With matplotlib installed, let's start by making a basic display. We'll first import pyplot from matplotlib. Then, we'll use the pyplot.axes function, which basically says that we're going to have a data plot that uses axes. We could provide parameters to specify the appearance and range of these axes, but we don't provide any parameters—we can just use the defaults.

› Finally, once we've created a data plot, we'll need to show it, so we call show.

```
from matplotlib import pyplot
pyplot.axes()
pyplot.show()
```




› If we run this, we see that matplotlib has created a plot with axes in the range of 0 to 1 in both *x* and *y*. And it includes some graphical tools at the bottom that let you interact with the chart by zooming in, moving around, or saving it.

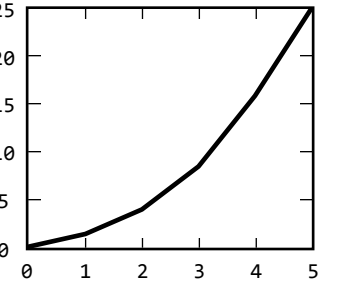
› We learn how to use graphical tools like this by looking at the documentation. In particular, the API is the application programming interface. The API documentation lists the various commands that are provided and the details of how each is used.

› For matplotlib, you can see the key plotting commands if you follow the link at the top of the page to "pyplot." That gives you a whole list of commands provided in pyplot, and if you click on each of them, it will give you a more detailed description of what the command does and what parameters it takes in.

› The plot command can take in few lists. The first one gives all the *x*-values. The second one gives all the *y*-values. In this case, we're using the *x*-values from 0 to 5, and then for the *y*-values, we're using the square of the *x*-values.

```
from matplotlib import pyplot
pyplot.plot([0,1,2,3,4,5], [0,1,4,9,16,25])
pyplot.axis([0,5,0,25])
pyplot.show()
```

- Also, notice that we're now passing a parameter to the axis command. The parameter is a list of four numbers, giving the minimum and maximum extents for the *x*-axis and the minimum and maximum for the *y*-axis. In this case, we say that the *x*-axis will go from 0 to 5 and the *y*-axis will go from 0 to 25. Running this gives the plot we'd expect.




- The following is a more compact version that makes the same plot. Notice that instead of manually making the lists, we made a list of *x*-values using the range command. Then, we built a list of *y*-values by going through the *x*-values and appending the square of that *x*-value onto the list.

```
from matplotlib.pyplot import plot, axis, show
xlist = range(0,6)
ylist = []
for i in xlist:
    ylist.append(i*i)
plot(xlist, ylist)
axis([0,5,0,25])
show()
```

- Also, notice that we're just importing functions from pyplot to make things simpler. Instead of having to write "pyplot" in front of each, we can write "plot," "axis," or "show" directly. And if we run this, we get the same results we had with the previous case.

- There are many ways to improve and change graphs. There are many options in the mathplotlib module—not only for how to do line plots, but also for numerous other types of charts and graphs. It's relatively easy to show many different graphical representations of data, and once you create the representation, it's also easy to incorporate graphical output into any larger program.

## [ SIMULATIONS ]

› Beyond visualizing the data we have, there's also data we would like to have but don't. This brings us to **simulations**. When we talk about simulations, we normally mix two ideas that are very related but distinct. The first idea is a **model**, which tells us what the laws, rules, or processes that we are trying to compute should follow. The actual simulation takes the model and some set of conditions and uses it to determine how the situation develops, usually over time.

› The model is the most important thing about the whole simulation process. If the model is incorrect, it doesn't matter how good the computer is at performing the simulation—it won't get the correct answer. It's also very important to have the correct initial conditions. Sometimes even tiny errors in initial conditions can have large effects later.

› For a typical simulation, we're given a model of behavior and some initial condition. We refer to the overall values we want to simulate as the **state** of the system. The initial conditions will be a starting state ($S_0$) at a starting time ($t_0$). Then, we are given some time in the future that we want to simulate to ($T$).

› We're also given what is called a **time step** ($h$). The idea is that we're going to take steps forward in time by that amount. That will let us determine a new state at that new time. We'll call this sequence of states $S_i$ and the sequence of times $t_i$. This will continue until we've reached the total time we want to simulate, $T$.

› Let's say that we want to see how an account accumulates interest over time. Suppose that we use $1000 to buy a 10-year certificate of deposit that earns 3% per year. We want to see how that grows over time.

- In this context, our model is the increase in interest rate—basically, that our value increases by 3% per year. Our initial state ($S_0$) is the initial balance, or $1000. And we'll call this year 0—the starting point of the simulation. The simulation will go forward in steps of 1 year, and we'll simulate up to 10 years. So, the simulation loop itself will repeat while $t_i$ is less than 10 and each time will calculate the balance 1 year later.

- The code is as follows. We'll use the variable time to keep track of time ($t$) and the variable balance to keep track of our state, and we'll initialize each of these to our starting conditions—time 0 and $1000 balance. Each of these is going to be stored in a list—"timelist" or "balancelist"—so that we can keep track of growth over time.

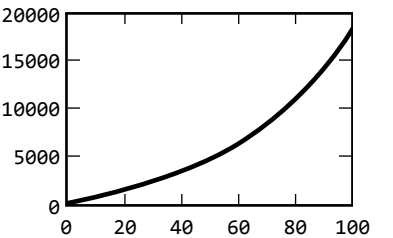
```
#Set initial conditions
time = 0
balance = 1000
#Set list to store data
timelist=[time]
balancelist=[balance]
while (time < 10):
    #Increase balance and time
    balance += balance*0.03
    time += 1
    #Store time and balance in lists
    timelist.append(time)
    balancelist.append(balance)
#Output the simulation results
for i in range(len(timelist)):
    print("Year:", timelist[i], "  Balance:", balancelist[i])
```

- We then have our simulation loop. In the loop, we increase the balance by 3% and the time by 1. We store these in the time and balance lists. And this continues until we've done this for 10 years. At the end, we print everything out.

› If we wanted a graph of the data, it's a simple matter of importing pyplot commands from matplotlib and calling plot and show.

```
from matplotlib.pyplot import plot, show
#Set initial conditions
time = 0
balance = 1000
#Set list to store data
timelist=[time]
balancelist=[balance]
while (time < 10):
    #Increase balance and time
    balance += balance*0.03
    time += 1
    #Store time and balance in lists
    timelist.append(time)
    balancelist.append(balance)
#Output the simulation results
for i in range(len(timelist)):
    print("Year:", timelist[i], "  Balance:", balancelist[i])
plot(timelist, balancelist)
show()
```

› We get an output showing an exponential growth pattern over the years.



## [ MONTE CARLO SIMULATIONS ]

› The model can be any kind of process. In many scientific simulations, the model is a set of differential equations. But let's focus on a particular class of simulations called Monte Carlo simulations. Monte Carlo simulations are based on the idea of simulating lots of random events, but doing it enough times that the overall outcome will be more understandable. A Monte Carlo approach is used in all kinds of simulations, from fluid physics to finance, especially situations in which there is a lot of uncertainty to include.

› Let's look at a simulation of finances, in which you take some set of investments and see how likely they are to meet your retirement goals. Let's start with the previous example, where we examined the growth in a certificate of deposit over a period of time. We can modify the code from that simulation as follows to handle more general investments.

```
from matplotlib.pyplot import plot, show
#Set initial conditions
time = 0
balance = 1000
#Set list to store data
timelist=[time]
balancelist=[balance]
while (time < 10):
    #Increase balance and time
    balance += balance*0.03
    time += 1
    #Store time and balance in lists
    timelist.append(time)
    balancelist.append(balance)
#Output the simulation results
for i in range(len(timelist)):
    print("Year:", timelist[i], "  Balance:", balancelist[i])
plot(timelist, balancelist)
show()
```

› We're going to start by changing the way we compute the change per year. We're going to follow the principle of abstraction to make a separate function that will calculate how the investment will increase (or decrease) in any particular year.

› So, we'll create a function, "ChangeInBalance," that takes in the current balance as a parameter and returns how much it changes 1 year later. In the earlier case, we had a 3% interest rate, so the change in balance is 3% of the original balance. In our simulation loop, each iteration of the loop increases the balance by ChangeInBalance.

```
from matplotlib.pyplot import plot, show
def ChangeInBalance(initial_balance):
    return initial_balance*0.03
#Set initial conditions
time = 0
balance = 1000
#Set list to store data
timelist=[time]
balancelist=[balance]
while (time < 10):
    #Increase balance and time
    balance += ChangeInBalance(balance)
    time += 1
    #Store time and balance in lists
    timelist.append(time)
    balancelist.append(balance)
#Output the simulation results
for i in range(len(timelist)):
    print("Year:", timelist[i], "  Balance:", balancelist[i])
plot(timelist, balancelist)
show()
```

› Unless we have some "guaranteed" investment, such as a certificate of deposit, the amount that the investment increases or decreases changes with time. Interest rates go up and down, and for investments that are traded, the fluctuations can be quite significant.

- Suppose that we have an investment that we know can fluctuate, but the return will never be negative. Instead of increasing our balance by 3% each year, we might change the balance by a random percentage. A simple way to do this might be to imagine that we can select a maximum level, and a minimum level, and every rate in between is equally likely.

- We could modify our code to pick a random rate of return each year for that investment. We import the random module. We then use the uniform command to pick a random rate in between some maximum and minimum. Let's say that our rate of return will be between 0% and 6%.

```
import random
from matplotlib.pyplot import plot, show
def ChangeInBalance(initial_balance):
    rate = random.uniform(0.0, 0.06)
    return initial_balance*rate
#Set initial conditions
time = 0
balance = 1000
#Set list to store data
timelist=[time]
balancelist=[balance]
while (time < 10):
    #Increase balance and time
    balance += ChangeInBalance(balance)
    time += 1
    #Store time and balance in lists
    timelist.append(time)
    balancelist.append(balance)
#Output the simulation results
for i in range(len(timelist)):
    print("Year:", timelist[i], "  Balance:", balancelist[i])
plot(timelist, balancelist)
show()
```

- We can run this code and see what the results would be. Every time we run the code, we get a different result. Over the 10 years, the results tend to come out pretty similarly, because the variations will tend to average out.

- If we want to get an even better sense of what the overall performance is likely to be, we can run this code multiple times. To do this, we need to essentially wrap up the simulation into another loop that will run the simulation over and over. And we need to store the results from each time we do that loop.

- We're going to get rid of the lists of the balances year by year and only store the final balances in a list. We'll also generalize things so that the number of years in the simulation and the total number of simulations are single variables that are easy to change.

- We still start with our function that computes the change in balance. We'll then have a loop for however many simulations we need. In each of them, we'll start at time 0, and with a balance of $1000, and simulate for a few years, just like before. We'll store the final balance into the final balances array. After this loop, we can print out all the final balances we found.

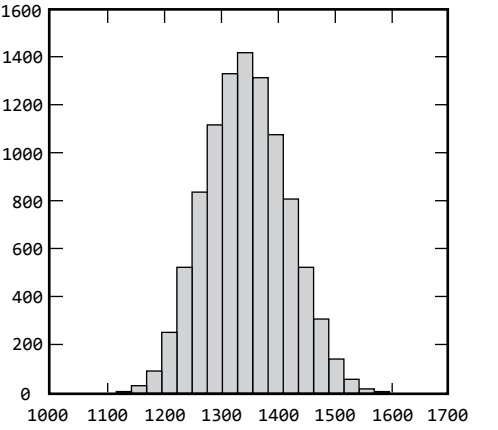
```
import random
def ChangeInBalance(initial_balance):
    rate = random.uniform(0.0, 0.06)
    return initial_balance*rate
number_years = 10
number_sims = 100
final_balances = []
for i in range(number_sims):
    #Set initial conditions
    time = 0
    balance = 1000
    while (time < number_years):
        #Increase balance and time
        balance += ChangeInBalance(balance)
        time += 1
```

```
    final_balances.append(balance)
#Output the simulation results
for i in range(number_sims):
    print("Final Balance:", final_balances[i])
```

› Given the results of all those runs, we can replace a simple printout of the values with a histogram to plot the results. The "hist" command in matplotlib will take in a list of results—the final balances, in this case—and plot a histogram. We set the number of bins in this case equal to 20, and we'll run 10,000 experiments.

```
import random
from matplotlib.pyplot import hist, show
def ChangeInBalance(initial_balance):
    rate = random.uniform(0.0, 0.06)
    return initial_balance*rate
number_years = 10
number_sims = 10000
final_balances = []
for i in range(number_sims):
    #Set initial conditions
    time = 0
    balance = 1000
    while (time < number_years):
        #Increase balance and time
        balance += ChangeInBalance(balance)
        time += 1
    final_balances.append(balance)
#Output the simulation results
hist(final_balances, bins=20)
show()
```

- When we run this, we get a wide distribution of results, from cases where we earned small amounts of interest to those where we earned a lot.




- To understand overall performance, we can compute some basic statistics on the final balances. To help with this, we can use the "statistics" module that's part of the Python standard library. It has functions such as "mean" and "stdev" to calculate the overall mean and standard deviation of a list. So, we can modify our code to import the statistics module, and then at the end of the program, we print out the average and standard deviation from all of our runs.

## Readings

Matthes, *Python Crash Course*, chap. 15.

Zelle, *Python Programming*, chap. 9.

## Exercise

Imagine that you are rolling 3 dice and are interested in the sum of those dice. Use a Monte Carlo simulation to simulate 10,000 rolls of 3 dice. Use matplotlib to plot a histogram of the results.

16 TRANSCRIPT

# Visualizing Data and Creating Simulations

Computers are famous for handling data, but data visualization and data simulation are two areas that often go underrecognized. Let's learn how to do both of these. One of the best packages to create visualizations of data is matplotlib. It has a very wide range of capabilities and is probably the most well-known and popular Python package for creating plots, graphs, and charts from data.

You're going to want to install matplotlib first. One option is to use pip. If pip is installed, you should be able to go to your python directory in the command line and type "python –m pip install matplotlib." If you aren't using pip, you can find out the details of how to install matplotlib at the matplotlib.org website.

OK, with matplotlib installed, let's start by making a really basic display. You can think of this like the "Hello, World" version of matplotlib. We'll first import pyplot from matplotlib. We'll then use the pyplot.axes function, which basically says we're going to have a data plot that uses axes. Now we could provide parameters to specify the appearance and range of these axes. But if we don't provide any parameters, we can just use the defaults. And finally, once we've created a data plot, we'll need to show it, so we call show.

If we run this, we see that matplotlib has created a plot for us with axes in the range of 0 to 1 in both *x* and *y*. And, it includes some graphical tools at the bottom that let you interact with the chart by zooming in, or moving around, or saving it.

OK, let's see how to display some data using the plot command. The plot command can take in a couple of lists. The first one gives all the *x* values. The second one gives all the *y* values. In this case, I'm using the *x* values from 0–5, and then for the *y* values, I'm using the square of the *x* values.

Also, notice that we're now passing a parameter to the axis command. The parameter is a list of 4 numbers, giving the minimum and maximum extents for the *x*-axis, and the minimum and maximum for the *y*-axis. In this case, we say the *x*-axis will go from 0 to 5 and the *y*-axis from 0 to 25. Running this code gives the plot just like we'd expect.

Here's a more compact version that makes the same plot. Notice that instead of manually making the lists, I made a list of *x* values using the range command. Then, I built a list of *y* values by going through the *x* values and appending the square of that *x* value onto the list. Also, notice that we're just importing functions from pyplot, to make things simpler. Instead of having to write "pyplot" in front of each, I can write "plot," "axis" or "show" directly. And, if we run this, we get the same results we saw in the previous case.

Now, there are a lot of ways to improve and change graphs. Say I wanted the plot to be a red line, and each of the points to be marked with a blue cross. And, imagine that I wanted a label for the graph, like "squares." So, in code, we would set some of the parameter values we were letting go to the default previously. We can set "squares" for the label, note that we want a marker that's a plus sign and that the marker should be in blue, and make the color of the graph itself red. For each of these, we list the parameter and the value that we want it to take. And, there's a new command, legend, that we have to import from pyplot. To display the name of the plot, we import the legend command from matplotlib. pyplot. When we run this, we see it that indeed creates a red line with blue crosses marking each data point.

We can also print two plots on the same graph. In this case, we'll create a set of cubes. We'll plot that one in green with green circles marking the data points. All it takes is to create another list of *y* values and then to call plot a second time, with the parameters for the second plot we want. And, we see a plot of this data, with the cubes and squares drawn in the different colors just like we specified.

We can even have different *y*-axes for different plots. This gets a lot more complicated to show, but here we have 3 different vertical axes,

showing three different data sets being plotted. When we see this plot, we see that we can color code based on the data set; we can show multiple graphs, et cetera.

Let's say that we wanted to graph the amount of a mortgage over time. Here's what the code might look like. We could first get the amount of the mortgage, the interest rate, and the monthly payment from the user. Then, we can generate values for each month. We'll keep this up until we've reduced the mortgage to zero or gone through 30 years. Each month we'll do several things. First, we'll update a month array with a new month. We'll also add on to three other arrays. First, we compute what the amount of interest generated in a month is. We then figure out how much of the payment went to interest vs. principal. Each of these is recorded in a list, as is the amount of the mortgage remaining.

We can easily print out the data using our plot commands. We use a red line to show the remaining mortgage, a blue one for the total principal paid, and a green one for the total interest paid. If we run this, and we enter some basic data, say a mortgage of $100,000, a 5% interest rate, and a paycheck of just $1000, we indeed see a plot showing how these values vary over time.

There are many other options in the matplotlib module. Not only for how to do line plots like this, but also for numerous other types of charts and graphs. The point is that it's relatively easy to show many different graphical representations of data. And once you create the representation, it's also easy to incorporate graphical output into any larger program.

Beyond visualizing the data we have, there's also data we would like to have, but don't. This brings us to simulations. A desire for simulations motivated development of the earliest computers. Harvard's Mark I computer did simulations for the U.S. Navy and the Manhattan Project during World War II. After the war, ENIAC, the first fully electronic computer, simulated ballistic trajectories for the U.S. Army starting in 1946.

Now, when we talk about simulations, we normally mix two ideas, that are very related but distinct. The first idea is a model. A model tells us

what the laws, rules, or process that we are trying to compute. The actual simulation takes the model and some set of conditions and uses it to determine how the situation develops, usually over time. So, if we want to simulate a comet in our solar system that might be heading toward Earth, our model would describe gravity of the sun and planets. The initial condition would give the starting position and velocity of the comet. Then the simulation would be used to predict where the comet would be in the future.

The thing to realize is that the model is the most important thing about the whole simulation process. If the model is incorrect, it doesn't matter how good the computer is at performing the simulation—it won't get the right answer. If our model of gravity is wrong, then our prediction about the path of the comet will be wrong. If our model of how a hurricane moves is wrong, then our predictions of hurricane paths won't be accurate. It's also very important to have the correct initial conditions. If we try to predict the path of a comet, but our starting position is wrong, we could end up computing a completely wrong path. Sometimes even tiny errors in initial conditions can have large effects later on.

You've probably heard the phrase "garbage in—garbage out" before. If your model or initial conditions are garbage, the output will be, also. It doesn't matter how great a programmer is, how sophisticated a simulation is, or how powerful a computer is—if the model or initial conditions are wrong, the result is unreliable.

For a typical simulation, we're given a model of behavior and some initial condition. We refer to the overall values that we're wanting to simulate as the state of the system. The initial conditions will be a starting state $S_0$, at a starting time $T_0$. Then, we are given some time in the future that we want to simulate to, T. And, we're given what's called a timestep, which is usually stated as *h*. That's in honor of 19th-century English physicist Oliver Heaviside, who pioneered a widely used step function in calculus. The idea is that we're going to take steps forward in time by that amount, *h*. That will let us determine a new state at that new time. We'll call this sequence of states $S_i$ and the sequence of times $t_i$. This will continue until we've reached the total time we want to simulate, T.

Let's say that we want to see how an account accumulates interest over time. Suppose we use $1000 to buy a 10-year certificate of deposit that earns 3% per year. We want to see how that grows over time.

Remember what should be needed for our simulation. In this context, our model is the increase in interest rate: basically that our value increases by 3% per year. Our initial state $S_0$ is the initial balance, or $1000. And, we'll call this year 0—the starting point of the simulation. The simulation will go forward in steps of 1 year, and we'll simulate up to 10 years. So, the simulation loop itself will repeat while $t_1$ is less than 10, and each time will calculate the balance one year later.

Let's see what the code for that will look like. We'll use the variable time to keep track of time t, and the variable balance to keep track of our state, and we'll initialize each of these to our starting conditions—time 0 and $1000 balance. Each of these is going to be stored in a list—timelist or balancelist so that we can keep track of growth over time. We then have our simulation loop. In the loop, we increase the balance by 3%, and the time by 1. We store these in the time and balance lists. And, this continues until we've gone for 10 years. At the end, we print everything out.

If we wanted a graph of the data, it's a simple matter of importing pyplot commands from matplotlib, and calling plot and show. And, we get an output showing an exponential growth pattern over the years.

The model can be any sort of process. In a lot of scientific simulations, the model is a set of differential equations. But for the rest of this lecture, I want to focus on a particular class of simulations called Monte Carlo simulations. Now, when I say Monte Carlo you might think of the famous casino in Monaco. If you think of casinos and gambling, you should think about probabilities, and how casinos make money. Although the result of any one roll of the dice or spin of the wheel might be random, they know that over time, there will be a certain percentage of the time that they win.

Monte Carlo simulations are based on the idea of simulating lots of random events, but doing it enough times that the overall outcome will be more understandable. A Monte Carlo approach is used in all sorts of

simulations, from fluid physics to finance, especially situations in which there's a lot of uncertainty to include. So, in the rest of this lecture, we're going to look at how we'd make a larger program that incorporates uncertainty and allows us to understand the range of possibilities that can occur.

We're going to be looking at a simulation of finances. You might have seen something like this before—where you take some set of investments, and you see how likely they are to meet your retirement goals. Let's start with our previous example, where we examined the growth in a certificate of deposit over a period of time. We can modify the code from that simulation to handle more general investments.

We're going to start by changing the way we compute the change per year. We're going to follow the principle of abstraction to make a separate function that will calculate how the investment will increase, or decrease in any particular year. So, we'll create a function, ChangeInBalance, that takes in the current balance as a parameter, and returns how much it changes one year later. In the earlier case, we had a 3% interest rate, so the change in balance is 3% of the original balance. In our simulation loop, each iteration of the loop increases the balance by ChangeInBalance.

Unless we have some guaranteed investment like a certificate of deposit, the amount that the investment increases or decreases will change itself over time. Interest rates go up and down, and for investments that are traded, like stocks and mutual funds, the fluctuations can be really significant. Suppose we have an investment that we know can fluctuate, but the return will never be negative. Instead of increasing our balance by 3% each year, we might change the balance by a random percentage. A simple way to do this might be to imagine that we can select a maximum level, and a minimum level and every rate in between is equally likely.

We could modify our code to pick a random rate of return each year for that investment. We import the random module. And, then we use the uniform command to pick a random rate in between some maximum

and minimum. Let's say our rate of return will be between 0% and 6%. We can run this code, and see what the results would be. Every time we run the code, we get a different result. Now, over the 10 years, it tends to come out pretty similar, since the variations will tend to average out. But, each time we run, we get a slightly different behavior.

Now, if we want to get an even better sense of what the overall performance is likely to be, we can run this code multiple times. To do this, we need to essentially wrap up the simulation inside of another loop that will run the simulation over and over. And, we need to store the results from each time we do that loop.

Let's see what that'll look like. We're going to get rid of the lists of the balances year-by-year and only store the final balances in a list. We'll also generalize things so that the number of years in the simulation and the total number of simulations are single variables that are easy to change. We still start with our function that computes the change in balance. We'll then have a loop for however many simulations we need. In each of them, we'll start at time 0, and with a balance of $1000, and we simulate for a few years, just like before. We'll store the final balance into the final balances array. After this loop, we can print out all the final balances we found.

Given the results of all those runs, we can replace a simple printout of the values with a nice histogram to plot the results. We can use the hist command in matplotlib. That will take in a list of results, which in this case is the final balances, and plot a histogram. We set the number of bins in this case to equal 20, and we'll run 10,000 experiments. You can see that when we run this, we get a wide distribution of results, from cases where we earned small amounts of interest to those where we earned a whole lot.

Furthermore, it's a relatively simple task to make our model more realistic. Say that instead of a random percentage increase between 0 and 6, you instead wanted to pick a random percentage from a normal bell-curve distribution, with an average of 3% and a standard deviation of 2%. How might you do this?

Well, you'd want to change your random function. In this case, we could use the "gauss" or the "normalvariate" function in the random library to create this distribution. In this case, I used "gauss," which is short for "Gaussian"—the normal bell curve distribution that we're familiar with. Because we've separated out the ChangeInBalance routine, it's easy to change to a different model where we calculate the change by some other mechanism.

Say, for example, that we wanted to simulate a stock fund. We could take a historical list of yearly increases or decreases in the S&P 500, and just select one of those. We would need to load in a list of yearly returns, and then randomly select one of the elements of the list, and use that percentage increase or decrease. In this case, I've made a list of 70 years worth of data, and I've hard coded it, but we could read data from a file. Then, we use the "choice" function from the random module to pick one of those 70 returns. If we run this code, we'll see a pretty wide range of outcomes, from ones where we actually lose a little money to ones where we make more than 10 times what we started with.

Let's say that we want to understand that overall performance. We can compute some basic statistics on the final balances. To help with this, we can use the "statistics" module that's part of the Python standard library. It has functions such as "mean" and "stdev," or standard deviation to calculate the overall mean and standard deviation of a list. The larger the standard deviation, the more volatile our investment is. So, we can modify our code to import the statistics module, and then at the end of the program, we print out the average and the standard deviation from all of our runs. If we run this code, we see we get a pretty high balance, and we also a pretty big standard deviation.

Different types of accounts will tend to have different performance characteristics. Stocks will tend to fluctuate a lot in price, but tend to do best on average. Bonds don't do as well as stocks on average, but the drops, as well as gains, will tend to be smaller. Money in cash, like savings accounts, will tend to fluctuate the least, but it'll have the worst average performance. If we have a different set of data, say for bond yields, we can calculate how a bond investment would perform.

So, here I've pulled out a set of bond rates, which I've again hard coded in to the program. I'll pull out a set of bond rates, and see how we might do if we had an account that earned that rate each year, instead of the stock market rate. In fact, we'll simulate both accounts. First, we have a list of historical bond yields. Next, we create a second function to compute the increase in balance for bonds, and we rename the one for stocks so that they're clearly different. So, ChangeInBalanceStocks will compute the change in balance assuming a stock account, and ChangeInBalanceBonds will compute the change in balance assuming a bond account. We'll basically keep track of everything else in pairs. We keep "balance_stocks" for stocks and "balance_bonds" for bonds, and we keep separate lists of final balances for each, as well. We update each one using the appropriate ChangeInBalance function. At the end, we will compute the mean and standard deviation for each of them separately. And, we can show a histogram of these two options, together in one plot. The way we do that is to make a list of the two lists and send that as a parameter.

Now, if we run the code, we'll see a plot showing both stock performance and bond performance. We can see in the histogram that the bonds have a much narrower range of outcomes than the stocks. And, if we look at the statistics that we printed out, we'll see that indeed, our stock funds have a higher average final balance, but also have a significantly higher standard deviation. Now, often people will not have all their money in one kind of account, like all in stocks or all in bonds. A more common approach will be to split the savings into two groups, such as half in a stock account, and half in a bond account. We can easily modify our code to see what that would look like.

Here we've created a third type of account, which we'll call the "mixed" account. We'll keep track of two balances in each iteration: the mixed balance for stocks and the mixed balance for bonds. In this case, I'll assume a 50/50 split between stock and bonds, so each of these will start with $500, keeping the total at $1000. Then, each year, we'll increase the mixed stock and the mixed bond accounts separately. After the new values are determined for each, we can look at the overall balance, and see whether they are out of balance. We can calculate

what 50% of the total was—this is the stock amount, and then the difference that we need to move from stocks to bonds or vice versa. Each year we'll do that rebalancing. One other change—rather than computing the two groups totally independently, we'll pick a year at random, and use the stock and bond data from that same year. Stock and bond performance can be related to each other, so we want to make sure we use comparable data for each. That means that we pick one of 35 years, and we pick the stock and the bond data for that year.

Again, running the code will show us a plot and the statistics we calculated. We can see that, like we'd expect, the mixed account will be in between the stock and the bond accounts in terms of average and in terms of standard deviation.

Now, what I've described is a relatively simple analysis, and there are some shortcomings. For instance, I'm just using some approximate overall stock and bond performances—particular types of stocks and bonds will perform differently. My point is not to make you an expert financial analyst, but it's to show you how you can use Monte Carlo simulations like this to analyze even your own accounts. This is exactly the type of approach that financial analysts will use to analyze different scenarios.

Before we go, let's look at one last simulation example, still a financial one, to see how it would work. Let's say that you have some retirement account, and have just retired. Hooray. You want to know if you'll be able to live off of that money during retirement. Now, there is a lot of uncertainty—how much will the account earn? How much will you have to withdraw each year to keep your standard of living, accounting for inflation? And, though it might not be the most pleasant thing to think about, how many years does it need to last?

Let's walk through a simple program to simulate all of these things. The program is similar in a lot of ways to the programs we saw earlier. We'll still have some sets of historical data, for stocks, bonds, and now for inflation. We'll also have functions to determine the increase in a stock account, an increase in a bond account, and the amount of inflation—

all of these just randomly chosen from among the historical examples. Notice that it would be easy to change this if we wanted. For example, if we wanted to assume that inflation was some fixed amount, we'd just change that inflation function to return that amount.

The program itself starts by asking the user to enter some key information. We'll ask how much money is in the account to begin with, what the annual expenses are, and how long the money needs to last.

We'll then set up some initial variables. We'll be keeping a count of how many times the money runs out. We'll assume we have a mixed account that is rebalanced, and we will set some percentage that should be in stocks—in this case, it is set to 50%. Then, we begin our Monte Carlo simulation, with a loop over some large number of scenarios.

Each iteration of this outer loop will be one simulation—what happens to the money in one example. So, we start out by setting the amount in the stock and bond accounts, and the annual expenses. We then loop through all the years, simulating the results from each year.

In this inner loop, representing one year, we first adjust our stock and bond balances based on returns and then rebalance. This is just like we did in the earlier examples, the only difference being that we have the amount to put in stocks as a variable.

After that, we calculate expenses. Each year, expenses should increase with inflation, so we multiply expenses times one plus inflation. Then, we subtract the expenses from our account, splitting the amount taken from stocks and bonds according to the percentage.

Now comes the important check—do we still have money? If our account has gone negative, it means we've run out of money. Oh no! What we'll do in this case is increase our count of running out of money, and then set our time to the maximum so that we won't keep going through this simulation.

At the end of the simulation, we'll check to see if we had a positive balance, and if so, store it in a list.

Finally, after all the simulations have been completed, we'll print out our information. We calculate the percentage of simulations where we had success—that is, where the money lasted as long as needed. For the successful cases, we will calculate the average ending balance and standard deviation, and plot the results in a histogram

OK, so if we run this code, we can enter some data. For example, if we say we have $1 million in the account and average expenses of $75,000, and we want this to last for 25 years of retirement, we can see the results. In this case, we've got about a 2/3 chance of making it. Now, if we do want to make it through retirement, we're likely to have a pretty good size account left over, on average more than $3 million on average, though keep in mind that inflation makes that not as big as it seems. But that 1 in 3 chance of not making it through retirement doesn't sound so great.

If you've ever used an online calculator to show you retirement projections or had a financial advisor show you retirement options, you might have seen results like this before. The code we've been using is a whole lot like what they will use—in fact, this code we have is more sophisticated than the systems that the average person encounters.

You could play around with this—try entering data from your own accounts, and see how it comes out. What happens if you change expenses, or the amount saved, or how long it needs to last, or the percentage in stocks vs. bonds? Can you keep some percentage in cash? And, you now know enough about programming that you can make this more sophisticated if you want to.

Keep in mind that the whole simulation is only as good as the model that we start with. For this case, we have a model based on sampling historical data about stocks, bonds, and inflation to predict the future. There's no guarantee that our model's right. For instance, maybe our model underestimates the possibility of an extreme change in the

stock market every so often. That might suggest a sort of fat tail in the distribution instead of a nice bell curve distribution.

Or, maybe the model's ok, but we find ourselves not on track to make it through retirement. In that case, we could run additional simulations to help decide which changes in spending and investment would be most helpful. I'd encourage you to explore these and other computer simulations for yourself.

Overall, it's hard to overstate the importance of computer simulations. In science, where there are many studies on scales of time or space that are extremely large or small, simulations on large supercomputers have increasingly become as important as actual measurement and observation, which can sometimes be either impractical or flat-out impossible.

Predictions that we all rely on about hurricanes and other extreme weather events are based on large computer simulations. Cities and nations regularly use simulations to predict future trends, and to decide how to spend resources and plan for the future. And there are climate simulations, where computers are used to predict the future of our entire planet. Taken altogether, the decisions made based on simulations have consequences measured in the trillions of dollars, are sometimes matters of life and death, and affect basically everyone on the planet.

Simulation was also where, in the 1960s, a programming language called Simula pioneered the use of objects and classes, pointing the way to a new approach to programming that we'll dive into beginning next time: a way to bundle together functions and data that came to be called object-oriented design. I look forward to seeing you there.

17

# Classes and Object-Oriented Programming

Object-oriented programming is a newer approach to software that has become widespread since the 1990s. One of its key benefits is called **encapsulation**, which means that all the parts and tools you need get packaged together—encapsulated in a **class**, whose individual instances are known as **objects**. As you will learn in this lecture, using classes and objects will help keep related code together and make it easier for you to design and manage different parts of the software.

## [ OBJECT-ORIENTED DESIGN ]

- An object-oriented approach differs from top-down and bottom-up approaches, which are both task-oriented designs. They tend to focus on the task and how to accomplish the task, either by decomposing the task into more basic tasks or building up from existing tasks we already know how to perform. In computer science terms, both approaches are focused on operations and bundling those operations into more and more sophisticated functions.

- By contrast, neither really focuses on the parts and materials we might need to accomplish any of those tasks. In computer science terms, neither squarely focuses on data.

- Classes and objects allow us to combine operations and data. When they are packaged all together, we have the data we need as well as the operations that work with that data.

- A class can be thought of as the general blueprint for some category, while the objects are specific instances of that category. In other words, the class is a type, while each object is just a variable.

› In object-oriented design, the design decisions we make have to do with what we want to represent and then the data and functions that are needed to represent that thing.

## [ CREATING CLASSES ]

› Let's say that we want to write some software that deals with a bank account. First, we want to think about the types of things we need to know about the bank account and the types of things we'd like to do with it.

› The main thing you need for a bank account is the balance—how much money is in the account. You might want to deposit money, or withdraw money, or maybe just check on how much is in the account.

› In order to create code that lets us manage bank accounts in a compact way, we can introduce the idea of classes. Classes are the containers in which we can package the data and the functions that belong with the data. Classes are basically a new type that a variable can take on, like integers, floats, strings, or lists.

› When we're defining a class, we start with the word "class," followed by the name of the class that we want to use. In this case, we're defining BankAccounts, so we'll name the class "BankAccount," which is followed by a colon. Then, everything in the class definition will be indented from there. The indentation shows that this is the stuff that belongs to that class. Our bank account needs to keep track of the current balance, so that is a data item we have. Indenting in, we call this data item "balance," and we start out by setting it to 0.

```
class BankAccount:
    balance = 0.0
```

› Let's see how we use these classes that we define. First, we can create an instance of the class. Each instance of a class is known as an object. We create an object that's an instance of a class by writing the class

name, followed by parentheses. We can assign this instance to a variable. In this case, we have a class called BankAccount, and we create one instance of the class, which we assign to the variable my_account.

- So, my_account is a single instance of BankAccount. We can then access the attributes of a BankAccount. We do this using a single period after the variable name and then stating the attribute of the class that we want. So, in this case, we write "my_account.balance," and that is going to give us the value of the balance. If we print that variable out, we will get 0, which was the value the balance was set to in the class definition.

```
class BankAccount:
    balance = 0.0
my_account = BankAccount()
print(my_account.balance)

OUTPUT
0.0
```

- Classes are a way of defining a new category of variable. A particular instance of that class is called an object. When we talk about object-oriented programming, we are talking about programming centered around creating and using objects—in other words, defining classes and then using instances of those classes in our programs.

- The idea of classes and objects is common across many languages, but terms for the variables that are within objects vary. In Python, the variables that are inside a class, and help define the class, are called "attributes" of the class. In Java, the parts of an object are called "fields"; in C++, they are called "member variables."

- In Python, some attributes can be set to apply equally across all members of a class, so that every object has that attribute, while other attributes can be defined individually for only some objects. In the previous example, "balance" was an attribute across the entire class—in fact, it was the only attribute of the class.

- To access an attribute within Python, we pick an object in the class and attach a period, followed by the name of the attribute.

- Let's look at our code again. First, we can assign values to the members of an object. After creating my_account, we can set my_account.balance to 100. Then, if we print out my_account.balance, it is 100.

```
class BankAccount:
    balance = 0.0
my_account = BankAccount()
my_account.balance = 100.0
print(my_account.balance)

OUTPUT
100.0
```

- Next, we'll create a second object, called "your_account." We'll set the balance of my_account to 100. If we print out the balance of your_account, the output is 0. We create two separate objects, each of which gets its own place in memory. So, my_account is one object, and your_account is another object. When we set the balance of my_account to 0, it only affects the "balance" attribute of my_account, and there's no change to the your_account balance. So, when we print out the your_account balance, we still get 0.

```
class BankAccount:
    balance = 0.0
my_account = BankAccount()
your_account = BankAccount()
my_account.balance = 100.0
print(your_account.balance)

OUTPUT
0.0
```

## [ MUTABLE DATA ]

› Let's say that we want to keep track of the deposits that were made to the bank account. So, in a bank account, we want to have a list of deposits made. We'll add a new attribute to the BankAccount class, called "deposits," and initialize it to be an empty list. So, BankAccounts now has two attributes: balance and deposits.

```
class BankAccount:
    balance = 0.0
    deposits = []
```

› Let's say that we create a BankAccount object and call it "checking_account." We can access the "deposits" part of the checking_account by writing "checking_account.deposits.append[100.0]," which should append the value 100 into the deposits list. If we print out "checking_account.deposits," we will get a list with 100.0 in it.

```
class BankAccount:
    balance = 0.0
    deposits = []
checking_account = BankAccount()
checking_account.deposits.append(100.0)
print(checking_account.deposits)

OUTPUT
[100.0]
```

› Let's say that we create a second BankAccount called "savings_account." We'll still append 100 into the checking_account list. If we print out the deposits list for the savings account, we get a list that has the 100 in it. We didn't change anything about the list in the savings_account, but it somehow has the value 100 in it.

```
class BankAccount:
    balance = 0.0
    deposits = []
checking_account = BankAccount()
savings_account = BankAccount()
checking_account.deposits.append(100.0)
print(savings_account.deposits)

OUTPUT:
[100.0]
```

- To understand this, we have to see what's happening in memory. Recall that a list is a mutable data type, which means that when we create a list, the variable doesn't store a copy of the list itself—it just has a value that says "the list is here." So, when we create a list, the actual list of elements is stored in one place, but the variable stores "this is the location of the list."

- In this case, the class description that defined "deposits" as an attribute said that deposits will be an empty list, but the problem is that it gives the same location of that empty list to every instance. Every object we create is going to start out with the exact same value for deposits, so it's going to be referring to the exact same list in memory. So, when we append 100 onto the checking_account list, that's the same list as the savings_account would see.

- One way around this is to reset the value of the attribute for a particular object. In the following, we set the value of checking_acount.deposits to be an empty list. This creates a new empty list and sets the value of deposits in the checking account to that list. The "deposits" attribute of the savings_account still points to the original empty list, and if we had other instances of BankAccount, they would, too. But now checking_account has its own deposits list to work with, separate from the rest. So, when we add 100 to the checking_account.deposits list, the savings_account list is unchanged.

```
class BankAccount:
    balance = 0.0
    deposits = []
checking_account = BankAccount()
savings_account = BankAccount()
checking_account.deposits = []
checking_account.deposits.append(100.0)
print(savings_account.deposits)

OUTPUT:
[]
```

› This works, but we could avoid this problem if we could just create a list to begin with that was separate for each object. In fact, there is a better way to approach the issue of attributes.

## [ METHODS ]

› In a Python class, we can have two types of attributes: class variables and instance variables. Everything we've seen so far is a class variable—that is, it's one variable defined for the class. When we create an object—that is, when we create an instance of the class—that instance will get its own versions of the class variables.

› But any initial values set in the class are going to be shared across all instances, which leads to problems with mutable data types. The alternative is to create instance variables, which are created separately for each instance of the class. With instance variables, we never need to worry about the changes to one object inadvertently affecting the attributes of a different object.

› To create instance variables, we need to introduce the topic of **methods**, which are like attributes, but instead of defining data, they define functions. One very special method is the "init" method, which gets its name from the fact that it is initializing an object within the class. The init method is commonly called a **constructor**.

- The init method gets defined much like a regular function, but it's inside the class definition. The init function also has syntax that differs in a few ways from other functions: It starts with a double underscore, then "init," and then another double underscore. It will take one parameter. Then, you have the colon, and the function definition is indented from there. In this case, we'll have one line in the init function: "self.deposits = []."

```
class BankAccount:
    balance = 0.0
    def __init__(self):
        self.deposits = []
```

- This special init function is a function that is executed whenever a new instance of the class is created. The term for this is **instantiation**. When you instantiate a new object, the Python compiler will find the constructor—that is, the init method—and run it.

- This first parameter, "self," in the init command is also a special one; it's not one you pass in, but it's automatically filled in. The self parameter refers to the current instance of the object. Because of the self parameter, we have a way of clearly referring to things within this particular instance of an object. So, if we write "self.balance," we mean the balance in this one instance.

- In this example, we have a deposits list that we want to be unique for each instance. We write "self.deposits" and set it equal to the empty list. That creates a unique "deposits" attribute for the instance and initializes that deposits list to the empty list.

- If we create two different BankAccount objects, like we did before, this init command is being called for each of them. We can still access the "deposits" attribute of the checking_account, just like before, and append something onto it. But notice that now it does not affect the "deposits" list for the savings account. Our init command created separate instance variables and initialized those individual instance variables independently.

```
class BankAccount:
    balance = 0.0
    def __init__(self):
        self.deposits = []
checking_account = BankAccount()
savings_account = BankAccount()
checking_account.deposits.append(100.0)
print(savings_account.deposits)

OUTPUT:
[]
```

› In general, we should try to use instance variables instead of class variables. So, instead of creating a "balance" class variable, it's better to create an instance variable in the init function, like we did for deposits. Practically, it's not much different, but it helps ensure that we know that the variable is something that can change from object to object. Generally, the only time we should use class variables is when there's some single value, usually one that's not likely to change, that should be the same across all instances of the class.

## Readings

Gries, *Practical Programming*, chap. 14.

Lambert, *Fundamentals of Python*, chap. 5.

Zelle, *Python Programming*, chap. 12.

## Exercises

For exercises 1 through 3, assume that you have the following class to keep track of inventory.

```
class Inventory:
    item = ""
    barcode = 0
    quantity = 0
    price = 0.00
    sales = 0.00
    def __init__(self, product, bar, pr):
        self.item = product
        self.barcode = bar
        self.price = pr
    def changeprice(self, newprice):
        self.price = newprice
    def sell(self, n):
        self.quantity -= n
        self.sales += self.price*n
    def restock(self, n):
        self.quantity += n
```

**1** What would be the output of the following code?

```
widget = Inventory("widget", 1112223334, 10.00)
widget.restock(30)
widget.sell(10)
print(widget.quantity)
print(widget.sales)
widget.changeprice(20.0)
widget.sell(10)
print(widget.quantity)
print(widget.sales)
```

2 What would be the output of the following code?

```
shoes = Inventory("shoe", 12345123245, 30.00)
shoes.restock(100)
shirts = Inventory("shirt", 9876598765, 25.00)
shirts.restock(80)
shoes.sell(10)
shirts.sell(30)
shoes.sell(50)
print(shoes.quantity)
print(shoes.sales)
print(shirts.quantity)
print(shirts.sales)
```

3 Write a method, "print," that will print out information about all the information about the inventory. For example, calling "widget.print()" would print out all the information about name, bar code, etc.

Imagine that you wanted a class to keep track of movies you've watched. You will want to keep track of the name of the movie, the genre, and a numerical rating of how much you liked it.

4 Define a class with attributes for these three characteristics, with some default values.

5 Write a constructor method that takes in values for all three attributes as parameters.

6 Write code that constructs a list of movies by asking a user for the appropriate information, until the user enters a movie with rating less than 0.

# Classes and Object-Oriented Programming

Suppose you have to assemble a piece of furniture. The manufacturer might list the tools needed, but it's up to you to get them. You might also have to provide some of your own parts and materials. And, sometimes the instructions might be so bad that you don't even know what tools or materials you need ahead of time. So you have to keep stopping to go find something or just get everything together. Here's the better idea: How about the manufacturer includes, in the package, all the tools and all the materials you need to assemble the product. All in one place. Wouldn't that be a whole lot better?

Clearly, when we have all the necessary tools and parts right there together in one package, life is much easier. And that's the beauty of object-oriented programming, a newer approach to software that has become widespread since the 1990s. One of its key benefits is called encapsulation, and that means exactly what I said—all the parts and tools you need are packaged together—encapsulated in a class, whose individual instances are known as objects.

An object-oriented approach differs from top-down and bottom-up approaches. Both top-down and bottom-up designs are task-oriented. They tend to focus on the task, and how to accomplish the task either decomposing it into more basic pieces or, or building up from existing tasks that we already know how to perform. In computer science terms, both approaches are focused on operations, and bundling those operations into more and more sophisticated functions. In terms of schoolhouse grammar, both top-down and bottom-up are focused mostly on verbs.

In contrast, neither really focuses on the parts and materials we may need to accomplish any of those tasks. In computer science terms, neither one is really focused on the data.

Classes and objects allow us to combine operations and data. Now, packaged all together, we have the data we need, and we have the operations that work with that data. In terms of grammar, we not only bring together all the verbs we want, but we also have all subjects and objects of those verbs.

A class can be thought of as the general blueprint for some category while the objects are specific instances of that category. So, if a class is a dog, then Toto, Lassie, Pluto, and Snoopy are objects in the class. The analogy is not perfect, but in terms of programming, a class Dog, in this case, is a new type of variable, while any particular object such as Toto is a particular variable. In short, the class is a type, while each object is a variable.

In object-oriented design, the design decisions we make have to do with what we want to represent, and then the data and functions that are needed to represent that thing. Let's consider an example—we'll talk about the design, and then we'll see how this gets put into code.

Let's say we want to write some software that deals with a bank account. First, we want to think about the types of things that we need to know about the bank account and the types of things that we'd like to do with it. Let's start really simple. The main thing you need for a bank account is the balance—how much money is in the account. Let's think of things you might want to do with the account. You might want to deposit money, or withdraw money, or maybe just check and see how much is in there. We'll make this a little more complex in a minute, but for now, that should be sufficient.

In order to create code that lets us manage bank accounts in a nice compact way like this, we can introduce the idea of classes. Classes are the containers in which we can package the data and the functions that belong with that data. Classes are basically a new type that a variable can take on, like integers, floats, strings, or lists.

Now, you still might wonder, why don't we just use functions to do everything? The reason is that when you just use functions, the functions

have no fixed tie to the data. In contrast, when we create classes we're tying these together—saying that some set of data and some set of functions are designed to work together. Then, if you want to deal with that data, your program already knows to use those functions to do it.

So, we're defining a class, and we start with the word "class," followed by the name of the class that we want to use. In this case, we're defining BankAccounts, so we'll name the class "BankAccount," and that'll be followed by a colon. Then, everything in the class definition will be indented from there. The indentation shows that this is the stuff that belongs to that class. Our bank account needs to keep track of the current balance, so that'll be a data item that we have. Indenting in, we'll call this data item "balance," and we'll start out by setting it to be 0.

Now let's see how we use these classes that we define. First, we can create an instance of the class. Each instance of a class is known as an object. We create an object that's an instance of a class by writing the class name, followed by parentheses. We can assign this instance to a variable. In the case we see here, we have a class called BankAccount, and we create one instance of the class, which we assign to the variable my_account. So, my_account is going to be a single instance of BankAccount. We can then access the attributes of a BankAccount. We do this using a single period after the variable name, and then stating the attribute of the class that we want. So, in this case, we write my_account.balance, and that is going to give us the value of the balance. If we print that variable out, here, we will get 0, which was the value the balance was set to in the class definition.

Let's look more closely at terminology. We first have classes, which are a way of defining a new category of variable. A particular instance of that class is called an object. When we talk about object-oriented programming, we are talking about programming centered around creating and using objects—in other words, defining classes and then using instances of those classes in our programs. The idea of classes and objects is common across lots of languages, but terms for the variables that are within objects vary.

In Python, the variables that are inside a class, and help define the class, are called attributes of the class. If you were in Java, the parts of an object would be called fields. In C++, the parts of an object are called member variables. In Python, some attributes can be set to apply equally to all members of a class, so that every object has that attribute, while other attributes can be defined individually for only some objects.

Now, in the example we just saw, "balance" was an attribute across the entire class—in fact, it was the only attribute of the class. To access an attribute within Python, we pick an object in the class, attach a period, followed by the name of the attribute.

OK, let's look at our code again. First, we can assign values to the members of an object. After creating my_account, we can set "my_account.balance" to 100. Then, if we print out "my_account.balance," it is 100. Let's see if you can figure the next one out. We'll create a second object, called "your_account." I'll set the balance of my_account to 100. What happens if I print out the balance of your_account? The output will be 0. What happens here is that we create two separate objects. Each of these objects gets its own place in memory. So, there is one object my_account and another object your_account. When we set the balance of my_account to 0, it only affects the "balance" attribute of my_account, and there's no change to the balance in your_account. So, when we print out the your_account balance, we still get 0.

Now, this all seems good, and you might think, "Hey, I understand this," but there's a big issue. Let's see how we can extend this class a bit. Say that we want to keep track of the deposits that were made to the bank account. So, in a bank account, we want to have a list of deposits made. We'll add a new attribute to the BankAccount class, called "deposits," and initialize it to be an empty list. So, BankAccounts now has two attributes, balance and deposits. Let's say that we create a BankAccount object and call it "checking_account." We can access the "deposits" part of the checking_account by writing "checking_account.deposits.append[100.0]." That should append the value 100 into the deposits list. Sure enough, if we print out "checking_account.deposits," we'll get a list with 100.0 in it.

Now, let's make this a little trickier. Let's say we create a second BankAccount, called "savings_account." We'll still append 100 into the checking_account list. Now, what if we were to print out the deposits list for the savings account? What do you think we'd get? We get a list that has the 100 in it. Woah. What is going on here? We didn't change anything about the list in the savings_account, and yet it somehow has the value 100 in it.

To understand this, we have to see what's happening in memory. Let's recall the whole discussion of mutable and immutable data types: in particular, remember that a list is a mutable data type. What that means is that when we create a list, the variable doesn't actually store a copy of the list itself, it just has a value that says "the list is here." So, when we create a list, the actual list of elements is stored in one place, but the variable stores "this is the location of the list". In this case, the class description that defined "deposits" as an attribute said that deposits will be an empty list, but the problem is, it gives the same location of that empty list to every instance. Every object we create is going to start out with the exact same value for deposits, and so it's going to be referring to the exact same list in memory. So, when we append 100 onto the checking_account list, that's the same list as the savings_account would see.

Now, this is probably not what we want, so let's look at a couple of ways around this. One way is that we could reset the value of an attribute every time we create an object. Here, we set the value of checking_acount.deposits to be an empty list. Now, when we do this, this creates a new empty list and it sets the value of deposits in the checking account to that list. The "deposits" attribute of the savings_account still points to the original empty list, and if we had other instances of BankAccount, they would too. But, now checking_account has its own deposits list to work with, separate from the rest. So, when we add 100 to the checking_account.deposits list, the savings_account list, is unchanged.

This works, but it'd be nicer if we could just create a list to begin with that was separate for each object so that we never ran into this problem.

In fact, the way I'm going to show you now is the better way to approach the whole issue of attributes.

In a Python class, we can actually have two types of attributes: class variables, and instance variables. Everything we've seen so far is a class variable. That is, it's one variable defined for the class. When we create an object, that is, when we create an instance of the class, that instance will get its own versions of the class variables.

But, any initial values set in the class are going to be shared across all instances, and that can lead to problems with mutable data types like we just saw. The alternative is to create instance variables, and these are variables created separately for each instance of the class. With instance variables, we never need to worry about the changes to one object inadvertently affecting the attributes of a different object. To create instance variables, we will need to introduce the topic of methods. Methods are like attributes, but instead of defining data, they define functions.

One very special method is the init method, which gets its name from the fact that it is initializing an object within the class. The init method is often referred to as a constructor. The init method gets defined much like a regular function, but it's inside the class definition. The init function also has syntax that differs in a few ways from other functions: it starts with a double underscore, then init, then another double underscore. It'll take one parameter. Then, you have the colon, and the function definition is indented in from there. In this case, we'll have one line in the init function: self.deposits equal the empty list. Let me explain how this works.

This special init function is a function that is executed whenever a new instance of the class is created. The term for this is instantiation. When you instantiate a new object, the Python compiler will find the constructor, that is, this init method, and run it. Now, this first parameter "self" in the init command is also a special one—it's not one you pass in, it's automatically filled in. The self parameter refers to the current instance of the object. Because of the self parameter, we have a way of

clearly referring to things within this particular instance of an object. So, if we write self.balance, we mean the balance in this one instance.

So, in the example that we have here with the init command for the bank account, we have a deposits list that we want to be unique for each instance. We write self.deposits and set it equal to the empty list. That creates a unique "deposits" attribute for that instance and then initializes that deposits list to the empty list.

If we go ahead and create two different BankAccount objects like we did before, this init command is being called for each of them. We can still access the "deposits" attribute of the checking_account, just like before, and we append something onto it. But, notice that now it does not affect the "deposits" list for the savings account. Our init command created separate instance variables, and initialized those individual instance variables independently.

Generally speaking, we should try to use instance variables instead of class variables. So, instead of creating a balance class variable, it's better style to create an instance variable in the init function, just like we did for deposits. Practically speaking, it's not much different, but it helps ensure that we know that the variable is something that can change from object to object. Generally, the only time we should use class variables is when there's some single value, usually one that's not likely to change, and it should be the same across all instances of the class. The knights of the round table will all be knights, but they may not all be brave.

Now, we can also modify the init function so that it takes in a parameter. Let's say that we want the bank account to be created with some initial balance. We can take that balance in as a parameter. We just list that parameter after the self parameter in the init statement. Remember that self is always going to be the first parameter, and it's not one that we actually pass in. So, the next parameter that we'll take in is initial_amount. And that will be used to set the balance to that initial amount.

Now, when we actually create an object in our program, we can pass this new variable in as a parameter. Here we'll create our checking account by creating a new BankAccount, passing in the parameter of 200. That parameter will be the initial_amount in our init function, and so our balance gets set to 200. Sure enough, when we print out the balance of the account, we see that it's 200.

We can give default parameters here, just like we have for other functions. It would make sense that if we don't have an initial balance specified, that it should be 0, so we'll use that as the default value. Then, we can create a new BankAccount object, either passing in the amount or just leaving that parameter value empty. As we see here, we create a checking account with an initial balance of 200, and a savings account with no initial balance specified. Then, printing out the balances, we see that they are 200 and 0, just like we'd expect.

Besides the init method, we can create other methods in our class. Let's think back to some of the functions that we wanted to do with a bank account. I remember three of them—making a deposit, making a withdrawal, and checking the balance. We call these functions, defined inside a class, methods. The C++ term is member functions. Methods are a part of the class itself and are defined in the context of the class.

We've seen this notation before, like when we worked with files. When we open a file, the thing that we've assigned to a variable is a file object, It not only has the file data but also the file operations that can work on the file. For example, it includes the operation "close" for when you are done with it and write to write data to the file. The close and write methods are part of that file object. As you start looking at Python code, including the code from lots of modules, you'll see these calls to methods all over the place.

Defining a method works just like defining a function, except that it's inside of the class definition. There's one other important difference: Methods always have this first parameter, which by convention is called self. Just like the init method, the self parameter doesn't get explicitly

passed in, and it's used to refer to the current instance of the object. Let's see an example, where we want to make a deposit.

We will define a function "makeDeposit" within the class definition. There will be two parameters, "self" and "amount". Inside the function, we will increase the balance of the account by "amount" and we will also append this on to the list of deposits that we've made. Notice that we have to refer to self-dot-balance and self-dot-deposits here.

Here's how we'd actually use this. Let's say we created a checking account with an initial balance of $100, and want to deposit another $50. We would call "checking_account.makeDeposit" and pass in the number 50. Notice that we don't need to pass in "self," since that first parameter of the method is filled in automatically. Then, if we print out the balance and the deposit list, we'd see that the balance is indeed $150, and the $50 deposit is in the list.

Let me give you something to try. Say you wanted the class to include a "withdraw" function that would let you withdraw a certain amount of money. Try writing a "makeWithdrawal" method, and an example of it being used.

Here is what that might look like. Inside the class, we have our method definition for makeWithdrawal. The method takes in two parameters, self and amount. It then decreases the balance by the amount. That's all. If we call this function to withdraw $70, in this case saying "checking_account.makeWithdrawal (70)," we indeed have the balance reduced from the $150 it was to just $80 left.

Now let's create a way to check the balance. That should be really easy—we've already been doing it. We can just take the account name, and write .dot-balance after it. We could just access the balance directly. And, our line "print [checking_account.balance]" would print out the current balance.

That seems like a nice simple way to access data, and truthfully it is. If you have a really simple class, it might be that this is how you access

the attributes. But, in practice, it's probably best not to just access an attribute like that. Let me give you a sense of why another way would be better. Say we had a bank account, that started at $100. Then, let's say we decided to make a deposit. Instead of using that "makeDeposit" method that we just defined, instead, we'll just increase the balance directly. That's doing the same thing, right?

Well, no. In fact, we were doing more than just increasing the balance when we made a deposit; we were also keeping a list of the deposits that had been made so far. When we bypassed the "makeDeposit" method on our own, we created an inconsistency—we did change the balance, but we forgot to update the list of deposits. So, when we print out the results, we see the increased balance, but our list of deposits is empty.

This goes to a principle of good object-oriented design: that we should only deal with the data in an object—I'm talking about the attribute values—through the methods of that object. In our bank account case, that means we should use the makeDeposit and makeWithdrawal commands to add and withdraw money from the account, rather than directly manipulating the balance ourselves.

In Python there is a convention you can use to help encourage this good behavior. The idea is that all the attributes that a person really shouldn't access directly should have a name starting with an underscore. Let's see how this would look in the class definition. The change to what we already have is pretty simple. We just replace "balance" and "deposits" with "_balance" and "_deposits". Everything else works the same.

Now, if we try to have the same code we saw earlier, we get an error. Specifically, we get an "AttributeError" telling us that there is no attribute named "balance." These attributes that we're not supposed to access directly are called private attributes. We could still get around this by putting "_balance" in our code, but we'd, at least, have to be intentionally acknowledging that we're trying to access something that shouldn't.

The problem is that we still want to be able to find the balance in our account, and to get a list of the deposits. So, we'll actually need to create methods to provide this information, rather than accessing the attributes directly. We can do this by creating functions getBalance and getDeposits. Each of these can return the associated data that was meant to be hidden or private. We can then call these functions to get the correct data returned. We can see the balance with getBalance and the list of deposits with getDeposits. And, we never had to directly touch the actual attribute inside the class—we just called these methods to work with that data.

This is another example of abstraction in action. The user of a class doesn't need to worry about the internal structure, but just how to make the calls to get the information needed. For example, maybe instead of keeping a list of deposits, we keep a list of both deposits and withdrawals. When someone asks for the deposits list, we could go through the list and extract the deposits into a separate list that gets returned. It shouldn't matter to the user of the class exactly how that information is stored inside the object.

Also, I want to be crystal clear. It is OK for the attributes of a class to be other classes. Many modules and packages don't just provide functions; they also provide classes that we can use. An example of this is the standard module datetime. Among the classes it provides is one for storing and working with dates.

We could easily incorporate a date class into our BankAccount class. Just to illustrate, consider this example. We first import the date class from the datetime module. Then, in our initialization routine, we can also create a new attribute, opendate. For simplicity, I'll just initialize the date to March 15 of 2011. When we create a checking account and print the opendate attribute, we see that the March 15, of 2011 date is indeed printed.

Notice that we used an object within an object. We can nest classes inside of classes like this indefinitely: a class can have an attribute that's a class, which can have an attribute that's a class, etc., etc. And, the

class can be imported from a module—it doesn't have to be something we create ourselves.

Before we finish up, I want to talk for a minute about mutable data. First, I want to show you something about how returned values work, in particular why you need to return a copy of a data list, instead of the actual list of deposits. This sort of thing is not unique to object-oriented programming, but it does tend to come up much more in object-oriented programming.

In our example, we passed back a list of deposits. Let's look at a little code. Let's say that we create a bank account object, make a deposit, and then get a list of deposits. In this case, we've created a checking_account, deposited $50, and then gotten our list of deposits, which we have assigned to the variable *x*. Now, if we append a value like 1000 on to *x*, and then later we print out the list of deposits, what do you think will happen? Keep in mind that lists are mutable data types.

The resulting list is going to have both the 50 and the 1000 in the list. Keep in mind that for a mutable data type, like a list, when we pass it back and forth, we are passing something that can be changed. So, when we return our list of deposits from the bank account, we are returning a list that can be changed—anything you do in one place will affect what shows up everywhere with that list. In this case, when we append onto the list in the main code, it's affecting the list that's inside the checking_account object.

Now, this is probably not the way we want this to behave. When we return a list of deposits, we don't want whatever users do with that list to affect what's inside the object. So, how might we do that—how might we return something out of the object that's not going to let people mess with the object's internal information? The solution here is to make a copy of the list, so I'll show you a simple way to make one.

Let's go back to our getDeposits method. We just returned the deposits list that was one of our internal attributes. Here's a new version of that method. We first create a copy of that deposits list. Then, we return

that copy. We create the copy using the slicing operations that we can perform on a list. In general, slicing creates a new list by specifying a subset of some other list, where the starting and ending points are specified in brackets, with a colon in between. In this case, leaving the starting and ending points empty means we want to start at the beginning and go to the end. So, when we write "self._deposits[:]," we're creating a new list that's a copy of the entire length of the old one. In other words, we've made a complete copy of the list.

Now, if we return to the same code we had earlier, where we pull out a list of deposits, and we add something to it, we'll see that the original list stored in the object itself is unchanged. Again, although the idea of passing around mutable data is not unique to object-oriented programming, it does come up here a lot more frequently, so you need to be cautious as you pass data around, to make sure you are using copies, whenever you don't want the underlying data to be changed.

Now, one other important thing to note about objects is that all objects are themselves mutable data. So, we can pass in an object to a function, change the object within the function, and return and the object will be changed. Let's look at a quick example.

Let's say we want to create a function to simulate winning the lottery. We take in an account, and we make a $10 million deposit to that account. So, in the code you see here, we have that winLottery function definition, where we bring in an account and call makeDeposit on the account. We then have some code that creates a new checking account, which should have a balance of $0 since we haven't specified a starting balance. We then call winLottery on that checking account. Now, if we print balance of the account, we will get back $10 million. In other words, calling the winLottery command actually changed the value of the account itself. If the checking_account was not mutable, this wouldn't have happened.

Most of the time, it's really nice that objects are mutable. That's what makes it so easy to write a command like winLottery. If only winning the actual lottery was so easy. But, there may be times like we saw when

we were returning the list of deposits, where having a mutable data type is not ideal. The key issue is to be aware of what type of data you are using and make sure you make copies when needed.

One other point: In Python, object-oriented programming does not need to replace other ways of programming. You can augment programs you have by including classes and objects. For example, the financial simulation that we made earlier could be converted to an object-oriented approach very easily. We could extend the bank_account class we've been using in this lecture, and let it hold balances of stocks and bonds as separate attributes. We could also create a method to apply one year of interest. This would make our operations in any one loop much simpler.

When object-oriented programming really started taking off in the 1990s, it offered a way of organizing code in ways that people already knew was good, but didn't have the built-in structures to support.

The early promises of object-oriented programming were that it would revolutionize software development, reducing the challenges in design and lead to much more reusable code. And while it might not have achieved the goals of its most optimistic proponents, programmers have found it to be a very useful way to organize their data and operations, and it's become a central part of most modern programming languages. As you develop code, I'd encourage you to use classes and objects as often as you can—whenever data and actions on that data can naturally be grouped together. Using classes and objects will help keep related code together and make it easier for you to design and manage different parts of the software.

In the next lecture, we'll see how encapsulation provides the basis for two other object-oriented features, inheritance and polymorphism, and we'll see more about how we can use object-oriented design as we put together our programs. I'll look forward to seeing you then.

## MORE PYTHON PRACTICE

Let me warn you against one thing that can be done with classes. Let's say that we create a checking account, and then we try to set the "name" attribute of the account. Wait a minute—we didn't have a "name" attribute in the account. What do you think will happen? Oddly enough, Python will let you assign something to an attribute that didn't already exist. When this happens, a new attribute is created for just that one object. So, in the code you see here, we assign the name "My Checking Account" to the "name" attribute. We can then print out that name and sure enough, the name is just what we had set. Now, say that we had another bank account, a savings account. If we try to access the name element of the savings account, what will happen? We get an error.

What has happened is that our checking account memory has had an extra attribute, name, added on to it. The attribute is not added to the class, so the savings account doesn't have any corresponding "name" element. We can access "name" just fine in the checking account object, but not in the savings account object.

Now, generally, I'd recommend that you avoid doing things like this. It can be really confusing when you've created one object that has different attributes from all the other objects from that class. But, you should be aware of this, in case you read some code where someone did this, or you accidentally do something like this yourself in your own code—since it's not a bug, the compiler won't catch it for you.

18

# Objects with Inheritance and Polymorphism

Children inherit from their parents the fundamental structure of DNA that makes us human—the fundamental traits that make human bodies work and grow. Inheritance plays a similar role in programming, thanks to object-oriented design and programming. The fundamental idea of object-oriented design is encapsulation, where we put together the data, and functions that work on that data, into a single package. But there are two other aspects of object-oriented programming—inheritance and polymorphism—which you will learn about in this lecture.

## [ INHERITANCE ]

- Imagine that we have a program that we're going to use to keep track of statistics for different players on a sports team. Often, players with different positions will have different statistics that are relevant to them. In football, players on offense will have different statistics than those who play defense.

- For a quarterback, we probably want to know the player's name and team, as well as data like the number of passes attempted, the number of completions, and the number of passing yards. We can set up some functions associated with the quarterback to help us compute percentages and averages, such as the percentage of completed passes or average yards gained per pass.

- For the position of running back, we would want the player's name and team, as well as the number of rushes and rushing yards gained.

- With just these two positions, one possibility would be to create two classes. We could create a "Quarterback" class (which would have attributes for name, team, pass attempts, completions, and passing yards) and a "RunningBack" class (which would have attributes for name, team, rushes, and rushing yards).

- Notice that both quarterback and running back share some attributes. They both have the player's name and the player's team. In fact, this would be true for all the various positions we might want to define.

- So, let's imagine a different organization. Let's say that we have some base type, which we'll call "FootballPlayer," which will have all the attributes common across the various types of players. In this case, that's the player's name and the team. We can then define a Quarterback as a type of FootballPlayer, with some additional attributes: passes attempted, completions, and passing yards. Likewise, a RunningBack is also a type of FootballPlayer with some additional attributes: rushes and rushing yards.

- What we've just seen is called **inheritance**. You can think of the football player as the parent. Then, the quarterback and the running back are children. The children inherit the characteristics of the parent. In this case, the children inherit the name and team attributes defined for the football player.

- We define each of the classes individually. For the FootballPlayer class, we define it the same way we have been. For the Quarterback and RunningBack classes, we put the name of their **parent class** in parentheses. Each of them defines the attributes unique to that class—the attributes that were not in the parent class.

```
class FootballPlayer:
    name = "John Doe"
    team = "None"
class Quarterback(FootballPlayer):
    pass_attempts = 0
    completions = 0
    pass_yards = 0
class RunningBack(FootballPlayer):
    rushes = 0
    rush_yards = 0
```

- So, we have our FootballPlayer class with a name and team defined, and we set the values to be "John Doe" for the name and "None" for the team.

For a Quarterback, we note that it is a child of the FootballPlayer class, and then we define the "pass_attempts," "completions," and "pass_yards" attributes, initializing all of them to 0. For a RunningBack, we again declare that it is a child or the FootballPlayer class, and then we define the "rushes" and "rush_yards" attributes, again initializing them to 0.

› With those classes defined, we can then create an instance of a class. Let's say that we create a player, called player1, that is a Quarterback. We can print the player's name. Becuase we didn't set the name, it uses whatever the default name was for all football players, which we said would be "John Doe"—that is, the quarterback has a "name" attribute because its parent class, FootballPlayer, had a name attribute. We can also print out the pass_yards attribute, which is 0, just like we initialized it to be.

```
player1 = Quarterback()
print(player1.name)
print(player1.pass_yards)

OUTPUT:
John Doe
0
```

› Let's say that instead of a quarterback, we had said that player1 was a RunningBack. We'd still have the name attribute, because a running back is a child of the FootballPlayer class, and FootballPlayer has a "name" attribute. However, if we tried to print off the pass_yards, we'd get an error. Pass yards were defined only for the Quarterback class.

```
player1 = RunningBack()
print(player1.name)
print(player1.pass_yards)

OUTPUT:
John Doe
AttributeError: 'RunningBack' object has no attribute 'pass_
  yards'
```

› Creating different players is straightforward. For example, we can create player1 as a Quarterback and player2 as a RunningBack and set all the values appropriately.

```
player1 = Quarterback()
player1.name = "John"
player1.team = "Cowboys"
player1.pass_attempts = 10
player1.completions = 6
player1.pass_yards = 57
player2 = RunningBack()
player2.name = "Joe"
player2.team = "Eagles"
player2.rushes = 12
player2.rush_yards = 73
```

› It's not just attributes that can be inherited. We can also inherit methods. In the following, we've augmented our classes to include some methods. For the FootballPlayer class, we'll define a method, "printPlayer," that prints out the name and team of the player. We'll also add some methods for the Quarterback and RunningBack classes to compute some statistics specific to those positions. For a quarterback, that will be the completion rate and yards per attempt, and for the running back, that will be the yards per rush.

```
class FootballPlayer:
    name = "John Doe"
    team = "None"
    years_in_league = 0
    def printPlayer(self):
        print(self.name+" playing for the "+self.team+":")
class Quarterback(FootballPlayer):
    pass_attempts = 0
    completions = 0
    pass_yards = 0
```

```
    def completionRate(self):
        return self.completions/self.pass_attempts
    def yardsPerAttempt(self):
        return self.pass_yards/self.pass_attempts
class RunningBack(FootballPlayer):
    rushes = 0
    rush_yards = 0
    def yardsPerRush(self):
        return self.rush_yards/self.rushes
```

› We can go back to our two players that we defined earlier, and then we can call the methods for these players. Notice that we can call printPlayer for both player1 and player2. Because the method is defined in the parent, it's automatically inherited by the children. We can also call those statistics methods that are specific to each of the children classes.

```
class FootballPlayer:
    name = "John Doe"
    team = "None"
    years_in_league = 0
    def printPlayer(self):
        print(self.name+" playing for the "+self.team+":")
class Quarterback(FootballPlayer):
    pass_attempts = 0
    completions = 0
    pass_yards = 0
    def completionRate(self):
        return self.completions/self.pass_attempts
    def yardsPerAttempt(self):
        return self.pass_yards/self.pass_attempts
class RunningBack(FootballPlayer):
    rushes = 0
    rush_yards = 0
    def yardsPerRush(self):
        return self.rush_yards/self.rushes
```

- When people discuss inheritance, there are different terms used for the different classes. Sometimes, we call the parent class the "base" class, and we call the children "derived" classes. Other times, we call the parent a "superclass" and the children "subclasses." All of these terms refer to the same thing.

- Just like in biology, where you can inherit traits from more than just one parent, classes can inherit properties from multiple parents. But for the most part, you should stay away from multiple inheritance. It can make your code more confusing to follow. Plus, it's very rare that multiple inheritance is actually the "right" solution to your problem.

## [ POLYMORPHISM ]

- The third main feature of object-oriented programming is **polymorphism**, which means that a function, or method, can take on many different forms, depending on the context.

- Let's return to our example with the football players. Let's imagine that we want to assess whether each player is "good" or not, according to some measure we devise. Clearly, the way we determine whether a quarterback is good at throwing is different from the way we determine whether a running back is good at running.

- So, let's say that we'd like to have a method called "isGood" that returns "True" or "False," depending on whether a player is good or not, according to whatever method we devise. We can augment our earlier definitions to include this function.

- Let's put a function "isGood" in the FootballPlayer class. It's not possible to determine whether a generic football player is good or not, given that all we have is a name and team. So, in this case, we'll print out some sort of error message, saying that we called a function that wasn't defined.

```
class FootballPlayer:
    name = "John Doe"
    team = "None"
    years_in_league = 0
    def printPlayer(self):
        print(self.name+" playing for the "+self.team+":")
    def isGood(self):
        print("Error! isGood is not defined!")
        return False
```

› Let's assume we have two players that we've created, just like before. We're now going to create a "playerlist," and we'll add both player1 and player2 into that list. Then, we'll go through each of the players in the list, using a for statement. For each player, we'll print out the player information using the printPlayer method, and then we'll call isGood and print out whether the player is a good player or not a good player.

```
player1 = Quarterback()
player1.name = "John"
player1.team = "Cowboys"
player1.pass_attempts = 10
player1.completions = 6
player1.pass_yards = 57
player2 = RunningBack()
player2.name = "Joe"
player2.team = "Eagles"
player2.rushes = 12
player2.rush_yards = 73
playerlist = []
playerlist.append(player1)
playerlist.append(player2)
for player in playerlist:
    player.printPlayer()
    if (player.isGood()):
        print("  is a GOOD player")
    else:
        print("  is NOT a good player")
```

```
OUTPUT:
John playing for the Cowboys:
Error! isGood is not defined!
  is NOT a good player
Joe playing for the Eagles:
Error! isGood is not defined!
  is NOT a good player
```

› When we run this, we get error messages printed. That's what we'd expect.

› An error is not what we want; we want to be able to make comparisons. So, we'll define isGood as a function in each of the children classes. The following is what that will look like. In both of the child classes, we'll create a function isGood. For the quarterback, we return whether the yards per passing attempt are above some level. For the running back, we return whether the yards per rush are above some level.

```
class FootballPlayer:
    name = "John Doe"
    team = "None"
    years_in_league = 0
    def printPlayer(self):
        print(self.name+" playing for the "+self.team+":")
    def isGood(self):
        print("Error! isGood is not defined!")
        return False
class Quarterback(FootballPlayer):
    pass_attempts = 0
    completions = 0
    pass_yards = 0
    def completionRate(self):
        return self.completions/self.pass_attempts
    def yardsPerAttempt(self):
        return self.pass_yards/self.pass_attempts
    def isGood(self):
        return (self.yardsPerAttempt() > 7)
```

```
class RunningBack(FootballPlayer):
    rushes = 0
    rush_yards = 0
    def yardsPerRush(self):
        return self.rush_yards/self.rushes
    def isGood(self):
        return (self.yardsPerRush() > 4)
```

› Using the exact code as we had before, if we run it now, we get an output without the error messages.

```
player1 = Quarterback()
player1.name = "John"
player1.team = "Cowboys"
player1.pass_attempts = 10
player1.completions = 6
player1.pass_yards = 57
player2 = RunningBack()
player2.name = "Joe"
player2.team = "Eagles"
player2.rushes = 12
player2.rush_yards = 73
playerlist = []
playerlist.append(player1)
playerlist.append(player2)
for player in playerlist:
    player.printPlayer()
    if (player.isGood()):
        print("  is a GOOD player")
    else:
        print("  is NOT a good player")

OUTPUT:
John playing for the Cowboys:
  is NOT a good player
Joe playing for the Eagles:
  is a GOOD player
```

- When the Python compiler sees the call to isGood, it first looks at the definition of isGood in the child class. If there's not a definition of that method there, it will look at the parent to see if the method is defined there.

- Inheritance is useful if you're defining your own set of classes, but we can actually inherit from any other class. Python even lets you treat basic types like strings or integers as a parent class. Especially useful is the fact that we can use inheritance to create our own exceptions (which are used to catch errors that would otherwise cause the program to crash).

## [ JSON AND PICKLE ]

- JSON and pickle are two important Python modules that make objects much more usable. They are included in the standard library and can help make it easy to handle objects.

- JSON (JavaScript Object Notation) is a way of structuring data in a text format. It uses a syntax for writing objects that's similar to the way objects are defined in Java and Javascript. JSON data is a human-readable string as opposed to binary data. JSON groups information in objects using curly braces, with each attribute written as an attribute name, followed by a colon, followed by the value. The value can be a new object, nested inside the previous object.

- JSON is perfectly capable of representing our objects, and because it's a text format, we can read and write JSON data easily. Plus, it's something that is independent of the language that it was produced in. For this reason, JSON is the most common way that data files are transmitted over the web.

- Python's JSON module includes commands that let us convert data to and from JSON. Basically, the JSON routines let us convert a piece of data into a JSON string. Most, but not all, Python data types can be converted to JSON. The JSON string can be written to or read from a file like any other string.

- Pickle is a module that lets you read and write data other than strings more easily. Pickle lets us read and write data from a file in binary format (which is how most files you encounter every day are stored, from images to word-processing files). And it works for even more data types than JSON.

- Pickle is a Python-specific format, though. If you write a file using pickle commands, it needs to be read by another Python program also using pickle commands. Pickle should not be used for writing data that you need to send to other people, and you should never read pickle-produced files from others unless you are certain of the source, because it's easy for them to contain malicious data.

## Readings

Lambert, *Fundamentals of Python*, chaps. 5–6.

Zelle, *Python Programming*, chap. 12.

## Exercises

**1** Assume that you have a class, "Game," defined as follows.

```
class Game:
    name = ""
    numplayers = 0
```

How would you define the following?

a) A video game class "Videogame" that has the same attributes as a "Game" and also keeps track of the platform that it is.

b) A board game class "Boardgame" that has the same attributes as a "Game" and also has a number of pieces and a size (stored as a list of two numbers: a length and a width).

2 For the “Game” class, assume that there is a method defined.

```
def print(self):
    print(self.name)
    print("Up to ", self.numplayers, "players")
```

How would you define functions so that calling “Videogame.print()” and “Boardgame.print()” give different printouts, reflecting the information they contain?

3 Write code to create a video game, and then print its information out.

4 How would you use the pickle module to save the video game from exercise 3 into a file, “Game.dat”?

5 How would you read in a game saved in “Game.dat” to a variable “savedgame”?

TRANSCRIPT 18

# Objects with Inheritance and Polymorphism

Children inherit some really important stuff from their parents. I'm not just talking about eye color or height; I mean the fundamental structure of DNA that makes us human. Children can vary from parents in lots of ways, but the most fundamental traits that make human bodies work and grow are inherited. Believe it or not, inheritance plays a similar role in programming thanks to object-oriented design and programming.

Remember what an object is. An object lets us group both data and functions together into one package. We can put a lot of related data together in that package, along with functions to operate on the data. This gives us a level of abstraction. We don't worry about how the stuff inside the object works, we just worry about what method we can call to work with it. But most importantly, objects let us group together stuff that's related. Also, remember that objects are defined within a class, where an object is an actual instance of that class. My personal bank account might be one of many objects within a class called BankAccount.

Classes, let's recall, are defined by a class name. Inside the class are a set of variables called attributes, and a set of functions called methods. These attributes and methods are actually part of the class. The example here shows a class for recording information about gifts. There's the init constructor that gets called for every new object, as well as several accessor functions. To create an object, we assign a variable a new instance of that class. We can then call the methods on that object or access the attributes of that object. Here, I'm using that gift class to store information about a shirt given to me by my mother on my birthday in 2015. I then change that record so that it's from my sister instead.

This is encapsulation, the fundamental idea of object-oriented design where we put together the data and functions that work on that data into a single package. But there are two other aspects of object-oriented

programming, and those are the ones we'll talk about in this lecture: inheritance and polymorphism.

Let's first start by seeing how inheritance works. Imagine that we have a program, and we're going to use it to keep track of statistics for different players on a sports team. Often, players with different positions will have different statistics that are relevant to them. For example, in soccer or hockey, a goalie is going to have saves while other players are going to have goals. In baseball, a pitcher will have different statistics than a batter, and in football, players on offense will have different statistics than those on defense.

I'll use an American football illustration here, though you don't really need to know anything about football to follow along. For a quarterback, we probably want to know the player's name and team, as well as data like the number of passes attempted, the number of completions, and the number of passing yards. Someone who follows this sport closely might want a lot more information, but this is just for illustration.

Now, we can set up some functions associated with the quarterback to help us compute percentages and averages, such as the percentage of completed passes, or the average yards gained per pass. Now, for the position of running back, we would again want the player's name and team, and now the number of rushes and how many rushing yards were gained. We could continue in this way to other positions such as wide receivers, and defensive players, and kickers, and so on.

But let's stay with just two positions. One possibility would be for us to create two classes. We could create a quarterback class and a running back class. The quarterback class would have the attributes for name, team, pass attempts, completions, and passing yards. The running back class would have attributes for name, team, rushes, and rushing yards. But, let's look at this for a minute. Notice that both the quarterback and the running back share some attributes. They both have the player's name and the player's team. In fact, this would be true for all of the various positions that we might want to define.

So let's imagine a different organization. Let's say that we have some base type, which we'll call Football Player. A football player will have all the attributes that are common across all the various types of players. In this case, that's the player's name and the team. We can then define a quarterback as a type of football player with some additional attributes—passes attempted, completions, and passing yards. Likewise, a running back is also a type of football player with some additional attributes—rushes and rushing yards.

What we've just seen is called inheritance. You can think of the football player as the parent, then the quarterback and the running back are the children. The children inherit the characteristics of the parent. In this case, the children inherit the name and team that were defined for the football player.

Let's look at some code, and to keep things simpler, I'm going to illustrate with examples where the attributes are not private. That is, I'm not going to have attributes starting with an underscore. Nor am I going to have accessor functions.

We define each of these classes individually. For the FootballPlayer class, we define it the same way we have been. For the Quarterback and RunningBack classes, notice that we put the name of their parent class in parentheses. Each of them just defines the attributes that are unique to that class—the attributes that were not in the parent class. So we have our FootballPlayer class defined with a name and team, and we set the values there to be John Doe for the name, and None for the team. For a Quarterback, we note that it is a child of the FootballPlayer class, and then we define the pass_attempts, completions, and pass_yards attributes, initializing all of them to 0. For a RunningBack, we again declare that it's a child or the FootballPlayer class, and then we define the rushes and rush_yards attributes, again initializing them to 0.

With those classes defined, we can then create an instance of a class. Say we create a player, called player1, that's a Quarterback. We can print the player's name. Since we didn't set the name, it uses whatever the default name was for all football players, which we said would be

John Doe. That is, the quarterback has a name attribute since its parent class, FootballPlayer, had a name attribute. We can also print out the pass_yards attribute, which is 0, just like we initialized it to be.

Now, say that instead of a quarterback, we had said that player1 was a RunningBack. We'd still have the name attribute since a running back is also a child of the FootballPlayer class, and FootballPlayer has a name attribute. However, if we tried to print off the pass_yards, we'd get an error. Pass_yards were defined only for the Quarterback. Creating different players is straightforward. As we see in this example, we can create player1 as a quarterback, and player2 as a running back, and we can set all of the values of the attributes appropriately.

Now, it's not just attributes that can be inherited. We can also inherit methods. We've augmented our classes here to include some methods. For the FootballPlayer class, we'll define a method, printPlayer, that prints out the name and team of the player. We'll also add some methods for the Quarterback and RunningBack classes to compute some statistics specific to those positions. For a quarterback, that'll be the completion rate and yards per attempt, and for the running back, that'll be the yards per rush.

We can go back to our two players that we defined earlier, and then we can call the methods for those players. Notice that we can call printPlayer for both player1 and player2. Because the method is defined in the parent, it's automatically inherited by the children. We can also call those statistics methods that are specific to each of the child classes. In the example we have here, we call printPlayer for the quarterback, then print out the completion rate and yards per attempt statistics. We then call printPlayer for the running back and print out the yards per rush statistics for that player.

Let me say a brief word about terminology. When people discuss inheritance, there are different terms used for the different classes. Sometimes, we call them parent and child classes, like I've been doing so far. Sometimes we call the parent class the base class, and we call the children derived classes. The idea here is that the parent

class is a base that's shared by everything, and then we derive—or pull out—an extension from the base class. Other times, we'll call the parent a superclass and the children will be subclasses. The idea here is that the parent class encompasses all the other classes which are specializations of it. So, whether you hear parent or base or superclass, or you hear child or derived or subclass, just know that they're referring to the same thing.

Now, just like in biology where you can inherit traits from more than one parent, classes can inherit properties from multiple parents. And you'll see some books with a lot of discussion about what's called multiple inheritance, so you might be tempted to try it. But let me give you a piece of advice. For the most part, you should stay away from multiple inheritance. It can make your code more confusing to follow since you have multiple parents that you could inherit different attributes from, leading to conflicts on which one wins out, plus it's very rare that multiple inheritance is actually the right solution to your problem. So it's good to know that multiple inheritance exists, but we're going to avoid situations where one class can have two or more parents.

The third main feature of object-oriented programming is polymorphism. Think about that word. Poly means many, and morph refers to shape, so polymorphism refers to things taking on many shapes. When we use the term polymorphism, what we mean is that a function or a method can take on many different forms depending on the context. It'll be easiest to see it in action.

Let's go back to our earlier example with the football players. Let's imagine that we want to assess whether each player is good or not, according to some measure we devise. Clearly, the way we determine whether a quarterback is good at throwing or not is going to be different from the way we determine whether a running back is good at running or not.

So, say we'd like to have a method called isGood that returns true or false, depending on whether a player is good or not according to whatever method we devise. We can augment our earlier definitions to

include this function. Let's put a function isGood in the FootballPlayer class. Now, it's not possible to determine whether a generic football player is good or not, given that all we have is a name and team. So in this case, we'll print out some sort of error message saying that we called a function that wasn't defined.

Now, let's see how we can use this. Let's assume we have two players that we've created, just like before. We're now going to create a playerlist, and we'll add both player1 and player2 into that list. Then, we'll go through each of the players in the list, using a for statement. For each player, we'll print out the player info using the printPlayer method, and then we'll call isGood and print out whether the player is a good player or not a good player. So what happens when we run this? Well, we get the error message printed out, right in the middle, and that should be what we'd expect.

Now, an error is not what we want. We want to be able to actually make comparisons so we'll define isGood as a function in each of the children classes. Here's what that'll look like. In both of the child classes, we'll create a function isGood. For the quarterback, we return whether the yards per passing attempt are above some level. For the running back, we return whether the yards per rush are above some level. So how does this work if we run it now? Using the exact same code as we had before, we now get an output without the error messages. For one of our players, the quarterback, we find that the player is not a good player, while the other player, the running back, is a good player.

What's happened here is that when the Python compiler sees the call to isGood, it first looks at the definition of isGood in the child class. If there's not a definition of that method there, it'll look at the parent to see if the method is defined there.

Now, this is one example of how polymorphism works. When we went through that player list, we were just calling the isGood method on each player but didn't know how it was implemented. In effect, the isGood method was taking a different shape in the different objects—that is, it was polymorphic. This made it a whole lot simpler for us.

We didn't have to call separate loops to call isGoodQuarterback and isGoodRunningBack or anything like that. Instead, we could have a single method defined, and all the objects that inherited from that method would also have that function defined. I want to show you one more modification to the code we just saw so that you can see another example of how polymorphism can work. This is a way of forcing us to use polymorphism.

Let's say that instead of going through the list of players and having an if statement, we instead just called a method called printGood that checked whether the player was good or not and printed the appropriate message. This would mean that we'd want to add that printGood method into the base class FootballPlayer. The printGood method will check whether the isGood method returns true or false and print the appropriate message.

Now, take a close look at the FootballPlayer class. Do you see anything missing? Well, there's no isGood function defined in the FootballPlayer class anymore. I deleted that function to show a bit more about how polymorphism can work. The printGood method is still calling isGood, but isGood is not defined here in this base class. Instead, we have to go to one of the derived classes to actually resolve what the function is supposed to do. Notice that this is forcing us to implement isGood in each of the children classes. Since the printGood method is calling isGood, if the isGood command is not defined in the child, then printGood will crash. But if we run this code, it's going to turn out exactly like what we had before, since we actually did define isGood in each of the child classes.

One term you'll sometimes hear people use when they discuss polymorphism is a virtual function or an abstract interface. Both of those terms give the connotation of something that's not really there. It's a function that's not really defined or an interface that's abstract instead of concrete.

Now, inheritance is useful if you're defining your own set of classes like we've seen in these examples. But we can actually inherit from any other

class. Python even lets you treat basic types like strings or integers as a parent class. That's just confusing to me, though, so let's not talk about that. There is one place where inheritance from a standard type is especially useful, though, and that's exceptions. So let me spend just a minute about them.

First, let me remind you of what exceptions are. Exceptions are used to catch errors that would otherwise crash the program. For example, if we had a function to compute a division, we might catch a couple of types of error: a TypeError when the input values could not have a division defined, or a ZeroDivisionError when we try to divide by zero. If we encounter an error on our own, we can raise an exception. But even though there are a lot of exceptions defined, sometimes those exceptions don't really capture the nature of the problem.

We don't have an exception that describes this problem, so we're going to make our own exception, which will also demonstrate inheritance. This exception will be specific to our case, and we'll call it MissingChildMethodError. To create this new type of exception, we create a new class with that name, and list Exception as the parent, thus it inherits all the functionality that the exception class provides. I don't happen to have anything special to do with this exception, so I'll just define the body of the class as pass, meaning it's essentially identical to the generic Exception, but it has a different name. Now, in the isGood method, I can raise MissingChildMethodError instead. And when we run the code, indeed we see the MissingChildMethodError exception listed.

Even better, though, we can now catch that particular exception. If we set up a try-except statement in our main code like you see here, instead of the exception crashing the program, we can handle it. We can print our own nice message, and we can exit cleanly, or we could choose to do something else. The key is that we created our very own exception class, and that let us raise that exception and then deal with it nicely in a try-except block. So what do you think this code will do?

Well, we have a try-except block. The exception occurs in a call within the try statement, so the exception is handled by the except clause,

which prints out "Whoops—we forgot to define isGood," and the code exits cleanly instead of crashing. In fact, given that we caught the exception, we could have continued on processing other data, without crashing our program. So that demonstrates just one of the useful ways we can use inheritance to create our own exceptions.

Now that we've seen objects, I want to introduce two important Python modules that make objects much more usable. After all, it's one thing to just write out a number or string, but to write out an object, that's a little bit trickier. Both of the modules that I want to introduce are included in the standard library and can really help make it easy to handle objects. They're called JSON—J-S-O-N—and pickle.

Remember that for our basic routines for reading and writing files, we deal with strings. We write out strings, and we read in strings. As you might imagine, it can be kind of a pain to have to deal so much with strings, especially when we have data in other formats. The idea of taking that data and turning it into a string to save in a file, or reading in strings and pulling out the individual pieces one at a time, can sometimes be overwhelming.

When we have data that's not a string, we need to convert it into one. JSON, which stands for JavaScript Object Notation, is a way of structuring data in a text format. It uses a syntax for writing objects that's very similar to the way that objects are defined in Java and JavaScript. JSON data is a nice, human-readable string as opposed to binary data. JSON groups information in objects using curly braces, with each attribute written as an attribute name, followed by a colon, followed by the value. The value can be a new object itself, nested inside the previous one.

So JSON is perfectly capable of representing our objects, and because it's a text format, we can read and write JSON data easily. Plus, it's something that is independent of the language that it was produced in. For this reason, JSON is the most common way that data files are transmitted over the web.

Python's JSON module includes commands that let us convert data to and from JSON. Basically, the JSON routines let us convert a piece of data into a JSON string. Most, though not all, Python data types can be converted to JSON. The JSON string can be written to or read from a file like any other string.

To use the Python JSON routines, we need to start out by importing JSON at the beginning of our file. We can then create a JSON string by using the command "json.dumps," and then for the parameter in parentheses, we put the data that we want to convert to JSON format. That gives us a string that we can output to a file just like any other. We can add a newline character to separate it from the other strings.

Later, we could read in the file by reading in a line as a string. For basic data types, we can use the "json.loads" command, putting the string in parentheses. The result of that command will be a data element that we can assign to a variable.

Let's first see a simple example. Imagine that we had the following code. We have a length and width defined, as 20 and 15. We open up a file for writing. Then, we create JSON strings for the length, width, and some text, just saying that this is data for an example. Each of those will be written to the file. So this code is creating a file containing three JSON strings. Now, how would we go about reading that and using it?

Well, here's the code to read that file. We open up the same file for reading. Then we read a line, and we use the "json.loads" command to convert it from a JSON string to a data element. We read in the length, width, and description, and then when we print out the description as well as the product of length and width, we get the answers we would expect.

Now, to make the most use of JSON and really see its power, we'll want to use it to store lists and objects. The process for writing will be exactly the same. Typically, when using JSON, the entire file is just one JSON element, so you're just writing one giant JSON string. The big advantage, though, is that it handles all the conversions between the data types and a string for you.

To see how we can write objects, consider our football player objects that were defined earlier. We can write them out by converting each object to a JSON string, which we can then write out. To do this, we don't pass the object to "json.dumps," but rather the object-dot-underscore-underscore-dict-underscore-underscore. Dict will give a dictionary version of the elements of the object, and that's what will get converted to a JSON string. We'll learn more about dictionaries soon. If we look at the actual output file, we can see that basically every attribute has been printed, along with its value. Now, reading in a JSON string into an object is trickier. Basically, each of the attributes will have to be read out of that string and set in the object one by one. It's a little painful, but it can be done.

Now, besides JSON, there's another Python module that's very useful for working with more complex data and which makes it possible to trivially read objects in addition to writing them. Pickle is a module that lets you read and write data other than strings more easily. Pickle lets us read and write data from a file in binary format, which is how most files you encounter every day are actually stored, from images to word processing files. And it works for even more data types than JSON.

Pickle is a Python-specific format. If you write a file using pickle commands, it basically needs to be read by another Python program also using pickle commands. Pickle should not be used for writing data that you need to send to other people, and you should never read pickle-produced files from others unless you're absolutely sure of the source, since it's easy for them to contain malicious data. Let me show you how to use pickle commands to read and write numbers more easily.

We start by importing the pickle object. We also need to change the way we write and read files. When we open a file for writing, or for reading, or for appending, we will need to use not just *w, r* or *a*, but will also need to add a *b* at the end. So to open a file for writing, we'll put *wb* as the second string inside the parentheses following open. Likewise, to read, we would write *rb*, and to append we would write *ab*. The *b* indicates that this will be a binary file—that is, a machine-readable file in binary, not a text-oriented file like those we've had previously. This also

means that the pickled files that we will write and read are not ones that we'd be able to understand just by opening them in a word processor or something like we could do with JSON files.

Now, to write a value to the file, we'll write "pickle.dump," and then as parameters, we'll have two things. The first one is the data we're going to output. The data can be almost anything—a string, a number, a list, or an object. Following this, there's a comma, and then the name of the file that we're going to output to.

So, say we wanted to store data that we were keeping about a bank account. Maybe we want to store the account number, and the name of the account holder, and the balance, in that order, and we want it in a file called BankAccount.dat. How would we write that data out using pickle?

Well, first we'd have our "import pickle" command. We'd then open up the BankAccount.dat file for writing, being sure to put *wb* in at the end. Then we'd call the "pickle.dump" command, First, we would dump the account, then the owner, then the balance, each time specifying the file that we're dumping to. Finally, we close that file.

Now, what about reading from a file that was produced with pickle? Basically, we need to read the data out of that file in the same order that it went in. Here's how that works. When we open our file for reading, we have to note that it is *rb* instead of just *r*. The actual command to load the data in pickle is "pickle.load," and then in parentheses is the name of the file. That will pull out the next item that was written to the file. We need to store that item, so we have to assign the value to a variable.

Let's say that we wanted to read in the data for a bank account that we just saved. We can use the "pickle.load" command three times, pulling out the account, then the owner, then the balance. Notice that we're pulling out the data in the order that it was put in.

Let's look at how we'd output the football player objects from before. We can just take each of the objects and dump it straight to the file. That's all. Now, we couldn't actually look at this file like we did with the JSON file

since it's in binary. The real advantage of pickle is on the other side. If we then want to read those objects back in, we can just use the "pickle.load" command and we get an object back. Unlike JSON, this is not a string or dictionary that we need to process further. We actually get an object back. And we can take that object, and we can compute with it, calling methods and everything, just like we had before. You see here that we can call things like the printPlayer and isGood methods on the objects.

The nice thing about pickle is that we don't need to worry at all about the details of how that data is getting stored in the file—no conversion to and from strings, no worry about how it's being written, and so on. However, it's not a universal file format, so it's not great, for instance, if you're going to produce data that people are going to download off of a web page. The place to use it is in your own programs, for your own use. For example, it's great to use pickle if you have a program that you need to be able to save and later load.

Traditionally, object-oriented programming is based on encapsulation, inheritance, and polymorphism—in that order. Encapsulation is easily the most important idea of the three since it helps you organize your code so that you don't have to worry about the details inside of an object, only the methods that are used to interact with it.

Inheritance lets you duplicate an encapsulation and pass it on to the next generation. In the early days of object-oriented programming, proponents claimed that inheritance would eliminate the need to rewrite code for new programs. That particular payoff from using inheritance didn't really pan out quite as much as originally hoped, but what object-oriented design does do is it makes it easier on the programmer to understand and revise their own program, and that makes it possible—whatever your own level—to build more complex programs than would otherwise be possible.

And polymorphism, in practice, is always used with inheritance. Polymorphism lets us treat lots of different types of objects the same way. We can write one set of code, calling methods without worrying about the particular way that that method is implemented.

In Python, object-oriented programming is more of an option rather than a requirement. But objects are such a useful way of designing code that objects and classes are pretty pervasive, especially in Python modules. The tkinter module for creating user interfaces relies heavily on inheritance and polymorphism, for instance.

The key in object-oriented design is to consider what the units of description are, and then turn these into classes—checking accounts, for instance, or vehicles, or people. You want to think about conceptual ideas that you can package together nicely in one class. What things are there, and then what operations have to be done to those things is the heart of object-oriented design. Once you've determined the things, and what needs to happen to them, you're on the way to designing a class and implementing it.

In the next lecture, we'll use classes again, no longer just for specific applications such as bank accounts, but for much more general ways of organizing data, known as data structures. I'll see you then.

19

# Data Structures: Stack, Queue, Dictionary, Set

An orderly and systematic method of organizing data makes it much easier to actually use that data. Our code can access the data more easily to find the particular part of the data desired, and this lets us create more efficient programs. The term we use in computer science to describe these ways of organizing data is "data structures." As you will learn in this lecture, structuring our data can make it possible to do things that we never could if it's unorganized.

## [ DATA STRUCTURES ]

- Classes and objects are great at tying together different types of data, but object-oriented design is focused more on bundling different types of data together. Data structures are focused on how to organize large amounts of the same type of data.

- One of the simplest data structures is what Python calls a "list," and other languages call an "array," which has an order—is linear—and lets us string many distinct things together in sequence (so it's sequential).

- But stringing things together is not the only way we could organize them. We could lay them out in a grid, for example. Either a list that's sorted, or a heap, would make it much easier to get the largest (or, alternatively, the smallest) value.

- Data structures can also be nonlinear and nonsequential. Maybe the data would be better organized around memberships, or geographic location, or a bunch of special-purpose keys associated with each object.

› There are many methods for organizing large amounts of data. For an army, organizing into a hierarchical structure might be great for helping make sure orders get followed. But that might not be a great way of organizing if the goal were to come up with creative ideas. In other words, organization affects operations.

## [ STACKS ]

› Imagine that we have a stack of books. Let's assume that they are heavy books, such that we can only hold one at a time. If we have a stack of these books, there are basically just two things we can do: add a book to the stack or take the top book off of the stack.

› The **stack** data structure is basically just this, only with data instead of books. If we add something new onto the stack, we'll call the operation a "push"; if we remove the top item from the stack, we'll call it a "pop."

› Let's see how this would work with a list and some of the commands already available for lists. First, just to extend the book analogy, let's assume that we've organized our book data into a book class, where we store a title and author per book. We also create three specific books: a long book, medium book, and short book.

```
class book:
    title = ""
    author = ""
long_book = Book()
long_book.title = "War and Peace"
long_book.author = "Tolstoy"
medium_book = Book()
medium_book.title = "Book of Armaments"
medium_book.author = "Maynard"
short_book = Book()
short_book.title = "Vegetables I Like"
short_book.author = "John Keyser"
```

- Our stack of books is going to be represented using a list. The first book in the list is the book on the bottom of the stack, and the last book in the list is the top book on the stack. So, to push a book onto the book stack, we would just use the "append" command on the stack of books.

- We start out with an empty list, which means that we have an empty stack. We then stack the books on top of each other. Let's say that we want to put the medium book down first. We'll append the medium book to the list. Next, we might want to stack the short book, so we append it. Finally, we stack on the long book.

```
book_stack = []
book_stack.append(medium_book)
book_stack.append(short_book)
book_stack.append(long_book)
```

- In memory, this set of books is treated as a list, with the medium, short, and long books listed in order. But we are supposed to think of it conceptually as a stack, with the medium book at the bottom, then the short, and then the long book.

- Now let's say that we want to pop the top book off of the stack. Lists have a built-in method named "pop," which will remove the last item from a list and return it. In this example, we assign the result of the pop to a variable "next_book," which now refers to the long book, because that was the first one on the stack. If we were to print the title and author of next_book, we would see the title and author of the long book. If we were to pop another book off the stack, the next one would be the short book.

```
book_stack = []
book_stack.append(medium_book)
book_stack.append(short_book)
book_stack.append(long_book)
next_book = book_stack.pop()
print(next_book.title+" by "+next_book.author)
```

```
OUTPUT:
War and Peace by Tolstoy
```

- Stacks give us what's referred to as "last in, first out"—that is, the last thing pushed is the first thing popped.

- Inside computers, stacks have a very fundamental use. As we make function calls, the computer memory is storing data in what's referred to as the **call stack**, also known as a "control stack" or "runtime stack" or "frame stack," which consists of function activation records, which keep track of all the variables and data defined in that part of the program.

## [ QUEUE ]

- What if we want a "first in, first out" process? This is what you encounter when people queue up to stand in line—the first one in line is the first one handled.

- We can implement a queue with a list, very similarly to how we implemented a stack. The order of data in the list is the same as in the queue. With a queue, just like with a stack, we can push new objects onto the end of the list using the "append" command. However, instead of popping from the end of the list, we instead need to take off the element at the front of the list.

- Python makes this really easy. The "pop" command can take a parameter, indicating which element gets taken out of the list. If no parameter is given, it defaults to the final element, as we saw with stacks. But for the first element in the list, we just pass in a 0, and the first element is removed.

- Let's look at some code to get the idea. Instead of a stack of books that we are piling up, we have a queue of books. Maybe we buy books one at a time and want to read them in the order we bought them, for example.

- We can build our queue like we built the stack. We start with an empty list we call "book_queue." Then, we add books to book_queue by calling the append methods. This creates the exact same list as we had in the earlier case. The only difference is in how we think about it—as a queue, versus a stack.

- Just like we could pull one item off of the stack, we can also pull one item off of a queue, by calling the pop method on book_queue. Notice the parameter 0 when we call pop. So, this call to pop would pull off the next book in the queue. When we finished that book, we could call pop with parameter 0 again to get the next book in the queue.

```
book_queue = []
book_queue.append(medium_book)
book_queue.append(short_book)
book_queue.append(long_book)
next_book = book_queue.pop(0)
```

## [ HASH TABLES ]

- Another really useful data structure is called a **hash table**, which works by mapping large data values into a smaller set of indices.

- To see how helpful it can be to map like this, imagine that you have a set of friends and they all have phone numbers, but all of them have blocked caller ID. So, when they call, you don't know whose phone number belongs to whom. You'd like to be able to enter a phone number and find who it is.

- It would not be a good idea to create a giant list, where the list element number corresponded to the phone number. So, if Sue Smith had phone number (135) 246-0987, then element number 1352460987 would have the value "Sue Smith." The problem is that most of the list is going to be empty.

› In fact, there will be 10 billion possible phone numbers, so you'd need a list with 10 billion elements. Even if we could store that whole list, maybe you have about 100 friends, so only 100 of those list values will even be filled in.

| | |
|---|---|
| 0000000000 | |
| 0000000001 | |
| 0000000002 | |
| ... | |
| 1352460987 | Sue Smith |
| ... | |
| 8647531234 | John James |
| ... | |
| 9999999999 | |

› Here's where a hash table comes in. Instead of a list of 10 billion elements, let's instead take a list of just 100 elements. We'll store each person in a slot that corresponds to just the last two digits of the phone number. So, Sue Smith, because her phone number ends in 87, would go into the list at index 87.

| | |
|---|---|
| 00 | |
| 01 | |
| 02 | |
| ... | |
| 34 | John James (8647531234) |
| ... | |
| 87 | Sue Smith (1352460987) |
| ... | |
| 99 | |

› This is a much more compact representation than the first list. But what if two people have the same last two digits to their phone number?

› Imagine that Bill Brown comes along with the phone number (808) 424-1287. He'll end up in the same position as Sue Smith. To resolve this, we can use **chaining**—just making a list of everyone in that slot. So, when we get to a slot, we can't just pull out the name; instead, we have to look at everyone in that list. But it's still much more practical than the giant list.

| | |
|---|---|
| 00 | |
| 01 | |
| 02 | |
| ... | |
| 34 | John James (8647531234) |
| ... | |
| 87 | Sue Smith (1352460987),<br>Bill Brown (8084241287) |
| ... | |
| 99 | |

› Imagine that instead of something simple like a phone number, we had some other way of identifying people. For example, maybe each of your friends has a nickname that he or she uses online, and you want to be able to look people up by that nickname. But we need some way of converting the nickname into a number. A function to convert some particular key phrase into a number that can be used to index into an array is called a **hash function**.

› Hash tables can get much more complicated than this, but fortunately, there's a tool in Python that basically implements hash tables for us, and we don't even have to come up with our own hash function. In Python, this tool is called **dictionary**, and the Python command to create a new dictionary is called "dict."

› Compared to the alphabetical list of a traditional dictionary in a book, a hash table is designed to be more efficient. And there are other names for hash tables, including “map,” “symbol table,” and “associative array.”

## [ SETS ]

› A different way that hash tables can be used is accessible with another built-in Python data structure: the **set**. A set is just a collection of items, but it will be stored using a hash table instead of a list so that it does mathematical set operations and checks set membership very quickly.

› In the following, we’ve created a set of people, and we initialize it with three people’s names. Notice that we have the elements of the set inside the curly braces, separated by commas. This code also shows that we can use the “in” statement to check whether or not a particular item is in the set or not. In this case, we check for the string “John.” Because that string was part of the set, this code will print out “Yes!”

```
people = {'John', 'Sue', 'Bill'}
if 'John' in people:
    print("Yes!")
else:
    print("No!")

OUTPUT:
Yes!
```

› Note that we could get the exact same effect by using the “set” command, as follows, instead of the curly braces. The set command takes in a list as a parameter, and all the elements of the list get put into the set. One big advantage of the set command over curly braces is that the set command lets us create an empty set. If we just have curly braces, with nothing inside, that will create an empty dictionary, not an empty set. So, if we want to start with an empty set and gradually add things to it, we have to use the set command.

```
people = set(['John', 'Sue', 'Bill'])
if 'John' in people:
    print("Yes!")
else:
    print("No!")

OUTPUT:
Yes!
```

- Sets have some additional operations defined on them that can be very useful. First, sets have an "add" method defined. To add a new element to a set, just call "add" as a method on that set and pass in the new element as a parameter. In the following example, we have a set of friends from work, and we add "Kathy" to that list. You can see from the output that Kathy is added to the set. Likewise, there is a "remove" method defined. In the example, we remove "Fred" from the list.

```
work_friends = {'Sue', 'Eric', 'Fred'}
print(work_friends)
work_friends.add('Kathy')
print(work_friends)
work_friends.remove('Fred')
print(work_friends)

OUTPUT:
{'Fred', 'Eric', 'Sue'}
{'Fred', 'Eric', 'Sue', 'Kathy'}
{'Eric', 'Sue', 'Kathy'}
```

## Readings

Gries, *Practical Programming*, chap. 11.

Lambert, *Fundamentals of Python*, chaps. 7–8 and 11.

## Exercises

1 Assume that the “Stack” class is defined as in the lecture. What would be the output of the following code?

```
namestack = Stack()
namestack.push("John")
namestack.push("James")
namestack.push("Joseph")
person = namestack.pop()
print(person)
person = namestack.pop()
print(person)
person = namestack.pop()
print(person)
```

2 Assume that the “Queue” class is defined as in the lecture. What would be the output of the following code?

```
namequeue = Queue()
namequeue.enqueue("John")
namequeue.enqueue("James")
namequeue.enqueue("Joseph")
person = namequeue.dequeue()
print(person)
person = namequeue.dequeue()
print(person)
person = namequeue.dequeue()
print(person)
```

**3** What would be the output of the following code?

```
cast = {"Cardinal Ximenez" : "Michael Palin", "Cardinal
   Biggles" : "Terry Jones", "Cardinal Fang" : "Terry
   Gilliam"}
cast["customer"] = "John Cleese"
cast["shopkeeper"] = "Michael Palin"
print(cast["shopkeeper"])
print(cast["Cardinal Ximenez"])
print(cast["Cardinal Fang"])
```

**4** What would be the output of the following code?

```
primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}
teens = set([13, 14, 15, 16, 17, 18, 19])
print(primes - teens)
print(primes & teens)
print(primes | teens)
print(primes ^ teens)
```

19 TRANSCRIPT

# Data Structures: Stack, Queue, Dictionary, Set

An orderly and systematic method of organizing data makes it much easier to actually use that data. Our code can access the data more easily to find the particular part of the data desired, and this lets us create more efficient programs. The term we use in Computer Science to describe these ways of organizing data is data structures. Structuring our data can make it possible to do things that we never could if it's unorganized. Classes and objects are great at tying together different types of data, but object-oriented design is focused more on bundling different types of data together. Data structures are focused on how to organize large amounts of the same type of data.

One of the simplest of data structures is what Python calls a list, and other languages call an array. A list or array has an order, it's linear, and it lets us string together many distinct things together in a sequence, so it's sequential. But stringing things together is not the only way we could organize them. We could lay them out in a grid. If we had a list that's sorted, or a heap, that would make it much easier to get the largest or the smallest value. Data structures can also be non-linear, non-sequential. Maybe the data would be better organized around memberships, or geographic location, or a bunch of special-purpose keys associated with each object.

There are many, many methods for organizing large amounts of data. For an army, organizing into a hierarchical structure might be great for helping make sure that orders get followed. But that might not be the greatest way of organizing if the goal were to come up with creative ideas. To state it simply, organization affects operations. We're going to start with two of the most basic data structures—stacks and queues—which can be implemented using lists. Then we'll move to a couple of

non-linear data structures—dictionaries and sets—both of which can be implemented using a hash table.

OK, imagine we have a stack of books. Let's assume that these are heavy books so that I can only hold one at a time. If I have a stack of these books, there are basically just two things I can do: I can either add a book to the stack, or I can take the top book off of the stack. The stack data structure is basically just this only, with data instead of books. If we add something new onto the stack, we'll call the operation a push. If we remove the top item from the stack, we call it a pop. That's all we want to do with a stack—push or pop.

Let's see how this would work with a list and some of the commands already available for lists. First, just to extend the book analogy, let's assume that we've organized our book data into a book class, where we store a title and author per book. We also create three specific books, which I've called a long book, medium book, and short book.

Our stack of books is going to be represented using a list. The first book in the list is the book on the bottom of the stack, and the last book in the list is the top book on the stack. So, to push a book onto the book stack, we would just use the append command on the stack of books, like this: We start out with an empty list which means we have an empty stack. We then stack the books on top of each other. Let's say we want to put the medium book down first. We'll append the medium book to the list. Next, we might want to stack the short book, so we append it. And finally, we stack on the long book. Now, in memory, this set of books is treated as a list, with the medium, short, and long books listed in order. But we're supposed to think of it conceptually as a stack, with the medium book at the bottom, then the short, then the long book.

OK, so now let's say that we want to pop the top book off of the stack. Well, it just so happens that lists have a built in method named pop. Isn't that convenient? Pop will remove the last item from a list and return it. In our example, we assign the result of the pop to a variable next book, and so next book now refers to the long book since that was the first one on the stack. If we were to print the title and author of next book, we'd see

that the title and author are those of the long book. If we were to pop another book off of the stack, the next one would be the short book.

The key thing to keep in mind is to think of this as though we're using a stack. Even though the way we actually implement it is with a list, which we use for all sorts of other things, we want to think of it as though it's a stack and just use our push and pop operations. To help force ourselves to use the stack this way, we could even create a separate stack class that just has methods named push and pop so that we're kind of forced to obey the rules of a stack. Stacks give us what's referred to as last in, first out—that is, the last thing pushed in is the first thing popped out. An example of a stack in everyday life is when a workforce has layoffs that are based strictly on seniority—the last people hired are the first ones fired.

The card game Solitaire is a good example of a stack. In Solitaire, you draw cards from a deck, and as cards are drawn, they're placed onto a stack called the waste. We would say they get pushed onto the waste stack. Then during the player's turn, they always have the option of taking the current top card off of the waste stack to play it. We would say that the top card gets popped off of the waste stack.

Let's see how the code for a waste stack would work where we're dealing with a 3-card draw. I'm not going to show a whole Solitaire game here, but just the part dealing with the waste pile. We would first set up our waste pile as a stack. Then, while the player is playing, there are two possible moves that would require something to be done with that stack. One move would be to draw. In this case, three cards are drawn from the deck, and those cards are placed onto the waste pile. So we'll call the push method on the waste pile object three times, once for each card. The other move that involves the waste pile is when a player wants to use a card from the waste pile. In this case, we'll call the pop method on the waste pile object. This removes the top card from the stack and it returns it as current card. The program would then have to handle the card getting played wherever it goes.

Inside computers, stacks have a very fundamental use. As we make function calls, the computer memory is storing data in what's referred to as the call stack. The call stack—also known as a control stack, or a runtime stack, or a frame stack—consists of function activation records, where the function activation record keeps track of all the variables and data defined in that part of the program.

To illustrate, consider a program with functions *a*, *b*, and *c*, where *c* calls *b* and *b* calls *a*. Somewhere in the main code, we have a call to function *c*. When we first start out the main program, our call stack just consists of the main program. When we call *c*, the function activation record for *c* is created and pushed onto the call stack. When it calls *b*, we get a function activation record for *b* pushed on. And when *b* calls *a*, then *a*'s function activation record is pushed on. When we're finished with *a*, its record is popped off the stack. Then, *b* finishes, and its record is popped off. And finally, *c* is finished and its record is popped off, leaving us back in the context of the main program.

Now, what if we want the opposite of a stack, where we want a first in, first out process? This is what you encounter when people queue up to stand in line—the first one in line is the first one handled. A streaming video service may let you create a personal queue of what to watch next. We can implement a queue with a list, very similar to how we implemented a stack. The order of data in the list is the same as in the queue. With a queue, just like with a stack, we can push new objects on to the end of the list using the append command. However, instead of popping from the end of the list, we instead need to take off the element at the front of the list. Python makes this really easy. The pop command can take a parameter, indicating which element gets taken out of the list. If no parameter is given, it defaults to the final element, as we saw with stacks. But for the first element in the list, we just pass in a zero and the first element is removed.

Let's look at one small piece of code just to get the idea. Now, instead of a stack of books that we're piling up, we have a queue of books. Maybe we buy books one at a time and we want to read them in the order that we bought them. We can build our queue like we built the stack. We start

with an empty list that we call book queue. Then, we add books to book queue by calling the append methods. This creates the exact same list that we had in the earlier case. The only difference is in how we think about it—are we thinking about it as a stack or as a queue?

Well, just like we could pull one item off of the stack, we can also pull one item off of a queue by calling the pop method on book queue. Notice the parameter zero when we call pop. So this call to pop would pull off the next book in the queue. When we finished that book, we could call pop with parameter zero again to get the next book in the queue.

Just like with stacks, we could create a separate class for queues if we want. That'll help force us to treat the list like a queue. The typical terms we use in programming to describe adding on to a queue or taking the first element are enqueue and dequeue. So our queue class can be defined with an attribute that's just a list that begins empty, along with an enqueue operation that appends to the list, and a dequeue method that just pops off the first element in the list. I've also added a method to tell me if the queue is empty or not. It'll just check the length of the list, and if it's zero, it'll return true.

Suppose we have a business where we sell shrubberies. The business has a list of orders that we want to process on a first-come, first-served basis for as long as the inventory holds out. Here's how a program to implement this might look. We have our queue class that I just discussed, and we'll also have a class defined to hold an order. An order will have a customer name and an amount, which is the number of shrubberies ordered.

The order class is pretty straightforward. In our main code, we'll start out by creating a variable, orders—that will be our queue of orders. There are lots of ways we could generate orders, but for this example, I'm just generating 20 random orders. The for loop will go over 20, and will generate an order for a random amount from 1–200. The customer name will just be Customer followed by the order number. From this, we create a new order, and we add that to our queue by calling the enqueue method on orders.

At this point, we have our queue of orders. We'll then set an inventory amount—in this case, that's 1000 shrubberies—and we'll start processing the queue as long as it's not empty. The first thing we do is get the next order from the queue—that's the dequeue method called on orders. We check to see if we have enough inventory to fill that order. If so, we print a message to go ahead, and fill it and we deduct that amount from inventory. If we don't have enough inventory, we give a message to notify the customer that it won't be filled. Now, if we look at a run of this code, we see that we fill several orders for customers 0 through 8. We don't have enough to fill the orders of the next few, but later we run across a few people who have small orders that we can still fill.

Another place that queues come up is in what's called a buffer for handling software events, such as a mouse movement or a keyboard key being pressed. This buffer is just a queue of the various events. That way, if lots of events happen quickly and they come in faster than they can be handled, you don't lose your data. If you've ever typed something in on a keyboard and you've seen it take a second before it appears on the screen, you've probably seen the hidden effects of a queue somewhere in the system.

Another really useful data structure is called a hash table, and it works by mapping a large number of data values into a smaller set of indices. To see how helpful it can be to map like this, imagine you have a set of friends and they all have phone numbers, but all of them have blocked caller ID so when they call, you don't know whose phone number belongs to who. You'd like to be able to enter a phone number and find out who it is.

Something that would not be a good idea would be to create a giant list where the list element number corresponds to the phone number. So if Sue Smith had phone number 135-246-0987, then element number 1352460987 would have the value Sue Smith. Now, I hope you can see the problem here. Most of that list is going to be empty. In fact, there will be 10 billion possible phone numbers, so you'd need a list with 10 billion elements. That's crazy. Even if we could store that whole list, maybe you have about 100 friends, so only 100 of those list values will even

be filled in. Now, here's where a hash table comes in. Instead of a list of 10 billion elements, let's instead take a list of just 100 elements. And we'll store each person in a slot that corresponds to just the last 2 digits of the phone number. So Sue Smith, since her phone number ends in 8 7, would go into the list at index 87. And John James, whose phone number ends in 3 4, would go into the list at index 34.

There are two things that I hope you're thinking—actually make that three—the first one being "I can't believe how cool this stuff is." But more to the point, the first thing I hope you notice is that this is a much, much more compact representation than that first list was. And the next thing that you might have thought was "What if two people have the same last two digits to their phone number?" Imagine that Bill Brown comes along with a phone number like 108-424-1287. He'll end up in position 87—the same position as Sue Smith. To resolve this, we can use chaining—just making a list of everyone in that slot. So when we get to a slot, we can't just pull out the name, we have to look at everyone in that list. But it's still a whole lot more practical than that giant list was.

Now, imagine that instead of something simple like a phone number, we had some other way of identifying people. For instance, maybe each of your friends has a nickname that they use online and you want to be able to look people up by that nickname. Nicknames don't usually have digits you can just pull out to find the index like a phone number did. Instead, we need some way of converting the nickname into a number.

A function to convert some particular key phrase into a number that can be used to index into an array is called a hash function. Let's just say that there's some hash function that when we apply it to the nicknames of your friends will give a value back between 0 and 99. We can put that friend in the slot corresponding to that number. For instance, if Sue Smith has the nickname Superstar and the hash function converts Superstar to the number 71, then we'll put Sue in the 71$^{st}$ element of the hash table. Later on, if we want to find who has the nickname Superstar, we convert that nickname to a number, 71, and then we look in slot 71 to see who's there. If there's a chain, we look at everyone in the chain to find the match.

Hash tables can get a lot more complicated than this. But fortunately, there's a tool in Python that basically implements hash tables for us, and we don't even have to come up with our own hash function. In Python, this tool is called Dictionary, and the Python command to create a new dictionary is called dict. Now, compared to the alphabetical list of a traditional dictionary in a book, a hash table is designed to be more efficient. And there are other names for hash tables, including map, symbol table, and associative array.

To create a dictionary in Python, we use curly braces. When we write "my_dictionary = {}," we're creating a hash table or a dictionary with no elements—with nothing inside of it. We can also set elements of the dictionary at the time we define it. So, if we want a dictionary that lets us store nicknames, we can write it as "nicknames = {}," and then some data inside the curly braces. That data inside of the curly braces gives the nickname and the full name of each person. We just give the nickname, followed by a colon, followed by the real name. And we use commas to separate however many initial data points we want.

We can access an element of the nickname list with a format very similar to the way we find an element of a list. We take the name of the dictionary, which in this case is nicknames, and then in square brackets we give the index. Instead of a numerical value like we had with lists, we'll have the nickname we're looking for as the index. The result is that the code you see here will identify the person who was entered into the dictionary with the given name.

We could also build up our dictionary in a completely different way. Instead of initializing the dictionary with data, we could begin with an empty dictionary and assign new elements to it one by one. The syntax here is, again, very similar to what we'd see in a list. We just put the nickname in the square brackets and assign the real name to that element.

Now, let's be clear about exactly how this is set up. Putting something into a dictionary requires two parts: a key and a value. The key can be any immutable data type, so it's okay to have a string, like we've seen in the nickname example, or an integer, or floating-point number, or even

a tuple. However, it's not okay to have a mutable data type, like a list or an object. The value, on the other hand, can be mutable or immutable.

Let me tell you about two more particularly useful things that you can do with a dictionary. Sometimes, we might want to delete an item. To do this, we write del, followed by the dictionary element that we want to delete—that is, the name of the dictionary, followed by brackets, with the key to delete inside of them. In the example we see here, the Superstar nickname is deleted from the dictionary. The second thing that's really useful is a way to check whether or not a key is in the dictionary. To do this, we write the key, then the word in, then the dictionary. And this will be true if that key is currently part of the dictionary, and false otherwise. So again, in the example, if we check for the nickname Cowboys Fan we'll get a true returned. And if we check for Superstar, we get a false returned, since this nickname was just deleted.

Just so you're aware, you can also iterate through all the elements of a dictionary using a for statement. Writing "for *x* in dictionary" will let *x* take on the values of all the keys in the dictionary. In the example here, I write "for nickname in nicknames." This means that in the loop, nickname will be the key, and we'll go through every key in the dictionary this way. And notice that when we print out the entries this way, they're not in the same order that they were put in. We put in Sue Smith, then Bill Brown, then John James. But when we print out all the keys, we find Sue Smith first, then John James, then Bill Brown. The reason is that underneath, there's a hash table, and the order is going to be based on the values that the keys were hashed to. That specific information is hidden from us, but we see the effect.

To illustrate a use for dictionaries, let me show you a short program that you could use to handle passwords. Our program is going to be a login part of a program. We'll ask for a username and password, and let someone through only if they have the right combination. Then, if they have three incorrect attempts, we leave the program. In our code here, we begin by setting up the passwords. In this case, our username will be our key, and the password will be the value. We initialize two variables—failed attempts, which will be used to count how many failed

login attempts there were; and verified, which will be used to note that someone has logged in correctly.

Our while loop will run as long as we're not yet verified. In the loop, we first get the user name and password. The key line is the next one—the if statement. We first check to see if the username that the person entered is even in our password dictionary. That's the first part of the if condition. Assuming that's true, the second part of the condition is checked, so we find out whether the password associated with that username is true or not. If the condition is true, the user entered a correct username and password combination, so we print a message and mark verified as true. The code will now leave the while loop and go on. On the other hand, if the user entered a bad username or the incorrect password, we print out a warning message and we increase the number of failed attempts. If this was our third failed attempt, we completely exit the program. Take some time to try this out for yourself and see how it runs in practice. Try setting up some different username/password combinations, and then run the code to make sure that it works

One quick warning: Although this general approach is indeed the way that passwords are handled, if we're writing code that will actually be used, we wouldn't list the actual passwords in the code like this. Instead, we would store encrypted passwords. That way, there's never a list anywhere of actual passwords that someone could grab and steal; there's only a list of encrypted passwords.

Now, a different way that hash tables can be used is accessible with another built-in Python data structure, and that's the set. You've learned about sets in math, and this is just a way of getting that same effect. A set is just going to be a collection of items, but it'll be stored using a hash table instead of a list. And this lets it do mathematical set operations and check for set membership really quickly. For instance, we might want to keep a list of the ingredients that are in a particular dish that we're fixing. We can create a set by using curly braces, just like we did with the dictionary. In contrast to a dictionary though, we won't have key-value pairs; but instead, we'll just have the individual items in the set. Let's see what this'll look like.

Here, we've created a set of people, and we initialize it with three people's names. Notice that we have the elements of the set inside the curly braces, separated by commas. This code also shows that we can use the in statement to check whether or not a particular item is in the set or not. In this case, we check for the string John. Since that string was part of the set, this code will print out "Yes!"

Now, note that we could get the exact same effect by using the set command—like you see here—instead of the curly braces. The set command takes in a list as a parameter, and all the elements of the list get put into the set. One big advantage of the set command over curly braces is that the set command will let us create an empty set. If we just have curly braces with nothing inside, that'll create an empty dictionary, not an empty set. So, if we want to start with an empty set and gradually add things to it, we have to use the set command.

Sets have some additional operations defined on them that can be really useful. First, sets have an add method defined. To add a new element to a set, just call add as a method on that set, and pass in the new element as a parameter. In the example you see here, we have a set of friends from work, and we add Kathy to that list. You can see from the output that Kathy is added to the set. Likewise, there's a remove method defined. In the example, we remove Fred from the list, and you can see he's gone. Sets support all the standard set operations, such union and intersection, but the notation is different from what you would see in a math class. Here, we have two sets of friends: one set from the neighborhood containing John, Sue, and Bill; and one set from work containing Sue, Eric, and Fred. Notice that Sue is in both sets.

We can find the people in one set but not the other using the minus symbol. So, when we say "neighborhood_friends - work_friends," we're left with just John and Bill; Sue was removed. We can find the union of two sets using a single vertical bar. In much of programming, the vertical bar means or, and that's how the union works—anyone in one set or the other is in the final set. In this case, we get the union, showing all five unique individuals. We can also get the intersection of two sets using a single ampersand for and, and that's how intersection works. The

intersection of the sets are those people in the first set and the second one. In this case, that's just Sue. And finally, to get the list of people in either set but not in both, we use the carat—the little arrow pointing up. This is what's called an exclusive or in logic terms, and it's the same as the union minus the intersection. In the example, this would give us everyone in the set except Sue.

Let's look at an example of how sets can be useful. Let's say that we're fixing a meal and we need to go out and buy ingredients. In the code, we'll have a recipe class that includes a name for the recipe and a list of ingredients needed. The initialization routine will take those items in as parameters. Then suppose our meal has three dishes—an omelet, bread, and cake. Each of those dishes has a list of ingredients, and we form a list of all those dishes.

Now, the next few lines are key. We start out with an empty set for a shopping list. We then go through each of the dishes in the list, and for each one, we form a set from the ingredients in the dish. Next, we form the shopping list by taking the existing shopping list and forming the union with the new set of ingredients. This will add any new ingredients to our list.

When this is finished, our shopping list consists of every ingredient we'll need for the meal. However, we also have several ingredients on-hand already. So, we take the set of these ingredients on-hand and perform a set difference to remove these ingredients from the shopping list. And finally, we print out the shopping list. If you look at the output, you'll see that we have a single list of all the items needed for any recipe, but that are not already on hand.

Data structures provide an organization that lets us perform operations more effectively. The Python list lets us implement stacks and queues very easily, all of which are linear or sequential data structures. Python also provides support for nonlinear, or non-sequential data structures such as dictionaries and sets—both of which are based on the hash table. And this is just the beginning of data structures. In future lectures, we'll see a couple more—graphs and trees—plus there are hundreds

of other data structures used for specific applications. The main thing I want you to take away is that by providing some structure to the data, we can speed up some of the operations on that data.

In the next lecture, we're going to look at how we can use algorithms to describe ways of working with these and other data structures to accomplish specific tasks. I'm really excited about this topic, since algorithms are what many people would argue is the heart of computer science. I'm looking forward to you joining me there.

# 20 Algorithms: Searching and Sorting

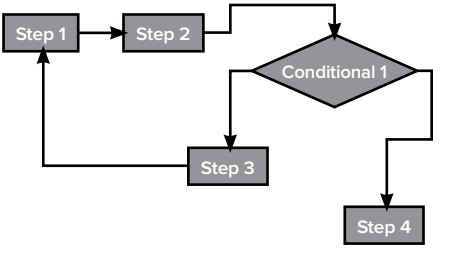
Algorithms form the core of computer science. Algorithms are how we describe what we actually want the computer to do. Computer programming is really just the process of taking an algorithm and converting it into a program that the computer understands. The program is the concrete incarnation of the more general algorithm. In this lecture, you will learn more about algorithms, including how they're described and how they're implemented in code. You will also learn a few of the most well-known algorithms for searching and sorting.

## [ ALGORITHMS ]

- Writing a program is essentially the same thing as writing an **algorithm**. The difference is that a program is a specific, concrete implementation in a particular programming language. An algorithm can be thought of as a more general description that's not necessarily tied directly to a particular programming language.

- We do not want to tie algorithms too tightly to any particular programming language, so we typically describe algorithms in a form that corresponds to some programming language but is not actual code in that language.

- One example of such a form is a **flowchart**, where we describe the algorithm graphically. We identify individual steps by shapes that contain text describing the step, with arrows showing how to move from one step to the next.



› We can also use **pseudocode** to describe the function of the algorithm. With pseudocode, we give an overview of the various steps of the algorithm and how they relate to each other. Pseudocode looks a lot like regular code in some languages, but many of the details can be eliminated along the way. Conversely, a single step in the algorithm might actually involve several lines of real code. If we design our algorithms well, it should always be straightforward to convert our algorithms into code.

```
1.  Step 1
2.  Step 2
3.  If (condition)
    a.  Step 3
    b.  Go to Step 1
4.  Else
    a.  Step 4
```

## [ SEARCHING ]

› To illustrate how algorithms work and how they're implemented, we're going to start with one of the simplest general algorithms: a **search**. We'll assume that we have some list and want to find whether a particular value is in the list or not. Keeping with the way algorithms are usually developed, we won't worry about exactly what we're searching for.

› Let's assume that we know nothing about this list—it's just a collection of values. We have a value that we're looking for, and we want to return either "True" or "False" in this case. Let's see how we might write some pseudocode for this algorithm.

› There are two pieces of data we need to run our algorithm. First, we need the list itself, which we'll designate as *L*. Next, we'll need the value we're looking for, which we'll designate as *v*. The output, the result of our algorithm, is going to be a Boolean value: either "True" if *v* is in *L* or "False" if it's not. If we're writing pseudocode, we want to be clear, at the beginning, about what's needed for input to the algorithm and what the resulting output will be.

› Next, we need to outline the steps to be taken. In this case, the idea is simple—we are just going to go through each element of the list, checking to see if there's a match. This is called a linear search: We're just going down the line, looking at each item, one at a time.

› If we find a match, we return "True," but if we get to the end of the list and haven't found anything, we return "False." We can write the pseudocode step by step. We first set a value, *i*, to be the first index in the list. Then, we go through a loop, as long as *i* is still within the range of the list. In each iteration, we compare the $i^{th}$ element with our value we are looking for, *v*. If it matches, we return "True" and are done. If not, we increment *i*. If we eventually reach the end of the loop, that means we never encountered the value *v* in the list, so we return "False."

```
Input:
    List of values: L
    Value to find: v
Output:
    True if v is in L, False otherwise

1.  Let i be the index of the first element in the list
2.  While i is less than the size of the list:
    a.  if element i of list L matches v, return True
    b.  otherwise, increment i
3.  return False
```

› It's pretty straightforward to put this algorithm into code. We can create a function that implements this algorithm almost exactly. We take in a list and a value as input. We initialize the index *i* to 0 and have a loop until *i* is no longer less than the length of the list. We have an if statement to compare element *i* of the list with our value, returning "True" if they match or incrementing *i* if they don't.

› The following example shows how we could use this. We have a list, "favorite_foods," and we call the function we just created on two values. When we look for a value that is in the list, we get a "True" back, and when we look for one not in the list, we get a "False."

```
def isIn(L, v):
    i = 0
    while (i<len(L)):
        if L[i] == v:
            return True
        else:
            i += 1
    return False
favorite_foods = ['pizza', 'barbeque', 'gumbo', 'chicken and
  dumplings', 'pecan pie', 'ice cream']
print(isIn(favorite_foods, 'gumbo'))
print(isIn(favorite_foods, 'coconut'))

OUTPUT:
True
False
```

- There is a built-in function within Python that implements this algorithm for us: the "in" command. So, when we ask whether some value is "in" some list, Python is doing exactly what we just showed in the background—it's just looping over all the elements to see if the one we want is there.

- Let's assume that instead of having a list of values in any order, our list was sorted, from smallest to largest. A command to sort the list would be to call a sort method on the list. But for purposes of writing an algorithm, we can just assume that the list has been sorted.

- There is a better way to write this routine—a way to make use of the fact that our input is sorted to search for the value more efficiently.

- If you had a dictionary and wanted to create a program to look up a word, going through every single word to see if it matches would be inefficient. A more efficient approach would be to start by checking some point in the middle of the dictionary. If that word was not the one you're looking for, then figure out whether it came before or after the word you wanted to find, and then look in the remaining half of the dictionary. You could continue this until you found the value, or else found that it was missing.

- How might we write the pseudocode for this? The input and output is the same as what we had before. We take in a list of values and a value to find and return "True" or "False." The only difference is that the list of values is given in sorted order.

- Now we need to describe the steps of our algorithm very precisely. Our approach will be to gradually narrow down the range of options until we find the one we're looking for. So, at any point, we will have a maximum and a minimum index of where the value might be. At the very beginning, our maximum and minimum indices will be from 0 to the list size minus 1. We'll check those values to make sure it's not matching them.

```
Input:
    List of values IN SORTED ORDER: L
    Value to find: v
Output:
    True if v is in L, False otherwise

1.  Set low = 0 and high = length of list - 1
2.  If L[low] == v or L[high] ==v, return True
```

- At this point, we know that the value we are looking for is somewhere between item "low" in the list and item "high" in the list. We are going to gradually narrow down low and high until either we find the point or there's nothing left between low and high.

```
Input:
    List of values IN SORTED ORDER: L
    Value to find: v
Output:
    True if v is in L, False otherwise

1.  Set low = 0 and high = length of list - 1
2.  If L[low] == v or L[high] ==v, return True
3.  While low < high-1       #Value is between L[low] and L[high]
```

- So, we are going to have a loop that continues as long as low is less than high minus 1. Notice that we want to continue only as long as the high and low indices are at least 2 apart so that there's some potential value in between. Once the high and low values are next to each other, we can quit the search, because there is no possibility of another value in between them.

- Notice that at each iteration of the loop, we still have that same condition: The value we're looking for is either between element low and element high, or it's not in the list. This condition—this thing that's the same every time we go through a loop—is called a **loop invariant**. When we're designing an algorithm, it often is helpful if we can identify such a loop invariant.

- Now we need to decide what's done in each iteration of the loop itself. We will compute a midpoint that's halfway between the high and low point and check to see if it matches the value. Notice that when we calculate the midpoint between low and high, we can do so by finding the difference between low and high, dividing it by 2, and adding it to the low. Notice that because we're dividing by 2, we could end up with a fractional value, which doesn't work for indices. So, we need to make sure that we are doing an integer division—keeping only the quotient but ignoring the remainder.

```
Input:
    List of values IN SORTED ORDER: L
    Value to find: v
Output:
    True if v is in L, False otherwise

1. Set low = 0 and high = length of list - 1
2. If L[low] == v or L[high] ==v, return True
3. While low < high-1        #Value is between L[low] and L[high]
    a. midpoint = low + (high-low)/2            #Integer division
    b. If L[midpoint] == v, return True
```

- Notice that because we know the high and low values are at least 2 apart from each other, high minus low divided by 2 is at least 1. So, the midpoint is guaranteed not to be the same as low or the same as high—it will be a new index somewhere between low and high.

- If the midpoint turns out not to be the actual value, we can at least use the midpoint to narrow our range. We're faced with one of two possibilities: either go forward with the range from low to midpoint or the range from midpoint to high. We can decide which of these is the right sub-range to continue with by comparing the value at the midpoint to the value we're searching for. If the value at the midpoint is less than *v*, it means that we need to use the upper sub-range. So, we can set low to be the midpoint. Going forward, we'll be looking between that midpoint and the high index. On the other hand, if the value at the midpoint is greater than *v*, it means that we should use the lower sub-range. So, we can set high to be the midpoint.

```
Input:
    List of values IN SORTED ORDER: L
    Value to find: v
Output:
    True if v is in L, False otherwise

1.  Set low = 0 and high = length of list - 1
2.  If L[low] == v or L[high] ==v, return True
3.  While low < high-1       #Value is between L[low] and L[high]
    a.  midpoint = low + (high-low)/2           #Integer division
    b.  If L[midpoint] == v, return True
    c.  If L[midpoint] < v, set low = midpoint
    d.  else set high = midpoint
```

- Notice that our loop invariant is maintained. The value we're looking for is either between low and high or it's not in the list at all.

- Finally, if we finish the loop, it means that we narrowed in, and the value we were searching for was not found. In this case, we'll return "False."

```
Input:
    List of values IN SORTED ORDER: L
    Value to find: v
Output:
    True if v is in L, False otherwise

1.  Set low = 0 and high = length of list - 1
2.  If L[low] == v or L[high] ==v, return True
3.  While low < high-1       #Value is between L[low] and L[high]
    a.  midpoint = low + (high-low)/2           #Integer division
    b.  If L[midpoint] == v, return True
    c.  If L[midpoint] < v, set low = midpoint
    d.  else set high = midpoint
4.  return False
```

› This is our algorithm description, and we've described it using pseudocode. The term for this type of search is a binary search, where at each iteration, we're reducing the search range by a factor of 2. This is much more efficient than the linear search, where we just looked at one item at a time, one after the other.

› Given an algorithm description, it's pretty straightforward to convert this to Python code. Notice that when we compute the midpoint, we are using integer division—the double slash rather than the single slash—to make sure that we get the integer quotient without any remainder.

```
def binaryIn(L, v):
    low = 0
    high = len(L)-1
    if L[low] == v or L[high] == v:
        return True
    while low < (high-1):
        midpoint = low + (high-low) // 2
        if L[midpoint] == v:
            return True
```

```
        elif L[midpoint] < v:
            low = midpoint
        else:
            high = midpoint
    return False
favorite_foods = ['barbeque', 'chicken and dumplings', 'gumbo',
  'ice cream', 'pecan pie', 'pizza']
print(binaryIn(favorite_foods, 'gumbo'))
print(binaryIn(favorite_foods, 'coconut'))

OUTPUT:
True
False
```

› When we run this code with a sorted list, we find that we correctly identify when an item is in the list or not. Realistically, we want to make sure that we tested this in a variety of situations.

## [ SORTING ]

› Linear and binary search are two of the simplest and most fundamental algorithms. Some slightly more complex algorithms are sorts. There are many different ways to sort, and different methods will work better or worse in different circumstances.

› What if we have a list of values that are in no particular order? If we want to find values in that list, we're stuck using a linear search. If we had a sorted list, though, we could use binary search and do the checks much faster. Often, if we are working with sorted data, our operations are much easier and simpler than if it's just a random collection of values. Sorting is a key tool.

› Let's compare two basic sorts: selection sort and insertion sort.

› Selection sort works as follows. We start with a mixed-up set of values that we want to put in order from smallest to largest. Because the first

thing we want is the smallest item, we look through all of our values and find the smallest one. We put that into the first place. We then repeat that process to find the next-smallest item and put that into the second place. Each time, we have to look at all of our remaining items so that we can select the one that is the smallest-remaining item. That's where we get the name "selection sort."

- Another decision that comes up when sorting is whether to move around the original elements of the list or make a copy. Lists are mutable data types, so you have the ability to reorder the elements themselves, if you want.

- If you directly sort the elements of the list, that's called an "in-place" sort. In contrast, an "out-of-place" sort means that you create a new list that's a copy of the original one. In this case, the original list stays unchanged, while we also have a new, sorted, list to work with.

- Choosing whether you want an in-place or out-of-place sort will depend on whether you need to maintain the original order for some reason. If so, you want an out-of-place sort. The more common case, though, is to just do the sort in place.

- In the selection sort, we can sort in place by making sure that every time we place a new element into its final position, we swap it with an existing element.

- Selection sort works, but it spends a lot of time going through the entire unsorted list on every iteration. Insertion sort is a different approach that can be much faster and simpler. For insertion sort, at iteration *n*, we've sorted the first *n* elements. So, the only thing to do on each iteration is add one more element into the right spot.

- With insertion sort, we start with the first item—and only the first item. When we take the second item, all we do is compare it with the first item and insert it in whichever position is correct. We continue making iterations in this way, where each time we take one more value and insert it into the list of sorted items. That's where we get the name "insertion sort."

› Sorting is such a common operation that Python has a built-in sorting function. For a list, we can call a sort method on the list by saying "sort()." This is an **in-place sort**. For an **out-of-place sort**, Python offers a more general command called "sorted()."

## Readings

Gries, *Practical Programming*, chaps. 12–13.

Lambert, *Fundamentals of Python*, chap. 3, p. 60–70.

Zelle, *Python Programming*, chap. 13.

## Exercises

We will show how to build another sorting routine: the bubble sort.

1 Write a function that takes in a list and an element number, *i*, and swaps element *i* with element *i*+1.

2 Write a routine, "one_bubble_pass," that implements the following pseudocode.

```
one_bubble_pass:
Input:
    List lst
Output:
    Modified list, True if a swap was made, False if not

1.  returnval = False
2.  Loop over all elements except last one in lst
3.  If lst[i] > lst[i+1] then swap elments i and i+1 and set
    returnval = True
4.  Return returnval
```

**3** The bubble sort just keeps calling "one_bubble_pass" until no more swaps can be made. Write a routine, "bubblesort," that implements the following pseudocode.

```
bubblesort:
Input:
    unsorted List
Output:
    sorted List

1.  Set flag to True
2.  While flag is True
3.  Set flag = one_bubble_pass
```

TRANSCRIPT 20

# Algorithms: Searching and Sorting

When we turn to algorithms, we're entering what's often regarded as the heart of computer science. The concept of algorithms comes to us from the Persian mathematician al-Khwarizmi, who wrote one of the most influential books of all time. Way back in the 9th century, his book systematically described precise procedures for solving equations, and in the process, really revolutionized mathematics. It formed the foundation for much of what we consider mathematics today.

The book was eventually translated into Latin and so its ideas spread from the Middle East throughout Europe. It was so influential that two words that we commonly use today come to us because this book introduced those ideas to the Western world. First, the title of the book contained the term al-jabr, referring to a method that al-Khwarizmi introduced for balancing equations. This word became our own familiar word algebra. The other term that we get from him is based on the author's own name. Al-Khwarizmi became the word algorithm, which is used to refer to a precise set of steps or rules to follow to accomplish some task.

These earliest algorithms offered some general techniques for solving basic arithmetic and algebra problems. If I asked you to tell me how to multiply two numbers, you could probably give me an algorithm—a general set of steps that you can use to multiply any two numbers together. School children usually learned one like this: First you take the ones digit from the first number and you multiply it by each digit of the other number, carrying the tens part over to the next column each time, and so on.

But the idea of laying out a precise set of steps or rules to accomplish a task doesn't apply only to arithmetic. If I asked you to tell me the process for making a sandwich, you could probably give me a precise set of

steps to follow involving a bottom slice of bread, some condiments, and so forth. You'd be giving me a sandwich-making algorithm.

As computers have been developed, the idea of algorithms has become more and more important. As I mentioned all the way back at the beginning of this course, computers are not really very smart at all and they can only follow very precise instructions given in very particular ways. So, to get computers to do all the wonderful things we think of them doing, we need to be able to formulate the bigger ideas that we have as these precise sets of instructions—that is, we need to be able to define algorithms.

Now, algorithms really do form the heart of computer science. When we use the term computer science, we're referring to a much bigger idea than just the programming that's been the main focus of this course. We're instead thinking about the broader ideas of how computation works and how we can organize that computation to achieve particular goals. Now, there are lots of different aspects to the study of computer science, but understanding algorithms is particularly important. Algorithms are how we describe what we actually want—in a more general way, what we want the computer to do. Computer programming is really just the process of taking an algorithm and converting it into a program that the computer understands. The program is the concrete incarnation of the more general algorithm.

Researchers in computer science often focus on coming up with new algorithms to solve various problems. The actual programming can become somewhat distinct, even secondary. There are different ways to program an algorithm—different languages to use, and different features of those languages. The key challenge, the important part, is coming up with the general steps to follow to achieve the goal. So for this lecture, we're going to focus on understanding more about algorithms—how they're described, how they're implemented in code, and we'll look at a couple of the most well-known algorithms for searching and sorting.

Let's first talk about what an algorithm really is. A particular analogy that's really useful is to compare an algorithm with a recipe. When

we have a recipe, we start with a list of ingredients. We then follow a particular set of steps, performing actions with those ingredients. At the end, we've produced some particular dish. In computer science, our ingredients are the data. The algorithm consists of the sequence of steps that we need to follow. Now, I've used this analogy before to describe programming, and there's a good reason—writing a program is essentially the same thing as writing an algorithm. The difference is that a program is a specific, concrete implementation in a particular programming language. An algorithm can be thought of as a more general description that's not necessarily tied directly to a particular programming language.

Now, you can think of this as writing a recipe where you know the general category of the ingredient—like it's a liquid, or a granular material, or a solid—without needing to know what particular substance it actually is. For example, the basic procedure for making vinaigrette is fundamentally the same, regardless of which specific oil and vinegar are used.

And this is how we typically describe algorithms. We do not want to tie them too tightly to any particular programming language, so we usually describe algorithms in a form that corresponds to some programming language, but it's not actual code in that language. One example of such a form is a flowchart, where we describe the algorithm graphically. We identify individual steps by shapes that contain text describing the step, with arrows showing how to move from one step to the next. Conditionals will have two arrows indicating what to do if the condition is true or false.

Another option is to use pseudocode to describe the function of the algorithm. With pseudocode, we give an overview of the various steps of the algorithm and how they relate to each other. Pseudocode looks a lot like regular code in some language, but many of the details can be eliminated along the way. Conversely, a single step in the algorithm might actually involve several lines of real code.

Now, we've actually been using this idea in the course whenever I've given you an overview of a function before actually showing the code to implement that function. What I've been doing in those cases is describing an algorithm before showing how that algorithm got turned into code. If we design our algorithms well, it should always be straightforward to convert our algorithms into code.

All right, to illustrate how algorithms work and how they're implemented, we're going to start with one of the simplest general algorithms—a search. We'll assume that we have some list and we want to find whether a particular value is in the list or not. Keeping with the way algorithms are usually developed, we won't worry about exactly what we're searching for. So, let's assume that we know nothing about this list; it's just a collection of values. We have a value that we're looking for, and we want to return either true or false in this case. Let's see how we might write some pseudocode for this algorithm.

There are going to be two ingredients—two pieces of data—that we need in order to run our algorithm. First, we need the list itself, which we'll designate as *L*. Then, we'll need the value we're looking for, which we'll designate *v*. Now, the output, the result of our algorithm, is going to be a Boolean value—either true if *v* is in *L*, or False if it's not. If we're writing pseudocode, we want to be clear, at the beginning, about what's needed for input to the algorithm and what the resulting output will be. Next, we need to outline the steps to be taken. In this case, the idea is simple—we're just going to go through each element of the list, checking to see if there's a match. This is called a linear search. We're just going down the line, looking at each item, one at a time.

If we find a match, we return true; but if we get to the end of the list and we haven't found anything, we return false. We can write the pseudocode out step by step. We first set a value, *i* to be the first index in the list. Then we go through a loop as long as *i* is still within the range of the list. In each iteration, we compare the *i*-th element with our value that we're looking for, *v*. If it matches, we return true and are done. And if not, we increment *i*. If we eventually reach the end of the loop, that means we never encountered the value *v* in the list, so we return false.

Now, it's pretty straightforward to put this algorithm into code. We can create a function that implements this algorithm almost exactly. We take in a list and a value as input. We initialize the index *i* to 0 and we have a loop until *i* is no longer less than the length of the list. We have an if statement to compare the element *i* of the list with our value, returning true if they match or incrementing *i* if they don't. The example shows how we could use this. We have a list, favorite foods, and we call the function we just created on two values. Sure enough, when we look for a value that's in the list, we get a true back; and when we look for one that's not in the list, we get a false.

Now, there's a built-in function within Python that actually implements this algorithm for us—that's the in command. So, when we ask whether some value is in some list, Python is doing exactly what we just showed in the background. It's just looping over all the elements to see if the one we want is there. Like I said, that's a really simple algorithm and it seems very straightforward. But let's assume that instead of having a list of values in any old order, that instead our list was sorted from smallest to largest. A command to sort the list would be to call a sort method on the list. But for purposes of writing the algorithm, we can just assume that the list has already been sorted. Now, is there a better way that we might write the search routine? Maybe a way that can make use of the fact that our input is sorted to search for the value more efficiently?

The answer is, of course, yes. You're familiar with this already. If I gave you a dictionary and I asked you to create a program to look up a word, how would you do it? Well, going through every single word to see if it matches would be very inefficient. A more efficient approach would be to start by checking some point in the middle of the dictionary. If that's not the word that you're looking for, then you can figure out whether it came before or after the word that you wanted to find, and then look in the remaining half of the dictionary. You could continue this until you found the value, or else found that it was missing.

So, how might we write the pseudocode for this? Well, the input and the output is the same as what we had before. We take in a list of values and a value to find, and we return true or false. The only difference

is that the list of values is given in sorted order. So, now we need to describe the steps of our algorithm very precisely. Our approach will be to gradually narrow down the range of options until we find the one that we're looking for. So at any point, we'll have a maximum and a minimum index of where the value might be. At the very beginning, our maximum and minimum indices will be from 0 to the list size −1. We'll check those values to make sure it's not matching them.

So, at this point, we know that the value that we're looking for is somewhere between item low in the list and item high in the list. And we're going to gradually narrow down low and high until either we find the element that we're looking for or there's nothing left between low and high. So we're going to have a loop that continues as long as low is less than high minus one. Notice that we want to continue only as long as the high and low indices are at least two apart so that there's some potential value in between. Once the high and low values are next to each other, we might as well quit the search, since there's no possibility of another value in between them. Notice that at each iteration of the loop, we still have that same condition—the value we're looking for is either between element low and element high, or else it's not in the list. This condition—this thing that's the same every time we go through a loop—is called a loop invariant. When we're designing an algorithm, it's often helpful if we can identify such a loop invariant.

So, now we need to decide what's done in each iteration of the loop itself. We'll compute a midpoint that's halfway between the high and low point, and we'll check to see if it matches the value. Notice that when we calculate the midpoint between low and high, we can do so by finding the difference between low and high, dividing it by 2, and adding it to low. Notice that since we're dividing by 2, we could end up with a fractional value, which doesn't work for indices, so we need to make sure that we're doing an integer division—keeping only the quotient but ignoring the remainder. And notice that since we know that the high and low values are at least two apart from each other, high minus low divided by 2 is at least 1. So, the midpoint is guaranteed not to be the same as low, or the same as high; it'll be a new index somewhere between low and high.

All right, if the midpoint turns out not to be the actual value, we can at least use the midpoint to narrow our range. We're faced with one of two possibilities: We can either go forward with the range from low to midpoint, or the range from midpoint to high. We can decide which of these is the right subrange to continue with by comparing the value at the midpoint to the value that we're searching for. If the value at the midpoint is less than *v*, it means that we need to look in the upper subrange, so we can set low to be the midpoint. Going forward, we'll be looking between that midpoint and the high index. On the other hand, if the value at the midpoint is greater than *v*, it means that we should be using the lower subrange, so we can set high to be the midpoint. Notice that our loop invariant is maintained. The value that we're looking for is either between low and high, or it's not in the list at all. Finally, if we finish the loop, it means we narrowed in and the value that we were searching for was not found. And so, in this case, we'll return false.

So, this is our algorithm description, and we've described it using pseudocode. The term for this type of search is a binary search. When you hear binary, you should think two. And notice that at each iteration, we are reducing the search range by a factor of two. This is much more efficient than the linear search that we saw earlier, where we just looked at one item at a time, one after the other. If we had a huge list—say the size of a real dictionary—and we were searching through it, that binary search would be a whole lot faster than the linear search.

Now, given an algorithm description, it's pretty straightforward to convert this to Python code. Basically, every line of pseudocode gets converted into a line or two of Python code, and pretty much every line is taken directly from the pseudocode. Notice that when we compute the midpoint, I'm using integer division—the double slash rather than the single slash—to make sure that we get the integer quotient without any remainder. When we run this code with a sorted list, sure enough, we find that we correctly identify when an item is in the list or not. Now realistically, we'd want to make sure that we tested this in a variety of situations. For example, what if the list was empty?

To handle the possibility of an empty list, we can make one change to the pseudocode at the very beginning. If the list was empty to begin with—that is, if the length was less than 1—then we immediately return false. And that prevents us trying to access an element at index 0, which wouldn't exist. The rest of our testing would address questions like: What if we had just one element or just two? What if the thing we were looking for was the very first one, or the very last one? What if it was a giant list? Realistically, if we were writing an algorithm, we'd need to go through a testing process, just like always. And if we find errors, we might have to adjust the algorithm itself. However, here, I'll just tell you that this code actually does work in those cases.

Now, before we move on from searching, let's make one change. Often, we don't just want to know if something is in the list, we actually want to know where it is in the list. So, what if we wanted to modify our algorithm so that instead of returning true or false, it returned the index that matched the search value? If the value wasn't found, we could return an invalid index, like –1.

First, looking back at the algorithm pseudocode for binary search, think about how you would modify it to return a specific location. The changes are pretty minor. First, we'd want to change the description of the output for the algorithm in the algorithm itself. We're going to be returning the index if we find a match. Notice that we need separate checks for low and high, since they're separate values. If we find a match at a midpoint during the loop, we return the midpoint. And if we didn't find the value at all, we return –1.

Next, we could modify our Python code to reflect this algorithm. Given that new algorithm, how would you modify the code so that we return the index, or –1, if we don't find the value? Here's the code. Notice that our changes directly reflect the changes in the algorithm. We return the index whenever we find a match, or if we get to the end of the routine, we return –1. Linear and binary search are two of the simplest and most fundamental algorithms. Let's turn our attention to some slightly more complex algorithms—sorts.

Sorting algorithms have a long history in computer science, and they're traditionally the main example we use to illustrate the basics of algorithm development and analysis. One of the cool things about sorting is that there are lots of different ways to do it, and different methods will work better or worse in different circumstances. Plus, sorting is something that we're all familiar with in everyday life, so it's easy to make a connection between the ways you might sort things at home and the way you should sort them in a program. In this lecture, I'll introduce a couple of sorting algorithms, and we'll see a couple more next lecture.

We've already seen a hint of the problem we face when sorting, and why sorting can be useful. What if we have a list of values and the values are in no particular order? If we're wanting to find values in that list, we're stuck using a linear search. If we had a sorted list though, we could use binary search and do the checks a lot faster. Often, if we're working with sorted data, our operations are a whole lot easier and simpler than if it's just a random collection of values. Sorting is a key tool in our toolbox that we'll want to understand very well.

I want to compare two basic sorts—selection sort and insertion sort. Selection sort works like this: I'll start with a mixed up set of values that I want to put into order from smallest to largest. Since the first thing I'm going to want is the smallest item, I'll look through all of my values and I'll find the smallest one, and I'll put that into the first place. I'll then repeat that process to find the next smallest item, and I'll put that into the second place. Notice that each time, I'm going to have to look at all of my remaining items so that I can select the one that's the smallest remaining. That's where we get the name selection sort from. Another decision that comes up when sorting is whether to move around the original elements of the list or make a copy. Lists are mutable data types so you have the ability to reorder the elements themselves, if you want. If you do directly sort the elements of the list I gave you, that's called an in-place sort.

In contrast, an out-of-place sort means that you create a brand new list that's a copy of the original one. In this case, the original list stays unchanged while we also have a new, sorted list to work with. Choosing

whether you want an in-place or an out-of-place sort will depend on whether you need to maintain the original order for some reason. If so, you want an out-of-place sort. Probably the more common case is to just do the sort in-place.

In the selection sort, we can sort in-place by making sure that every time we place a new element into its final position, we swap it with an existing element. Let's look at the pseudocode for this algorithm. Our input is an unsorted list, and the output is the sorted list. We're going to have a loop that goes from the beginning to the end of the list. Each time, we'll look at all the remaining values, select the smallest one, and then swap that with the index that we're on. It's very simple, but it's also looking at every element, every single pass, so it might not be particularly efficient.

Now, converting our pseudocode to code for selection sort means setting up a loop. The loop we have goes over the full length of the list. Inside the loop, we first find the smallest remaining element. To do that, we'll keep track of the index of that smallest value—that's our min index variable—and we start out by assuming the smallest one is the first one of the remaining list. We then loop through all the other remaining elements, and if we find one smaller than the current minimum, we update min index. Now, once we've done this for all of the elements and found the smallest of the remaining ones, we swap the min index element into the current position that we're trying to fill. We can test this by creating a short list of items in random order. Then we'll call selection sort on this new list, and then print the results. Sure enough, it comes out sorted.

Now, selection sort works but it spends lot of time going through the entire unsorted list on every iteration. Let's try a different approach called insertion sort. For insertion sort, at iteration *n*, we've sorted the first n elements. So, the only thing to do on each iteration is add one more element into the right spot. So, let's see how that'll work.

With insertion sort, I'm going to start with the first item, and only the first item. In this approach, I want to avoid scanning the whole unsorted list.

But since I start with just this one item to work with, for purposes of this first step, it's already in sorted order. When I take the second number, all I do is compare it with the first number, and I insert it in whichever position is correct. I continue to make insertions the same way, where each time I'm going to take one more value and insert it into the list of sorted items. That's where we get the name insertion sort. Now that we've described this algorithm, maybe you want to practice writing pseudocode yourself. Really, take a minute to see if you can sketch out the steps that are needed to run an insertion sort.

OK, let's see what the pseudocode will look like. This might look a little trickier. The input and output are still the same, and we're still going to have a loop where we visit each element. Inside the loop is what's different from selection sort because we want to insert the next element into the already sorted part of the list. So, notice what we do. L[i] is the element *i* at each iteration, and we copy that element that we want to insert next into a variable, temp. At each iteration, we need to work our way backward down the list to figure out where to insert the current element. This is what our variable *j* is keeping track of—working our way down the list to figure out where to insert. We'll keep checking whether element *j* is greater than the value we're trying to insert. If so, we copy that *j*-th element into the next slot down.

Eventually, we reach either the beginning of the list or an element that's not greater than the temporary element that we're wanting to insert. So, we can put that temporary element into that place. There will be a space just waiting for us there, since we already slid everything else down one. It might help you to look at this pseudocode and try it out for yourself, to make sure you see how this is working.

Now, given that pseudocode, what would the code itself look like? Remember, the pseudocode did almost all the work for us so it should be pretty straightforward to convert the pseudocode into actual code. Here's the code. Now notice, the pseudocode already did almost all the work for us. In fact, it's basically one line of Python for each line of pseudocode. And when we run the code, it does indeed sort our data. Sorting is actually such a common operation that Python has built-in a

sorting function for you to use. For a list, we can call a sort method on the list, by just saying sort. That's right. We just call sort and it will sort the list. This is an in-place sort, so what does that mean for the original data? Right, it's been sorted in place. The list is rearranged.

What if we want an out-of-place sort? Maybe we want to keep the original list in its own order, but we also want a sorted version. For those cases, Python offers a more general command called sorted. We can use the sorted function, passing the list in as a parameter. The function returns a sorted list. If we print out the original list, we see it's still in the same order that it was before, and the returned list is in sorted order.

Both the built-in sort command and the built-in sorted command can be modified to sort in reverse order. To do this, we just set the reverse parameter to true when we call the function. You can see here an example of doing a reverse sort on the dataset, in this case, with the out-of-place sort. Python's built-in sort function is famously great and it's even been adapted for use in Android and Java. But how it works combines insertion sort with another basic sorting algorithm called merge sort, which we'll discuss in the next lecture.

Developing algorithms that can be applied to solve problems is a big part of what computer science is all about. And as the search and sort algorithms illustrate, we can design algorithms to address more general problems as well. The process of algorithm design is so critical to programming that most of our code can be thought of as just an implementation of algorithmic ideas. Sorts, in particular, provide an easy way of illustrating many key concepts of algorithm design. Once you understand how various sorts work, you understand a lot of the fundamental ideas in algorithm development that can be used to make more complex programs.

In the next lecture, we're going to expand on algorithms by introducing a particularly interesting idea called recursion, and introduce a couple more sorts that make use of recursive ideas. And we'll see how we can analyze algorithms to compare how well different algorithms perform work in practice. See you then.

# 21 Recursion and Running Times

One algorithm can take so long to run that it will never complete in our lifetimes, while another one, solving the same problem, might take less than a second. The choice of which algorithm to use can be critical. But how do we know whether or not a particular algorithm is a good one to use in our program? To help answer this question, you will be introduced to an approach known as algorithm analysis.

## [ RECURSION: MERGE SORT ]

› Recursion can be, but isn't always, a great way to create efficient code. The great trick in recursion is that a function is calling itself.

› Let's say that we want to print a countdown. We want some function that takes in an integer value and then counts down to 0 from there. That function might look like the following. We define the function countdown, which takes in a number, *n*, as a parameter. That will be the number we are counting down from. We then print out the number that was passed in. Assuming that the number is greater than 0, we are going to call our own self again, but with *n*-1 as the parameter.

```
def countdown(n):
    print (n)
    if n > 0:
        countdown(n-1)
countdown (5)
```

```
OUTPUT:
5
4
3
2
1
0
```

- If you think of the function countdown as "a function that prints all numbers from *n* down to 0," then this makes a little more sense. When we call countdown with n-1 as the parameter, we're just saying "we are printing the numbers from *n*-1 down to 0." So, the overall function is "print the number *n* and then print the numbers from *n*-1 down to 0." Thinking about the function that way makes a little more sense.

- Of course, there are other, better, ways to count down from *n* to 0—that's what loops are made for. But this idea of recursion is going to let us do a few things that don't have such a nice non-recursive version, and it will help us organize some of our programming so that even if we can find a non-recursive solution, we'll have a tool for thinking about problems.

- One of the main approaches that can rely on recursion is what's called **divide and conquer**. The idea is that it's easier to deal with two smaller problems rather than one big one. But there's a more particular meaning to the term "divide and conquer" in computing. When we use the term, we mean that we are taking a large data set and dividing it into subsets that we handle independently.

- Let's look at two algorithms that rely on divide-and-conquer approaches, both of which are sorting algorithms.

- First, we have merge sort. Let's assume that we're given some completely unordered set of numbers. We're going to do three steps to get these sorted. First, we'll divide the set of numbers in two—using the first half to form one list and the second half to form the other list.

› The second step is to sort each of those lists. We can use the merge sort routine to sort the lists, and the sorting process is an example of divide and conquer. We're taking one large sorting problem and reducing it to two small sorting problems that we solve recursively. Finally, there's a merge stage, where we'll merge those two sorted lists into one bigger sorted list. To merge, we'll work through both lists, pulling out the smallest one left from whichever list.

› Let's put all of that into pseudocode, which is a great intermediate step for writing algorithms because it lets us specify the key ideas of the steps without also needing to specify all the syntax at the same time. In fact, less detail in pseudocode is sometimes better, because that leaves the programmer more flexibility to determine how to implement an item.

```
Input: unsorted list L, length n
Output: L, with elements sorted smallest to largest

1.  If n <= 1
    a.  Return L            #a list of length 1 is already sorted
2.  L1 = L(0:n/2-1)         L2 = (n/2:n-1)
3.  MergeSort(L1)           MergeSort(L2)
4.  L = Merge(L1, L2)       #Merge will be defined separately
```

› Like other sorts, we'll be taking in an unsorted list and returning a sorted one. The actual routine will start out with a special case, though. We'll first check to see if we have a list of length 1, and if so, we just return that list—because if we have a list of length 1, it's already sorted. Also, if our list has only one element, we can't divide it into two lists, so the rest of the routine isn't going to work.

› We refer to this sort of special case check as a **base case** when we are discussing recursion. A recursive routine that keeps calling itself has to stop at some point, or it will go on forever. The point where it stops is the base case.

- We have a less-than sign in there just in case someone sends us an empty list—there's no reason to try sorting anything less than the base case, either.

- If we have more than the base case—that is, if we have a list of 2 or more elements—we'll go through our three steps. First, we'll form two lists, *L*1 and *L*2, made from half of the original list. We'll then sort each of those lists by a recursive call to this very routine. Finally, we'll merge those lists together.

- In the actual code, we define our function, **mergeSort**, and take the list in as the parameter, *L*. We'll store the length of *L* in a variable, *n*. First, we handle the base case: If *n* is less than or equal to 1, we just return, because the list is already sorted. Otherwise, we'll form our two shorter lists.

```
def mergeSort(L):
    n = len(L)
    if n <= 1:
        return
    L1 = L[:n//2]
    L2 = L[n//2:]
    mergeSort(L1)
    mergeSort(L2)
    merge(L, L1, L2)
    return
```

- Notice two things about the transition from pseudocode to Python syntax in the next lines of code. First, we're using the slicing operation to take a subset of the lists, and we're using *n*/2 as the splitting point. So, we can write ":n/2" for the first sublist and "n/2:" for the second sublist.

- Second, in order for our code to specify that splitting point, *n*/2, we had to use integer division, where we drop the remainder, to make sure that we have an integer result for the index. That integer division is the double slash, as opposed to the single slash for regular division.

- The next two lines are the recursive calls to mergeSort.

› Finally, we have a call to a merge routine, which takes two lists and merges them into a third list.

```
def merge(L, L1, L2):
    i = 0
    j = 0
    k = 0
    while (j < len(L1)) or (k < len(L2)):
        if j < len(L1):
            if k < len(L2):
                #we are not at the end of L1 or L2, so pull the
                  smaller value
                if L1[j] < L2[k]:
                    L[i] = L1[j]
                    j += 1
                else:
                    L[i] = L2[k]
                    k += 1
            else:
                #we are at the end of L2, so just pull from L1
                L[i] = L1[j]
                j += 1
        else:
            #we are at the end of L1, so just pull from L2
            L[i] = L2[k]
            k += 1
        i += 1
    return
```

## [ RECURSION: QUICKSORT ]

› Another example of a recursive sorting routine is called quicksort, because it works quickly on typical cases. In quicksort, the idea is still divide and conquer, but the division is done differently than in merge sort.

- In quicksort, we pick some value to split everything around—typically just the first value in the list. We call this term the **pivot**. We then form two new lists: one with all the values less than the pivot and one with all the values greater than the pivot. Next, we sort each of those lists—again with a recursive call, to quicksort. After that, the whole list is sorted: We have the first list, followed by the pivot, followed by the last list.

- When we write pseudocode for quicksort, the input and output are just like we've had before. We'll take in an unordered list, and our output will be a sorted list. For a recursive routine, we want to have a base case. The base case will be just like in merge sort—we want to return if we have a list of length 0 or 1.

- Next, we pick the first element of the list, the pivot. Then, we form two lists, *L1* and *L2*—one with elements below the pivot and one with elements above. We then recursively sort the two lists. This is our recursive call, where we call the quicksort function from within the quicksort function. Once the lists are sorted, we form our final list, by joining the two lists and the pivot.

```
Input: unsorted list L, length n
Output: L, with elements sorted smallest to largest

1.  If n <= 1
    a.  Return L            #a list of length 1 is already sorted
2.  pivot = L[0]
3.  Form lists L1 and L2 of remaining elements less/greater than
    pivot
    a.  Set empty lists L1 and L2
    b.  Loop through elements of L from 1 onward
        1.  If the element is less than the pivot, add it to L1,
            otherwise add it to L2
4.  QuickSort(L1) QuickSort(L2)
5.  L = Join(L1, pivot, L2)
    a.  Clear L
    b.  Loop through L1, appending elements on to L
    c.  Append pivot to L
    d.  Loop through L2, appending elements on to L
```

› Let's use the pseudocode to write the code. The following is one way to implement quicksort.

```
def quickSort(L):
    #handle base case
    if len(L) <= 1:
        return
    #pick pivot
    pivot = L[0]
    #form lists less/greater than pivot
    L1 = []
    L2 = []
    for element in L[1:]:
        if element < pivot:
            L1.append(element)
        else:
            L2.append(element)
    #sort sublists
    quickSort(L1)
    quickSort(L2)
    #join the sublists and pivot
    L[:] = []
    for element in L1:
        L.append(element)
    L.append(pivot)
    for element in L2:
        L.append(element)
    return
```

## [ ASYMPTOTIC ANALYSIS AND RUNNING TIME ]

› In addition to the four different sort routines we've used—selection sort, insertion sort, merge sort, and quicksort—people have also developed a variety of other sorts. Sorting shows that there can be several different, sometimes very different, algorithmic solutions to the same problem.

- Given so many different possible solutions to a problem, how do we choose between them? The best choice of algorithm should be independent of the individual programmer. And because the motivation for much of computing has been increased efficiency, we often use efficiency as the criterion to choose one algorithm over another.

- In the case of sort routines, Python has a built-in sort routine, which uses a combination of a merge sort and an insertion sort. That built-in Python sort is really efficient. In fact, that should usually be the version you use. Most other languages will have some similar built-in sort function.

- To assess efficiency, computer scientists use what's called **asymptotic analysis**. This type of analysis looks at how a function performs as the input size grows larger and larger: How does the running time increase as the input size increases? For the algorithms we've looked at, such as searching in a list or sorting a list, the input size will be the length of the list.

- When it comes to asymptotic analysis, many programmers just need a general idea of practical running time. What will happen if we double the input size? For our searches or sorts, think of having a list twice as long.

- We also have to think about what part of the running time we care about. Do we care about the best case, the worst case, or the average case? Most of the time, computer scientists will analyze the worst case, because they want to make sure that things don't behave too badly. However, depending on the problem, we sometimes care about the best case or average case.

- As we work with various programs, we'll sometimes want to make sure that what we're doing is reasonably efficient. The bigger the data sets we deal with, the more important this is. But even with reasonably sized data sets, the difference in asymptotic complexity can make a big difference.

- The algorithm used in Python's built-in "sort()" function has been cleverly designed to do even better than each of the four algorithms we've been using, at least most of the time. Python's current algorithm uses a combination of merge sort and insertion sort.

- Even though merge sort works better on large problems, insertion sort works faster on smaller problems. The Python sort routine basically uses merge sort for the overall problem, but once it's dealing with sufficiently small problems, it switches to insertion sort.

- The Python algorithm uses recursion whenever there's a merge sort but avoiding recursion whenever there's an insertion sort. It does this because recursion is a useful way of describing certain calculations. In fact, for some calculations, it's the only good way of describing what needs to be done. But there are other times when recursion is possible but should not be used.

- By selecting a different algorithm, we can sometimes turn a problem that seems completely impossible into one that's easily solved. It's worth analyzing your code to determine running time, to make sure that you're not being wildly or unnecessarily inefficient in the algorithm you've chosen to solve a problem.

## Readings

Lambert, *Fundamentals of Python*, chap. 3, p. 49–59 and 70–81.

Zelle, *Python Programming*, chap. 13.

## Exercises

1 What does the following code do?

```
def dosomething(lst):
    if len(lst) == 0:
        return 1
    else:
        return lst[0]*dosomething(lst[1:])
```

**2** Consider the "swap," "one_bubble_pass," and "bubblesort" algorithms defined in the previous lecture's exercises. Considering that the worst case occurs when a list is entirely in reverse order, how would you characterize the running time of each of the three routines?

a) swap
b) one_bubble_pass
c) bubblesort

The options are as follows:

◊ constant time (independent of the number of elements in the list)
◊ logarithmic time (proportional to the log of the number of elements in the list)
◊ linear time (proportional to the number of elements in the list)
◊ *n* log *n* time (proportional to the length times the log of the length of the list)
◊ quadratic time (proportional to the square of the length of the list)
◊ exponential time (proportional to an exponential function of the length of the list).

TRANSCRIPT 21

# Recursion and Running Times

Here's something pretty amazing—one algorithm can take so long to run that it will never complete in our lifetimes, while another one, solving the exact same problem, may take less than a second. The choice of which algorithm to use can be critical. So, how do we know whether or not a particular algorithm is a good one to use in our program? The two broad approaches to algorithms that we'll explore in this lecture are iteration and recursion. They're more different than their names may sound.

Recursion is one of the most interesting ideas in computer science. We'll apply recursion to form a couple of sorting algorithms. To do this, we'll initially set up our programs using pseudocode. And, to see how to answer the question of which algorithm is good, we'll use an approach known as algorithm analysis to compare some of the built-in routines that Python provides to support sorting.

Now, recursion can be, but isn't always, a great way to create efficient code. The great trick in recursion is that a function is calling itself. To see what I mean, let me first give you an example of a name that refers to itself. There's a popular package manager for Python that we've talked about before called pip, and pip stands for "pip installs Python" or "pip installs Packages." There's no deep reason to have that name, but it's a playful example of recursion—the name is in the name. Similarly, if we have a program that exhibits recursion, we expect to see a function that calls itself.

Let's say that we want to print a countdown. Now, we've already seen how to do this with iteration; we just use a loop. But let's look now at how we can do a countdown with recursion. We want some function that takes in an integer value, and then counts down to zero from there. Here's what that function might look like. We define the function "countdown," which takes in a number, *n*, as a parameter. That'll be the

number we're counting down from—so far, so good. We then print out the number that was passed in. Again, this seems OK. Now, here comes the weird part. Assuming that the number is greater than zero, we're going to call our own self again, but with $n - 1$ as the parameter.

Now, if you think of the function "countdown" as a function that prints all numbers from $n$ down to zero, then this might make a little bit more sense. When we call countdown with $n - 1$ as the parameter, we're just saying we're printing the numbers from $n - 1$ down to zero. So, the overall function is print the number $n$ and then print the numbers from $n - 1$ down to zero. Thinking about the function that way makes a little bit more sense.

Now, iteration is better than recursion for counting down from $n$ to zero; again, that's what loops are made for. But, this idea of recursion is going to let us do some other things that don't have such nice, non-recursive solutions. One of the main problem-solving approaches out there that can rely on recursion is what's called "divide and conquer." The idea is that it's easier to deal with two smaller problems rather than one big one. But, there's a more particular meaning to the term divide and conquer in computing. When we use the term, what we mean is that we're taking a large data set and we're dividing it into two subsets that we handle independently.

Let's look at two algorithms that rely on divide and conquer approaches. Both of these are sorting algorithms, quite different from the ones that we've seen before. I'll explain both of them conceptually first, then we'll write some pseudocode, and then we'll use the pseudocode to write the code that we need in Python.

First, we have merge sort. Let's assume that we're given some totally unordered set of numbers. We're going to do three steps to get these sorted. First, we'll divide the set of numbers in two. That's really easy; just use the first half to form one list and the second half to form the other list. The second step is to sort each of those lists. So, do we have a routine to sort a list? Yes—we have the merge sort routine that we're using right now. The sorting process is going to be an example of divide and conquer. We're taking one large sorting problem and

we're reducing it to two smaller sorting problems, and we'll solve each of those recursively. Finally, there's a merge stage, where we'll merge those two sorted lists into one bigger sorted list. To merge, we'll walk through both lists, pulling out the smallest one left from whichever list.

Now, let's put all of that into pseudocode, which is a great intermediate step for writing algorithms since it lets us specify they key ideas of the steps without needing to worry about the details of syntax. In fact, less detail in pseudocode is sometimes better, since that leaves the programmer more flexibility in how to implement an item.

Like the sorts that we've seen in the past, we'll be taking in an unsorted list, and we'll be returning a sorted one. The actual routine will start out with a special case, though. We'll first check to see if we have a list of length 1, and if so, we just return that list. Why do we do that? Well, if we have a list of length 1, it's already sorted, so there's nothing more to do. Also, if our list has only one element, we can't really divide it into two lists, so the rest of this routine isn't going to work. We refer to this sort of special case check as a base case when we're discussing recursion. A recursive routine that keeps calling itself has to stop at some point, otherwise it's going to go on forever. The point where it stops is the base case. Oh, and we have a less-than in there just in case someone sends us an empty list. There's no reason to try sorting anything less than the base case, either.

Now, if we have more than the base case—that is, if we have a list with at least two elements in it—we'll go through our three steps. We'll first form two lists, and I'll call them L1 and L2, made from half of the original list. We'll then sort each of those lists by a recursive call to this same routine, the one that we're writing right now. And, finally, we'll merge those lists together. Merging is another function call that I'll define separately.

So, let's see what the actual code will look like. We define our function, mergeSort, and we take the list in as the parameter L. We'll store the length of L in a variable, *n*. First, we handle the base case: if $n <= 1$, we just return, since the list is already sorted. Otherwise, we'll form our two shorter lists.

Notice two things about the transition from pseudocode to Python syntax in these next lines of code. First, we're using the slicing operation to take a subset of the lists, and we're using $n \div 2$ as the splitting point. So, we can write colon *n* over 2 for the first sublist, and *n* over 2 colon for the second sublist. You might notice that the notation is a little different than in our pseudocode. That's because our pseudocode was written to describe the general case, whereas in Python, the way we write our code is more precise. Second, in order for our code to specify that splitting point, $n \div 2$, we had to use integer division, where we drop the remainder, to make sure that we have an integer result for the index. That integer division is the double slash, as opposed to the single slash for regular division.

The next two lines are the recursive calls to mergeSort. And then, finally, we have a call to a merge routine, which takes two lists and merges them into a third list. Rather than go through pseudocode on this part, I'll just jump straight to some finished code and walk you through it.

In this routine, we're going to keep track of three variables—*i*, *j*, and *k*—that keep track of the index for the L, L1, and L2 arrays. The basic idea is that we're going to go through the L1 and the L2 arrays, pulling the smallest value from between them and putting it into the L array. So, we'll start with a "while" loop that says we continue as long as we haven't hit the end of both the L1 and L2 arrays. First, we'll check to see if we're at the end of the L1 array—that is, if our *j* index is already at the length of the array. If it's too large, we'll follow the else clause in this statement, and we'll pull a value from the L2 array; otherwise, we continue on.

We next check to see if we're at the end of the L2 array—that is, if the *k* index is already at the length. If we've hit the length there, we follow an else clause that pulls a value from the L1 array. But, assuming we're not at the end of either L1 or L2, we're going to pull the value from whichever one has the smallest current value. So, we compare L1 sub *j* to L2 sub *k*, and we choose which of the two arrays to pull from. If we're pulling from L1, we assign the value in L1 to L, and we increment *j* to the next index. If we're pulling from L2, we assign the value in L2 to L, and we

increment the *k* to the next index. At the end of this loop, since we've added something to L, we increment *i* to the next index for the L array. And we can test this out using the same input that we've used to test earlier sorts. We'll create a list of foods and we'll call mergeSort, printing the results. Indeed, we get a sorted list out, just like we'd expect.

Let's look at one more example of a recursive sorting routine. This one is called quick sort, since it works famously fast on typical cases. In quick sort, the idea is still divide and conquer, but the division is done differently than in merge sort. In quick sort, we pick some value to split everything around, typically just the first value in the list, and we call this term the pivot. We then form two new lists—one with all the values less than the pivot, and one with all the values greater than the pivot. Next, we sort each of those lists, again with a recursive call to quick sort. After that, the whole list is sorted: we have the first list, followed by the pivot, followed by the last list.

Now, let's work through how we'd write some pseudocode for quick sort. The input and output are just like we've had in the past—we take in an unordered list, and our output will be a sorted list. Again, for a recursive routine, we want to have a base case. So, what do you think the base case will be for quick sort? Our base case will be just like in merge sort; we want to return if we have a list of length 0 or 1.

What's next? Well, the next thing we should do is pick the pivot, which we'll just do from the first element of the list. What comes after that? Next, we would form two lists, L1 and L2—L1 with elements that are less than the pivot, and L2 with elements that are greater than the pivot. Now, now that we have L1 and L2, what do we do next? Well, next, we recursively sort the two lists. This is our recursive call, where we call the quickSort function from within the quickSort function. And, once the lists are sorted, what do we do then? Well, then we form our final list, by joining the two lists and the pivot.

Let's use the pseudocode we have to write the code for this. First, give it a try yourself. I'll give you some hints, which you can also pause, to keep you going. So, first, define the function, and handle the check for

a base case. Second, just like step two of the pseudocode, set your pivot. Third, you'll want to form two lists, one with values less than the pivot, one with values greater. Start with two empty lists, and then add in elements to one or the other. And then, for the recursive calls, you'll just call quickSort, passing in those two lists that you made. To join the lists, you can start with list 1, append on the pivot, and then append the elements of list 2.

Let's look at a finished implementation, to see one way you might have implemented quick sort. I've included some comments to make sure that we're following. First, we have our function header, defining quick sort with the list L as our parameter. Next, we'll have our base case check: if the length of L < = 1, we're done, so we return. Next, we have our pivot defined, just taking the first element of the list L. All of this, so far, is pretty straightforward from our pseudocode. We next have the lines that form our two sublists. We'll start with two empty lists, L1 and L2, and then we'll have a loop that goes through all the elements, except the first element, in the input list. We write this with a "for" loop—that is, elements from the second one in L until the end. For each of those, we append to either L1 or L2 depending on whether we're below or above the pivot.

All right, next, we have our recursive calls to quick sort: one for L1, and one for L2. And, finally, we have our code to join the two lists. We start out by deleting the elements already in L. Remember that, because we're relying on L being mutable, we can't assign L itself to a new value—we have to just modify the elements in L. Then, we go through list L1, adding its elements to L. Next, we add the pivot. And last, we go through L2, adding its elements to L. Again, if we test this routine out, we'll find that it sorts input just fine.

So, we've now seen four different sort routines: selection sort, where we kept selecting the next smallest value; insertion sort, where we would take the next value and insert it into the sorted list; merge sort, where we divided the whole list into two nearly equal parts and sorted those recursively; and quick sort, where we divided the whole list into smaller and larger parts around a pivot, and then we sorted those recursively. People have also developed a variety of other sorts. Now, what sorting

shows very clearly is that there can be several different, sometimes very different, algorithmic solutions to the same problem.

Well, given so many different possible solutions to a problem, let's look at how we choose between them. In practice, sometimes people just choose "what's the easiest thing for a programmer to understand" or "what's the easiest thing to write code for," and these actually aren't bad reasons—you should only write code for things that you understand. But, the best choice of algorithm should be independent of the individual programmer, and, since the motivation for much of computing has been increased efficiency, we often use efficiency as the criterion to choose one algorithm over another.

In the case of sort routines, we already saw that Python has a built-in sort routine, which uses a combination of merge sort and an insertion sort. That built-in Python sort is really efficient. In fact, that should usually be the version that you use. Most other languages will also have some sort of built-in sort function. Now, to assess efficiency, computer scientists use what's called asymptotic analysis. This type of analysis looks at how a function performs as the input size grows larger and larger. We ask, "How does the running time increase as the input size increases?" For the algorithms that we've looked at, like searching in a list or sorting a list, the input size will be the length of the list.

Now we could spend a couple of lectures diving into asymptotic analysis, but, for many programmers, we just need a general idea of practical running time. A good way to think about this is, "What will happen if I double the input size?" For our searches or our sorts, think of having a list twice as long. We also have to think about what part of the running time we care about. Do we care about the best case, or the worst case, or the average case? Most of the time, computer scientists will analyze the worst case, since we want to make sure that things don't behave too badly. However, depending on the problem, you sometimes really care about the best case or the average case.

Let's start by thinking about a search. If I have a list with $n$ items in it, and I do a standard, linear search, in the best case, the item I'm looking for is

the very first one in the list. So, I just check one element and I'm done. In the worst case, I go through all *n* items and I don't find it at all, or I find it in the very last one. On average, I'll have to go through half the items—or *n* over 2—to find it, if it's in the list at all.

Now, what happens if I have a list twice the size—that is, 2*n*? I still have the same situation in the best case: one single check. In the worst case, I have to go through all 2*n* items. And, in the average case, assuming it's in the list at all, I'm going through half of the list, or *n* items. So, notice that when I doubled the input size, I doubled the worst-case and the average-case search times, and we describe this as a linear process, since the running time is a linear function of the input size. We'll also say it order *n*, and we'll sometimes write it big O of *n*. And yes, that's the right term; we call this big O notation.

Now, let's consider binary search, where I divide the list in half each time. Again, the best case is one where the first element I check is the one that I'm looking for. Now, each iteration of the search, I prune away half of the remaining elements. So, in the worst case, I keep dividing by 2 until I'm down to just one element. If I have less than 8 elements, then 3 checks is enough. If I have less than 16 elements, then 4 is enough. And, in general, for a list less than length *n*, I'll have log base 2 of *n* checks required. In the average case, I have just one less check, so it's only one less than log base 2 of *n* checks—basically the same. And, if I double my list size, in the worst case, I have just one more check.

So, mathematicians describe this as a log function, since the running time is a logarithmic function of the input size, and we'll write it as big O of log *n*. Clearly, if we're choosing between a linear process and a logarithmic process, given a sufficiently large data set, we'll always prefer the log process. If we have the option of a log search, we shouldn't choose a linear one.

Let's use this kind of comparison to see how some of the sorts work, in terms of running time. Let's start with selection sort—that's where we go through the whole list and pull out the smallest element, and then go through again to find the next smallest, and so on. To understand how

this runs, we should think of something of size *n*. On our first iteration, we have to check through all *n* items in the list, and then perform one swap. The next iteration, we check $n - 1$ items in the list, and perform one swap. Then we check $n - 2$ items, and so on, until finally we're down to just one item left.

So, the total number of checks that we needed in this case was $n + n - 1, + n - 2, + n - 3$ all the way down to 1. This is the sum from 1 to *n*, which you could do step by step or you could use the summation formula: $n \times n + 1$ over 2. Now, as *n* gets larger and larger, the only part of that that really matters is the *n* squared part—the minus *n* is insignificant compared to the *n* squared portion. This is called a quadratic running time, or more commonly, big O of *n* squared. Notice that selection sort always takes the same amount of time, so the best case and the worst case running times are the same.

Now, let's consider insertion sort instead. Remember that we'll always be inserting one new value into an already sorted list. In the best case, the whole list is already sorted, and in this case, we only have to do one comparison for each item, so the whole thing takes order *n* time. In the worst case, we're in complete reverse order. So, we can insert the first item with no comparisons; the next one, we'll have to compare once; the next one, we have to compare two times; the next one, three times, and so on. And, if you think about it, this is, again, a sum from 1 to *n* total comparisons. So, in the worst case, insertion sort is approximately *n* squared—just as bad as selection sort.

Now, let's consider merge sort. In merge sort, we have three steps each time. First, we split the list in two—that's easy, and it just takes a constant amount of time. Then, we sort each of those sublists. Then, we merge the two sublists together again. That merge takes order *n* time, since we visit each of the elements exactly once. Notice that merge sort is always order $n \log n$; there's no case better or worse than another.

Finally, we have quick sort. With quick sort, if we have a good split at each level, where about half the numbers are above and half are below, the running time is very similar to merge sort, for the same reasons. So,

it's order *n* log *n*. In the worst case, we have an already sorted, or an already reverse-sorted, list, and we end up operating a lot like a selection sort, so we'd be order *n* squared. On average, we're nearer to the best case. In fact, quick sort can be implemented very efficiently, so it's often faster than merge sort in practice, even though it could be worse in some cases. If we look at a summary of all of these algorithms, we can get a sense of how they'll behave in various cases. Notice that an insertion sort is order *n* in the best case. In fact, it's the best of any of these.

Now, as we work with various programs, we'll sometimes want to make sure that what we're doing is reasonably efficient. The bigger the data sets we deal with, the more important this is. But, even with reasonably sized data—like, up to 1000 elements—the difference in asymptotic complexity can make a difference. This is a good place to say a little more about the algorithm used in Python's built-in sort function, since it's been cleverly designed to do even better than each of the four algorithms that we've been discussing individually, at least most of the time. Python, in its early days, used quick sort, before its current algorithm was created by Tim Peters using a combination of merge sort and insertion sort.

Even though merge sort works better on large problems, insertion sort will work faster on smaller problems. The Python sort routine basically uses merge sort for the overall problem, but once it's dealing with sufficiently small problems, it switches to insertion sort. It also has some checks to find out if a portion of the list is already sorted. Lists that are at least partially sorted are more common than you might suppose, especially for data that isn't in absolutely random order. If we think about this in terms of recursion, the Python algorithm is using recursion whenever there's a merge sort, but it's avoiding recursion whenever there's an insertion sort. Why might that combination be a good idea?

Well, recursion is a really useful way of describing certain calculations. In fact, for some calculations, it's the only good way of describing what needs to be done. But, there are other times when recursion is possible, but you should, well—run away. To make this point clear, I'm going to show you a recursive program, and then I'm going to show you an alternative way of computing it that's a whole lot more efficient.

To do this, I'm going to use some Fibonacci numbers. Fibonacci numbers are numbers in a sequence, the Fibonacci sequence, where each number in the sequence is the sum of the two previous numbers. The first two Fibonacci numbers, the zeroth and first Fibonacci numbers, are 0 and 1. Then, we add the two adjacent numbers to get the next number. So, the second one is 0 + 1, or 1. The third one is 1 + 1, or 2. The fourth one is 1 + 2, or 3.

Fibonacci numbers have all sorts of interesting incarnations in math and nature. So, let's see how we might write that as a recursive routine. So, think about this: how would you write a routine, named Fib, where you pass in a number, and it gives you that number in the Fibonacci sequence by making a recursive call. So, passing in 0 or 1 would return 0 or 1; passing in 4 would return 3, and so on.

OK, here's one way you could do this. First, we define Fib, taking in the index *n*. We first check for the two base cases, if *n* is 0 or 1. In these cases, we just return 0 or 1. Otherwise, we compute Fib for *n* − 2 and Fib for *n* − 1, and we return that. Now, this code is correct, and it will compute Fibonacci numbers, but it's also horrendously slow. Try printing out Fib of 4 or 5 or 6, and you'll get an answer. But, try printing out Fib of 100, and your computer will take a long time to finish—too long.

The problem here is that each call generates twice the number of function calls as the previous level. So, a call like Fib 100 will generate 2 to the 100$^{th}$ recursive function calls until it reaches the base cases. That's something like a nonillion function calls. Have you even heard of a nonillion? It's a trillion times a trillion times a trillion. So, no wonder that was taking so long; this algorithm is taking exponential time. If we analyzed the running time of the algorithm, we'd have to conclude that it's not very practical. In fact, that's roughly the time it would take one of very fastest current computers if it ran for more than 10 million years.

Now, fortunately, we didn't need to use recursion to compute the Fibonacci numbers. If you look at the computation, you can see that there are a lot of cases where we're computing the same number over and over. Instead of writing a recursive routine, see if you can write a

routine that calculates the numbers in order, until you reach the one that you care about. So, you would first compute 0 and 1, then 2, then 3, and so on, until you get the one you want.

Here's a much better way of computing Fibonacci numbers. We create a list, F, initializing the first two elements to 0 and 1. Then, we'll keep adding on elements by adding the previous two together. In this way, we can compute all the way to the 100$^{th}$ Fibonacci number easily. The answer's really big, but at least it's quickly computed. Now, what kind of running time process does this new algorithm have? Think about it, and see if you can figure it out? This is a linear algorithm. The reason is that each of the Fibonacci numbers is computed exactly once. If we want to compute a number twice as big, it takes twice as long. That's a whole lot better than exponential.

Here's the point. By selecting a different algorithm, in this case a linear one that's iterative instead of an exponential one that's recursive, we can sometimes turn a problem that seems completely impossible into one that's easily solved. The recursive approach to the Fibonacci calculation is just not practical, whereas the iterative approach nicely matches how Fibonacci numbers are defined. The lesson to learn is that it's worth analyzing your code to determine running time, to make sure that you're not being wildly or unnecessarily inefficient in the algorithm you've chosen to solve a problem.

Understanding algorithms is of central importance for much of computer science, and the basic problems in searching and sorting that we've seen are the classic examples because they're so widely used. By analyzing the running time of our algorithms, we can make choices that will lead to better code and more efficient use of processor time. Recursion can offer an elegant way of solving some problems for us, but we have to be careful with it. Whenever iteration is possible, it'll usually be more efficient. However, there are cases where recursion is really the only formulation that makes sense. We'll see an example of this in our next lecture, when we discuss graphs, trees, and a couple of recursive algorithms.

# 22 Graphs and Trees

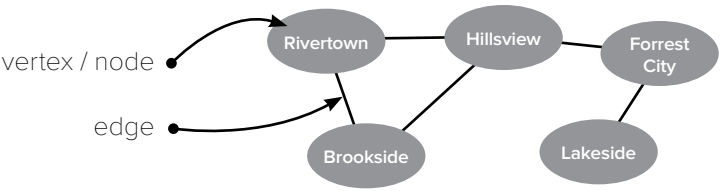
Graphs offer a structure for capturing an incredibly wide diversity of relationships and the connections that are formed all over. They represent connections between locations, between people, among species, and among data. These and many other relations are well represented on a graph. Once relationships are captured in a graph, algorithms let us use the graph effectively in our programs. In this lecture, you will learn about graphs, as well as trees—a particularly useful type of graph that makes organizing data easier.

## [ GRAPHS ]

- Imagine that you live in a kingdom with five main cities: Rivertown, Hillsview, Brookside, Lakeside, and Forrest City. Three of these cities are connected to each other directly with roads. Lakeside is connected only to Forrest City, and the only other road from Forrest City connects it to Hillsview.

- Whenever you have a bunch of entities—cities, in this case—with connections to each other, you have a data structure that's called a **graph**. In this sense, graphs, which are studied in a field called "graph theory," are used for everything from airplane connections, to ecological webs among species, to social networks.

- In a graph, we call the "things" that we're representing a "vertex" or a **node**. These are the cities in our example. The connections between nodes are **edges**. In our example, these are the roads. Edges let us know that there's a relation between the two nodes—for example, that they're connected by a road.




- In terms of writing code, graphs can have more than one representation. For example, there is one that is more global and one that focuses on adjacent neighbors. The more global option is to represent a graph as two lists: one list of nodes and one list of edges.

- Each node will contain information about itself. So, each of our city nodes might contain just the name of a city, or each node could also contain other information, such as city population, GPS coordinates for the city, and so on.

- An edge would have the names of the two nodes—in this case, the two cities that it connects. If it's a weighted graph, the edge would also store a **weight**, which in this case might be the length of that road, in miles or kilometers.

- Let's try to write code to support this. We'll want two classes, one for the nodes and one for the edges. Remember that a class helps us encapsulate the stuff that goes together, so the node class should incorporate the stuff in a node, and the edge class should incorporate the stuff in an edge.

- The node class will have a name for the city and a population. We'll set these with our initialization routine, which sets the instance attributes "_name" and "_pop." This is why we have the "self" reference; each node can have a different name and population. We make sure that we can access the name and population variables through two accessor functions, "getName" and "getPopulation."

› Likewise, our edge class would have the names of two cities, along with the distance along the road. These are set with the "init" initialization routine, which sets local instance attributes for "city1," "city2," and "distance." Then, we have a set of accessor functions to get each city name, or the pair of city names, and the distance.

```
class node:
    def __init__(self, name, population=0):
        self._name = name
        self._pop = population
    def getName(self):
        return self._name
    def getPopulation(self):
        return self._pop
class edge:
    def __init__(self, name1, name2, weight=0):
        self._city1 = name1
        self._city2 = name2
        self._distance = weight
    def getName1(self):
        return self._city1
    def getName2(self):
        return self._city2
    def getNames(self):
        return (self._city1, self._city2)
    def getWeight(self):
        return self._distance
```

› The following is how we might set up our node list and edge list for the city example. We'll create our five cities, each with some population, and add those to the city list. For example, we create a node for Rivertown with a population of 100 and append it to the cities list.

```
cities = []
roads = []
city = node('Rivertown', 1000)
cities.append(city)
```

```
city = node('Brookside', 1500)
cities.append(city)
city = node('Hillsview', 500)
cities.append(city)
city = node('Forrest City', 800)
cities.append(city)
city = node('Lakeside', 1100)
cities.append(city)
road = edge('Rivertown', 'Brookside', 100)
roads.append(road)
road = edge('Rivertown', 'Hillsview', 50)
roads.append(road)
road = edge('Hillsview', 'Brookside', 130)
roads.append(road)
road = edge('Hillsview', 'Forrest City', 40)
roads.append(road)
road = edge('Forrest City', 'Lakeside', 80)
roads.append(road)
```

› We'll also create our roads—five of them, in this case—and add them to the road list. For example, we form an edge between Rivertown and Brookside of length 100 and then append that edge to the roads list. It would be simple to add another road between two cities, or another city—just create a new edge or node and append it on.

› In addition to the first method of storing graphs—by keeping a global list of edges—there is a second way: by keeping a list of edges in each node. This second type is called an **adjacency list**.

› The global list of all the edges is probably most useful if you find yourself regularly needing to look at all of the edges. That's the approach commonly used to represent geometric models, like you would have in three-dimensional graphics.

› The second approach—the adjacency list, where you keep a list of the edges within each node—is useful in most typical graph operations, such

as airline connections, social networks, and so on. The adjacency list works well because most graph algorithms are already designed to work just looking at one node at a time and its neighbors.

- There is also a third method to store graphs, called an **adjacency matrix**, in which, instead of list, there are matrix entries that note which nodes are connected. This can be useful for operations where you need to quickly run over many different values or perform certain computations that can be expressed using linear algebra.

- An adjacency matrix can be the most compact form of a graph, especially when there are many edges. This is a key factor when graphs are huge. And it's the fastest of the representations if you need to check whether a particular pair of cities is connected.

- For any given graph, we can define a variety of graph algorithms. Some of these algorithms are simple. For example, returning a list of all the cities adjacent to one city is a very basic algorithm. The representation used to store the graph will determine exactly how the algorithm will perform.

- Graph algorithms let us analyze all kinds of things about the structure of graphs. For example, the **breadth-first search** algorithm lets us analyze how many degrees of separation there are between two people in a social network.

- With roads, we can usually travel the road in either direction—we just say that those nodes are connected. Graphs like this are called "undirected" graphs. A graph where people are friends is undirected: If person A is friends with person B, then person B is also friends with person A.

- However, we can also have cases where two things are linked, but it's not an equal connection between the two sides. For example, imagine cities connected by airline routes. Not all airline routes fly round-trips. Sometimes, a plane will fly from city 1, then to city 2, then to city 3, and then back to city 1. In this case, the edges go from one city to another, but not necessarily the other way around.

- So, there should be an edge from city 1's node to city 2's node, and one from city 2's node to city 3's node, and one from city 3's node back to city 1's node. These are called "directed" edges, and the resulting graph, made up of **directed edges**, is called a "directed graph," or "digraph."

- Web pages and the links between them form a directed graph. If web page A has a link to web page B, there's not necessarily one from web page B back to web page A.

- We say that a graph is **connected** if there is some sequence of edges connecting every pair of nodes. A graph has a **cycle** if there's some way to follow a set of edges and end up back where you started. In our city example, there's a cycle—you could go from Rivertown, to Hillsview, to Brookside, and then back to Rivertown.

## [ TREES ]

- A connected, undirected graph that does not contain a cycle is called a **tree**. Trees are such a useful structure that a whole set of algorithms have been developed just for trees.

- The following is an example of a tree. All the nodes are connected to each other, and there's no cycle in the graph.

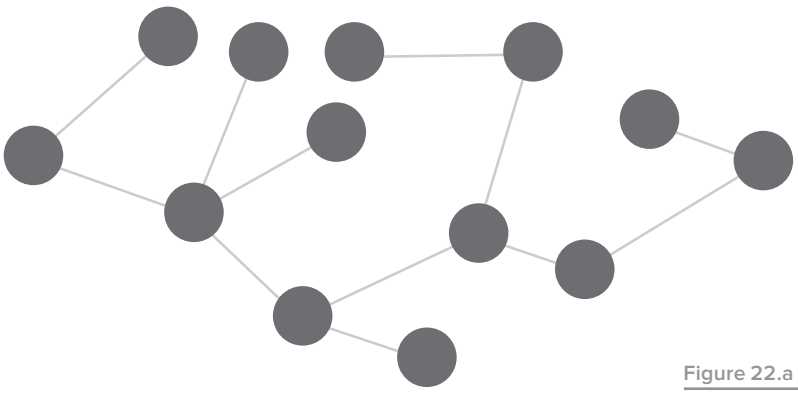


Figure 22.a

- Usually, when we talk about trees, we'll designate one node as the root, which can be thought of as the starting point, or the central point. It's the top level of a hierarchy. The rest of the tree can then be arranged in terms of "levels" from the root, where the root is at level 0, and each subsequent level is formed based on how many edges must be followed to get to the root.

- All the nodes connected to the root are considered its children and form the first level. The nodes they are connected to form the second level, and so on. For any node, the node it is connected to at the previous level is called its parent, and the nodes below it are called its children. A typical drawing of a tree will put the root at the top and the children in levels below.

- Because trees have this particular structure, they often have a particular way they are stored in code. Trees almost always use an adjacency list, where the edges are stored inside each node. And they usually store the parent and the children separately. So, any one node will store its own information, along with an index (or some other notation, such as a name) for the parent node, along with a list of its children.

- In code, we'll keep a variable to give the parent, and we'll keep a list of variables that are the children. We can add additional children whenever we need to.

```
class node:
    def __init__(self, name, parent=-1):
        self._name = name
        self._parent = parent
        self._children = []
    def getName(self):
        return self._name
    def getParent(self):
        return self._parent
    def getChildren(self):
        return self._children
```

```
    def setParent(self, p):
        self._parent = p
    def addChild(self, c):
        self._children.append(c)
```

› A very common type of tree is called a **binary tree**. In this case, every node has no more than two children. We'll usually call them a left child and a right child. So, a node will hold a parent, a left child, and a right child.

```
class node:
    def __init__(self, name, parent=-1):
        self._name = name
        self._parent = parent
        self._left = -1
        self._right = -1
    def getName(self):
        return self._name
    def getParent(self):
        return self._parent
    def getLeft(self):
        return self._left
    def getRight(self):
        return self._right
    def setParent(self, p):
        self._parent = p
    def setLeft(self, l):
        self._left = l
    def setRight(self, r):
        self._right = r
```

› Binary trees have many uses, but a common one is to store objects in sorted order. We call these **binary search trees**. Unlike arrays or lists that we might have to sort every time we add a new value, a binary tree can keep items always in sorted order. It's usually faster to add an item to a binary tree than to add it to an array or list that's been sorted.

› With a binary search tree, at any node in the tree, all the descendants on the left side are less than the node, and all those on the right side are greater than the node. Notice, for example, that 21 is greater than the root, 15, so it's on the right side of the root. It's greater than the next node, 18, so it's also on the right side of it. But it's less than the next node, 23, so it's on the left side of that one.

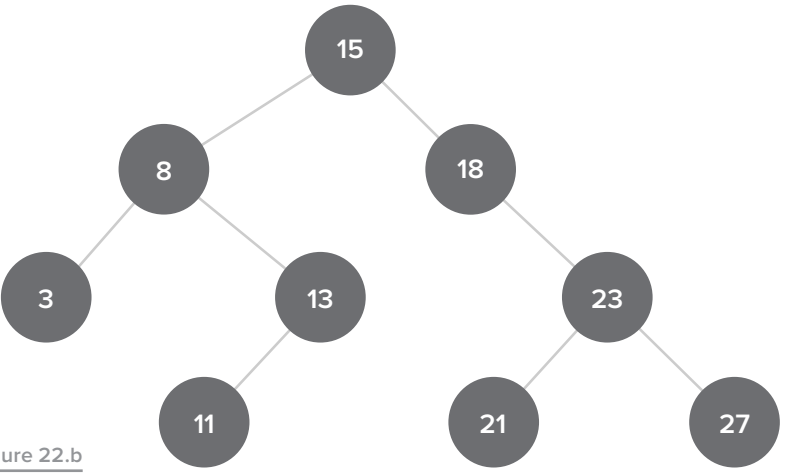



Figure 22.b

› There are various things we can do with a binary search tree, such as print out the entire tree in sorted order. This ends up being basically another sorting routine: We put elements into the binary search tree in order, and then when we print them out, we get a sorted list. If we take our set of nodes, insert them all into the tree, and then print it out, we get a sorted output list. This routine is fast, plus it works really fast, on average, if you have to update the sorted list.

## Reading

Lambert, *Fundamentals of Python*, chaps. 10 and 12.

## Exercises

1 Consider the parts of a body: head, neck, torso, arms, legs, hands, and feet. Draw a graph showing the connectivity between the parts of the body.

2 From the graph in exercise 1, what is the longest number of edges you would need to follow to go from one body part to another?

3 Is the graph in exercise 1 a tree? Why or why not?

4 Show the binary search tree that you would get by inserting nodes into an empty tree in the following order: 3, 1, 8, 2, 9.

5 Note that instead of storing indices for cities, we could instead store the cities in a dictionary instead of in a list. Following is part of the code used to build a list of cities in the code from the lecture. How would you modify this so that it uses a dictionary of cities instead of a list?

```
cities = []
city = node('Rivertown', 1000)
cities.append(city)
city = node('Brookside', 1500)
cities.append(city)
city = node('Hillsview', 500)
cities.append(city)
city = node('Forrest City', 800)
cities.append(city)
city = node('Lakeside', 1100)
cities.append(city)
...
road = roads[0]
pop1 = 0
pop2 = 0
for city in cities:
    if city.getName() == road.getName1():
        pop1 = city.getPopulation()
    if city.getName() == road.getName2():
        pop2 = city.getPopulation()
```

TRANSCRIPT 22

# Graphs and Trees

Imagine that you live in a kingdom with five main cities: Rivertown, Hillsview, Brookside, Lakeside, and Forrest City, and there are roads between these cities. Three of them are all connected to each other directly with roads. Lakeside is connected only to Forrest City, and the only other road from Forrest City connects it to Hillsview. Well, whenever you have a bunch of entities—cities, in this case—that have connections to each other, you have a data structure that's called a graph. Graphs, in this sense, which are studied in a field called graph theory, are used for everything from airplane connections, to ecological webs among species, to social networks.

Let me give you a little terminology. In a graph, we call the things that we're representing a vertex or a node. These are the cities in our example, the points that are on the graph. The connection between nodes are edges; in our example, these are roads. Edges let us know that there's a relation between the two nodes—for example, that they're connected by a road. In terms of writing code, graphs can have more than one representation. I'll show you two ways of representing a graph, one that's more global, and one that focuses on adjacent neighbors. The more global option is to represent a graph as two lists: one, a list of nodes; and one, a list of edges.

Now, each node will contain information about itself. So, each of our city nodes might contain just the name of the city, or it could also contain other information, such as city population, or coordinates for the city, and so on. An edge would have the names of the two nodes, in this case the two cities that it connects. If it's a weighted graph, the edge would also store a weight, which in this case might be the length of the road in miles or kilometers.

Let's try to write code to support this. We'll want two classes—one for the nodes, and one for the edges. Remember that a class helps

us encapsulate the stuff that goes together, so the node class should incorporate all the stuff in a node, and the edge class should incorporate all the stuff in an edge. So, what would our classes look like?

OK, we'll have two classes: a node class, and an edge class. The node class will have a name for the city, and I also stored a population. We'll set these with our initialization routine, which sets the instance attributes: _name, and _pop. This is why we have the "self" reference—each node can have a different name and population. We make sure that we can access the name and population variables through two accessor functions, getName and getPopulation. Now, likewise, our edge class would have the names of two cities, along with the distance along the road, and these are set with the init initialization routine, which sets local instance attributes for city1, city2, and distance. Then, we have a set of accessor functions to get each city name, or the pair of city names, and the distance.

Now, here's how we might set up our node list and the edge list for the city example. We'll create our five cities, each with some population, and we'll add those to the city list. For example, we create a node for Rivertown with a population of 100 and we append it to the cities list. We'll also create our roads, five of them in this case, and we'll add them to the roads list. For example, we form an edge between Rivertown and Brookside of length 100, and then we append that edge to the roads list. It'd be a simple matter to add another road between two cities, or another city—just create a new edge or a new node and append it on.

Now, think about how you might use an edge. Say you wanted to find the total population that lives along a road. You'd need to find the population of the cities at each end. Population is stored in the city nodes; the road only knows the name of the cities. So, think about this: how could you write a function to find the combined population along an edge, such as edge 0?

Here's the basic process. You'd first get the road you cared about. I said we wanted edge 0, so I've just pulled out the first edge from the roads list and called it road. Then, we've got to loop through the list of

cities. For each city in the city list, we check whether the name of that city matches the first name or the second name from the road. If so, we store that population. And, finally, we add up the population. If you had a large list of cities, this might be pretty annoying, because it's an order *n* operation just to get the population.

Instead of using the names of the two cities, we could use the index of the two cities in the list of nodes. The index is just the position where they are in that list according to whatever order they were put into the list. Using the index makes it easier to find the cities—we can jump right to the city. We don't even need to change our node or edge classes; they'll still work exactly the same. The only difference is that the parameters to the edge class now have a bad name for the parameter. Instead of name1 and name2, we probably want to use a different name so that it's clear what the edge is expected to store. So, let's call them index1 and index2.

We change each parameter from a city name to an index, and we change some of the return functions to match this. To create our edges, we're now going to pass in the indices for the cities. So, for the road between Rivertown and Brookside, we'll use 0 and 1 instead of the city names, and the same will hold for the other cities.

Let's say that we want to get the population on a road now. What would the code look like in this case? This code is a whole lot simpler. We have the indices of the two cities, so we can go directly to that city in the list—there's no more loop that makes us search through each element of the list. This is a constant-time operation; it doesn't matter how many things there are, it always takes the same time. That's instead of the linear one that we had earlier, where the more things we had, the longer it would take. And our code to find this information is a whole lot shorter to write.

Of course, there's a downside to using the index instead of the name: the list of cities must remain unchanged. We can't have the city list being re-sorted or something—that would change which city a particular index was referring to. We need to know the index of each city in the list when

we set up our edge, and if we can't live with these restrictions, we can't use this approach, even if it's a lot faster for some operations.

Using an index instead of the name is a common trick in programming. Because it's less direct than storing the name, it's called indirection. Instead of storing the actual value—the name—we store an index to the thing containing the name. A dictionary is another way of getting a similar effect. Indirection also works well for the second way we could store graphs, called an adjacency list.

Let's say that we want to be able to take a given city and find all the roads from that city. That's a painful operation in the approach we've just described. We'd have to go through the list of edges trying to find the ones that have our city as the first or second city in the edge.

Instead of keeping a single list of all edges in the graph, let's instead keep just a local list of the edges for each node. Each node will keep track of the edges connected to it. In this case, we'll have just one list: a list of nodes. Each node will have the name of the city, other information about the node, like population, and then a list of edges. The edges in this case just need to keep track of the city at the opposite end of the road, and then the weight, such as the length of the road or the time it would take to travel that road.

Let's see what the code for this might look like. Now, you might want to try creating some code yourself, first. Here's one way to do it. We've defined an edge by just giving a single city and a weight value, rather than two cities like we had previously. Like before, we could have used an index instead of a name if we wanted. Our nodes will have a city name, a population, and then a list of roads. Each element of that list of roads will be an edge. We'll also have addNeighbor and addNeighborRoad defined. AddNeighborRoad takes an existing road as input, and it adds it onto the list of roads connected to this city, stored in the _roads list in the class. AddNeighbor will take in a city name and a weight for the edge as input parameters. It first creates a road to that city, with that weight, and then it appends that road onto the roads list.

We can create a set of cities, just like before—nothing changes in this part of the code for us. However, creating an edge in the graph, that's a connection between two cities, is more complicated. To make things easier, we can define a function, addRoad, that lets us add a road between two cities, and this function will take in the city names and the distance, and will create a road link for each city. It does this by creating new edge objects, one to each city. Then, it'll search through the list of cities to find the matching city name, and when it does so, it'll add the edge it created earlier to that city, by calling addNeighborRoad. So, in this way, both cities along a road get a local edge saying that they're connected to a neighbor. As an alternative, which is also shown here, we could call the addNeighbor routine for each city. Either way will work, and it will create a road in each city that links to the city on the other side.

Now, adding our connections is just a matter of calling the addRoad function that we just defined. Going back to our cities and roads example, the calls to add the five roads we defined would be just like you see here. This allows us to find neighbors much more easily. For instance, we can print out a list of all the neighbors, by city, with just a couple of loops. We first will loop through all of the cities, and then, for each one, we'll print out a line saying that we're printing out the list of neighbors, and then we get that list of neighbors and loop through the list. For each neighbor, we print out the name of the city and the distance away. So, in the end, we can find out all the neighbors of all our cities.

Notice that going through to find neighbors of any particular node was a whole lot easier this way. However, it did require a somewhat more complex set of routines to set things up. And, if we wanted to do something like find every edge, this data structure isn't as well suited as the previous one.

OK, so that was two ways of storing graphs: first, by keeping a global list of all edges; and second, by keeping a list of edges in each node. This second type is called an adjacency list. The global list of all the edges is probably most useful—no surprise—if you find yourself regularly needing to look at all the edges. That's the approach commonly used to represent geometric models, like you would have in 3-D graphics.

The second approach, the adjacency list, where you keep a list of the edges within each node, is useful in most of our typical graph operations, such as airline connections, social networks, and so on. The adjacency list works well because most graph algorithms are already designed to work just by looking at one node at a time and its neighbors.

There's also a third method to store graphs, and I won't dwell on it here, but it's called an adjacency matrix. In that one, instead of a list, there are matrix entries that note which nodes are connected. This can be useful for operations where you need to quickly run over lots of different values, or perform certain computations that can be expressed using linear algebra. It can be the most compact form of a graph, especially if there are a lot of edges, and this is a key factor when graphs are huge. It's also the fastest of the representations if you need to check whether a particular pair of cities is connected.

Now, for any given graph, we can define a variety of graph algorithms. Some of these algorithms are simple. For example, returning a list of all the cities adjacent to one city, similar to that algorithm that we just saw—well, that's a very basic algorithm. The representation used to store the graph will determine exactly how the algorithm will perform. Graph algorithms let us analyze all sorts of stuff about the structure of graphs. For example, the breadth-first search algorithm lets us analyze how many degrees of separation there really are between two people in a social network. We'll see the details of implementing breadth-first search in the next lecture.

Now, with roads, we can usually travel the road in either direction—we just say those nodes are connected. Graphs like this are called undirected graphs. A graph where people are friends is undirected: if personA is friends with personB, then personB is also friends with personA.

However, we can also have cases where two things are linked, but it's not an equal connection between the two sides. For instance, imagine cities connected by airline routes. Not all airline routes fly round trips; sometimes, a plane will fly from city 1 then to 2, then to 3, and then

back to 1. In this case, the edges go from one city to another, but not necessarily the other way around. So, there should be an edge from city 1's node to city 2's node, and from 2 to 3, and one from 3 back to 1. These are called directed edges, and the resulting graph, made up of directed edges, is called a directed graph, or sometimes a digraph. Web pages and the links between them form a directed graph. If web page A has a link to web page B, there's not necessarily one from B back to A.

Storing a directed graph is not much different than storing an undirected graph. If we have a big list of edges, then we just need to make sure that each edge has a start and an end—our earlier code doesn't even need to change at all. Each edge already had city1 and city2 attributes, so we don't need to change it.

For an adjacency list, we also don't need any fundamental change to the node or edge classes. Each edge is essentially already a directed edge—it's stored with the starting point. We already had to create two roads—one from A to B, and the other from B to A—for each road. So, for a directed graph, we just have edges stored on the one side where the edge starts.

Now, we say that a graph is connected if there's some sequence of edges that connects every pair of nodes, and a graph has a cycle if there's some way to follow a set of edges and end up right back where you started. In our city example, there's a cycle, since you could go from Rivertown, to Hillsview, to Brookside, and then back to Rivertown.

OK, let's see a directed graph in an application involving money. Now, different currencies can be traded from one currency to another, at some exchange rate. For any pair of currencies, there's some rate at which you can trade one into the other, and we can form a directed graph of these exchanges. It will be a really dense graph, because every currency can be exchanged for a different one. It's a weighted graph, also—each edge can give the rate of exchange for the first one into the second one.

Now, currency arbitrage refers to taking advantage of an imbalance in the currency exchange rates, so that you can make money through changing currencies. Here's how this would work. Let's say that you can change U.S. dollars to euros, getting 0.95 euros per dollar. Then, you can change euros into British pounds, getting 0.75 pounds per euro. Then, you can exchange 1 pound for 175 yen. And, finally, you can exchange 1 yen for .0085 U.S. dollars.

So, if you started with $1000, you could convert that to 950 euros, and then to 712.5 pounds, and then to 124,687 yen, and then to $1059.84, and that's more than you started with. Wow, it's like getting free money. And, believe it or not, imbalances like this do come up from time to time, and there are some people who've become very wealthy exploiting these imbalances. Now, if we store currency exchange rate as a directed graph, what we want to do is we want to find a cycle in which the product of all the edges is greater than one. And that would mean that we could use arbitrage to generate money.

OK, let's see how we would write code to do this. We first create classes to hold the currencies and the exchange rates between them. So, we'll have a currency class like you see here. The init function will set up a name and an empty list of exchange rates, both as instance attributes. We'll also have accessor functions to return the name of the currency and the list of exchange rates. Plus, there'll be two functions allowing us to set exchange rates: one, addRateEdge, will assume we've already created an edge that describes the exchange rate, and are passing it in; and the other one, addRate, will just take in the currency name and the exchange rate for that currency and it'll create the edge. In both cases, the edge simply gets appended onto the list of exchange rates.

We'll also have some functions designed to let us specify pairs of currencies, and then set up the appropriate edges for those currencies. This is very similar to how we set up roads between cities. Finally, we'll set up a group of currencies—in this case dollars, pounds, euros, and yen—and then will set up exchange rates between all pairs. Now comes the fun part—we want to find out if there is some conversion path that'll let us convert between these to make money. I'll quickly walk you

through some of this code, but you'll probably want to study it a while to really understand it.

First, I'll create a member function of the currency class called findArbitrage. We'll call this for a particular currency. It'll have a "for" loop for all the exchange rates in that currency. For each currency, it'll have another loop to find the particular currency in the overall list—that's the purpose of the inner "for" loop and "if" statement. We then call findRates for that currency.

FindRates is going to be our main routine; it'll make recursive calls to itself to find whether some chain exists. It takes in the main currency that we're trying to find a cycle for as one parameter, and then the current combined exchange rate, and then how many conversions have already occurred. First, it handles two base cases for recursion. The first checks to see if we're back at the main currency that we care about. If we are, we see that we have a cycle, and we check to see if it made us money by checking whether the current_value is greater than one. If so, we print out that we've found a way, and we show what that combined exchange rate is, and we return true. We also have a second base case where we've had too many exchanges. If we get to this point, we just quit and return false. In this case, I've said that we can't have more than three exchanges, so this checks whether the depth is greater than three.

The other part of the routine goes through all the conversion rates possible from this currency, using a "for" loop. It computes the overall conversion for that rate, then it finds the currency that this corresponds to, and that's the purpose of the inner "for" loop and the "if" statement. Once it finds that, it recursively calls itself within the "if" statement. The call to itself passes in the main currency, the new combined exchange rate, and it increments the number of exchanges so far by one. Now, if that returns true, it means that there was a cycle to this point, so it prints out the combined exchange rate, and its currency name, and then it returns true itself. But, if we got through all the rates and we didn't already return true, it means that we didn't find a cycle through this exchange, so nothing printed and we just return false.

Now, finally, if we run this code with a simple call to findArbitrage on the first currency, we do indeed find a cycle. For the exchange rates that I put in, we can convert 1 dollar to 0.7 pounds, then to .9333 euros, then to 217.7 yen, and then finally to $1.85. Wow, that's a big improvement. Now, obviously, these are not real exchange rates. If they were, we could use this information to make ourselves very, very wealthy.

All right, what if we have a graph that's connected but doesn't have a cycle? A connected, undirected graph that does not contain a cycle is called a tree. Trees are such a useful structure that a whole set of algorithms have been developed just for trees.

So, here's an example of a tree. All the nodes are connected to each other, and there's no cycle in the graph. Now, usually, when we talk about trees, we'll designate one node as the root. The root of a tree can be thought of as the starting point, or the central point—it's the top level of a hierarchy. And the rest of the tree can then be arranged in terms of levels from the root, where the root is at level zero, and each subsequent level is formed based on how many edges must be followed to get to the root. All the nodes connected to the root are considered its children, and they form the first level; the nodes that they're connected to form the second level, and so on. For any node, the node it's connected to at the previous level is called its parent, and the nodes below it are called its children.

Now, looking back at our tree graph, say that the highlighted node was our root—our zero. Then, the other nodes would be at level 1, 2, or 3. In a typical drawing of a tree, we'll put the root at the top, and the children in levels below. Now, because trees have this particular structure, they often have a particular way that they're stored in code. Trees almost always use an adjacency list, where the edges are stored inside each node, and they usually store the parent and the children separately. So, any one node will store its own information, along with an index or some other notation—like a name for the parent node—along with a list of its children. In code, we'll keep a variable to give the parent, and we'll keep a list of variables that are the children. We can add additional children whenever we need to.

A very common type of tree is called a binary tree. In this case, every node has no more than two children. We'll usually call them a left child and a right child. So, a node will hold a parent, a left child, and a right child. Binary trees have lots of uses, but a common one is to store objects in sorted order. We call these binary search trees. Unlike arrays or lists that we might have to sort every time we add a new value, a binary tree can keep the items always in sorted order. As you might guess, it's usually faster to add an item to a binary tree than to add it to an array or a list that you have to keep sorted. As you can see in this example of a binary search tree, at any node in the tree, all the descendants on the left-hand side are less than the node, and all those on the right-hand side are greater than the node. Notice, for instance, that 21 is greater than the root, 15, so it's on the right-hand side of the root. It's greater than the next node, 18, so it's also on the right-hand side of it. But, it's less than the next node, 23, so it's on the left-hand side of that one.

Let's see how this works in code if we want to add a new item to the binary search tree. As with many tree and graph algorithms, we're going to use a recursive definition. The basic idea is to call insert on the root of the tree, and recursively work our way down through the tree, until we add the node at the right point. So, we start by comparing it to the value at the root. If our new node is less, we want to go to the left, and if it's greater, we want to go to the right. Whenever there's not another node to go to, then we can make this new node at that position. So, each time, we'll either set this node as the left or right node, or else we'll call insert recursively.

Let's see a quick example of how this works before we look at code. Say we have our tree from before, and we want to insert the number 7. Seven is less than 15, so we go to the left. It's less than 8, so we go left again. And it's greater than 3, so we go right. And, there's nothing already to the right, so we insert 7 there—that is, we set 7's parent to be 3, and we set 3's right child to be 7.

So, let's try writing the code for this insert routine. Here's the node class that we start with. It's the same as the binary tree node that I showed

before, but I've changed the name to a value. We can also define a set of nodes, like you see here, and our goal will be to take that set of nodes and turn it into a binary search tree by inserting one node at a time. So, how would we insert one node at a time? Remember, you'll want to write a recursive function called insert that takes in one node index and adds it to the tree.

Our code for the insert routine will be like this. We have "insert," and it takes in the index of the node to insert. The first thing it does is compare the value of the node being inserted to the current node's value. That is, it gets the node by taking nodelist sub insert_node, and it calls getValue on that node to get that node's value. It gets compared to the val part of the current node. If it's less, we'll work on the left-hand side; otherwise, we'll work on the right-hand side. If we're on the left, we check to see if there's anything on the left. If the index of the left node is minus 1, that means there's no node there. So, we set the left node to be this one, and we're done. Otherwise, there is a node on the left already. So, we find that node by going to nodelist sub self_left, and we call insert on it, passing in the insert node. Now, this recursive call will insert the node somewhere on that left side. The right side of the tree works the exact same way.

Now, if we have a bunch of nodes that we want to insert into a tree, here's how that'll look. We set the first node as the root. Then we go through the remaining nodes, inserting them one by one at the root. Each of these will add the node into the tree; for the values that we had earlier, inserting them in this order will form the tree that we saw previously.

Having this binary search tree is nice, and there are various things that we can do with it. One simple thing we can do is print out the entire tree in sorted order. To print the tree in sorted order, think of how to handle the root node. First, we would want to print the left branch; then, we'd print the current value; and then, we would print the right branch. And this is true for any node, not just for the root—we always want to print the left side, then the current value, then the right side. So, let's try writing code to do this.

To code for an "in order" printing, I called the routine printInOrder. We'll first check to see if there's a left node; if so, we recursively call printInOrder on that side. Then, we print the current value. And, finally, we check to see if there's a right node, and if so, we recursively call printInOrder on that side. Notice that what we've got here is basically another sorting routine. We're putting elements into the binary search tree in order, and then, when we print them out, we get a sorted list. Sure enough, if we take our set of nodes, insert them all into the tree, and print it back out, we get a sorted output list. Now, compared to the sorting routines that we saw previously, this one is, on average, as fast as that *n* log *n* mergeSort algorithm that we saw, plus it works really fast, on average, if you have to update the sorted list.

Graphs offer a structure for capturing an incredibly wide diversity of relationships, and the connections that are formed all over. They represent connections between locations, such as airline routes between cities; connections between people, such as friends in a social network; connections among species, like in a food web; and connections among data, like the hyperlinks in web pages. These and many other relations are well represented on a graph. Once relationships are captured in a graph, it's algorithms that let us use the graph effectively in our programs.

In the next lecture, we're going to look at a famous graph algorithm called breadth-first search, and we'll use that to create a fun word game. I'll see you then.

# 23

# Graph Search and a Word Game

The fundamental graph algorithm you will learn about in this lecture is called the "breadth-first search." Searching through a graph to connect a starting node with another node can take one of two approaches: either follow edges as far as you can until you can go no farther, or gradually spread out from the starting point. The breadth-first search, as the name suggests, takes the latter approach, which is more balanced.

## [ BREADTH-FIRST SEARCH ALGORITHMS ]

- Let's imagine that we have a social network, where people form the nodes, and we have an edge connecting people if they're friends with each other. If we want to know the shortest way to connect two people through a sequence of friends, we can find that out through a breadth-first search.

- In Figure 23.a on the following page, the node marked "0" is the node we're starting at, and the striped node is the one we're trying to find. In our social network, node 0 is the first person and the striped node is the person we want to connect to. People who are "friends" in the social network are represented as "neighbor" nodes in the graph.

- We'll number the nodes as we find them, and we'll keep a list of nodes, visiting them in order. The first thing we do is look at all of the neighbors of node 0. It has 3 neighbors, so we number them 1, 2, and 3. We didn't find the goal among them, so we are done with node 0.

- Now we move on to the next node, node 1. We will check its neighbors, numbering them. It has 4 neighbors, but one of those, node 0, already has been seen, so we don't number it. We check nodes 4, 5, and 6, giving them numbers. Then, we're done with node 1.

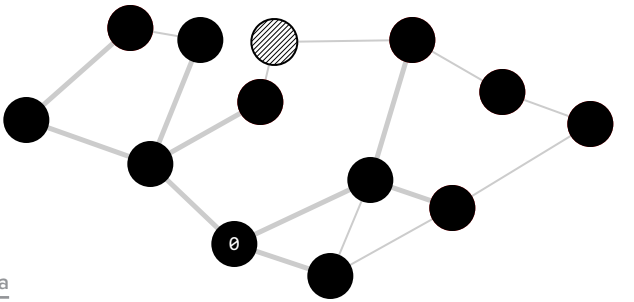



Figure 23.a

- We move on to node 2, which also has 4 neighbors, but two of them, nodes 0 and 3, have already been seen. So, we look at the two remaining neighbors, giving them numbers.

- Next is node 3. All of node 3's neighbors have been seen, so we have nothing else to do there.

- The process continues with node 4, and then with node 5. Finally, when we get to node 6, we have found our result. It turns out that there is a path from node 0 to 1 to 6 to our goal node. In other words, the person represented by node 0 has a friend who is friends with the friend of the person represented by the striped node.

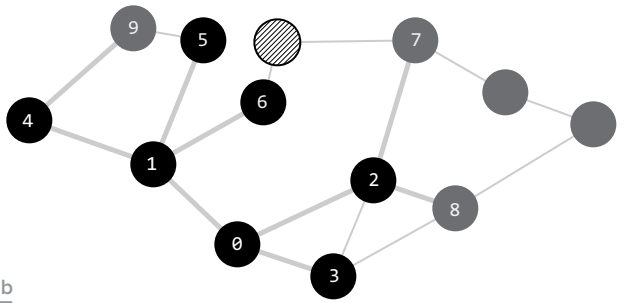



Figure 23.b

- With breadth-first searches, the general idea is that we visit a node, check all of its neighbors, and put them in a list to visit next if they haven't been visited. If we want to get that path out at the end, we need to keep track of each node along the way.

- Let's see how we would implement that algorithm in code. First, we'll write some pseudocode for the algorithm.

- To run a breadth-first search, we will need some additional information at each node. We'll want to classify each node as either "unseen" or "seen." It will start as "unseen," and when we first discover it, we'll mark it "seen." Also, we'll want to keep track of the previous node that was used to "see" this node for the first time so that we can get a list of edges out at the end.

- Next, we'll keep a queue of which nodes to visit. A queue is "first in, first out." We can implement a queue with just a list, and we have an "enqueue" to add something to the end of the queue and "dequeue" to take the next one from the front. Initially, the only node in our queue will be the starting node.

- Then, we'll go through and visit the nodes. Each time, we'll take whichever one is next in the queue. We can then check its neighbors. For each neighbor, we'll ignore it if it's already seen or visited—that means we've already discovered that node and don't need to worry about it again.

- However, if it's unseen, we will mark that neighbor as "seen" to make sure it knows that it was discovered from whatever node we are on now. We need to check to see if we've found the goal node, and if so, we can stop our loop. Otherwise, we need to add this node to the queue.

- At the very end, we will need to go back over the path that we found, by following the list of which node discovered the goal, then which one discovered that one, and so on until we're back at the start.

```
Input: A graph, a starting node, S, and a goal node, G
Output: A path of edges between G and S

1.  Add information to each node:
    a.  Unseen/seen - initialize to "unseen"
    b.  Previous node in BFS - initialize to none
2.  Initialize queue of nodes to visit with S, and initialize S
    to "seen"
3.  While goal is not found and queue is not empty:
    a.  Get next node in queue
    b.  Check all neighbors. If neighbors are "unseen":
        1.  Mark neighbor as "seen"
        2.  Set neighbor's previous node to current node
        3.  See if the neighbor is the goal, if so, exit the loop
        4.  Add this node to queue
4.  If goal was found:
    a.  Create list of edges by following "previous node."
```

› Given this pseudocode, let's see how we would implement it in actual code. Let's assume that we still are working with the social network. We'll have nodes for people, and each person will have a list of friends—that is, its neighboring nodes. We'll assume an unweighted graph, so our "edge" is just going to be a number of another node. This means that we don't even need an edge class—each node can just keep a list of neighbors, which are the indices for the neighbors.

› And we can have our "makeFriends" routine that lets us link two people together as friends. It will go through the "people" list to find the index of each of the names of the two people, and then add the corresponding index to the friend list of each.

```
class node:
    def __init__(self, name):
        self._name = name
        self._friends = []
```

```
    def getName(self):
        return self._name
    def getFriends(self):
        return self._friends
    def addFriend(self, friend_index):
        self._friends.append(friend_index)
def makeFriends(name1, name2):
    for i in range(len(people)):
        if people[i].getName() == name1:
            n1 = i
        if people[i].getName() == name2:
            n2 = i
    people[n1].addFriend(n2)
    people[n2].addFriend(n1)
```

› In order to run the breadth-first search, we'll need to augment our node structure, to have the additional features of being able to mark a node as "seen" or "unseen" and note the previous node on the breadth-first search path.

› We'll use a number to designate "unseen" and "seen"—0 and 1, respectively. We'll keep this in a local variable, "status," that we set to 0 on initialization. We'll also provide routines that let us set the status to "Unseen" or "Seen," and we'll provide routines that give us a "True" or "False" as to whether it's seen or not.

```
class node:
    def __init__(self, name):
        self._name = name
        self._friends = []
        self._status = 0
...
    def isUnseen(self):
        if self._status == 0:
            return True
        else:
            return False
```

```
    def isSeen(self):
        if self._status == 1:
            return True
        else:
            return False
    def setUnseen(self):
        self._status = 0
    def setSeen(self):
        self._status = 1
```

› We also need to augment the node so that it has some information about which node discovered it during the search. We'll add a routine that lets us set that node. Later, we'll need to do some more with this when we print out our result, but that's enough to be able to write our basic breadth-first search routine.

```
class node:
    def __init__(self, name):
        self._name = name
        self._friends = []
        self._status = 0
        self._discoveredby = 0
    def getName(self):
        return self._name
    def getFriends(self):
        return self._friends
    def addFriend(self, friend_index):
        self._friends.append(friend_index)
    def isUnseen(self):
        if self._status == 0:
            return True
        else:
            return False
    def isSeen(self):
        if self._status == 1:
            return True
```

```
        else:
            return False
    def setUnseen(self):
        self._status = 0
    def setSeen(self):
        self._status = 1
    def discover(self, n):
        self._discoveredby = n
    def discovered(self):
        return self._discoveredby
```

- At this point, we've augmented our nodes to hold the required information. We'll now actually write a function to do this search for us. We need to take in the graph, which is going to be our node list, a starting node, and a goal node. The nodes are just the index of the node.

- Let's see how we start our routine. We'll first have our queue routine—exactly the same as the one we developed previously. Then, we have the beginning of our breadth-first search (BFS) routine. The BFS function will take in a "nodelist," a start, and an end.

- The first thing to do was to mark the starting node "seen" and add it to the queue. So, within our routine, we create a new, empty queue, called "to_visit." We then mark the starting node as visited by going to nodelist[start]—the starting node—and calling "setSeen," which will mark it as "seen." We then add the start node to the queue, by calling "enqueue."

```
class queue:
    def __init__(self):
        self._queue = []
    def enqueue(self, x):
        self._queue.append(x)
    def dequeue(self):
        return self._queue.pop(0)
    def isEmpty(self):
        return len(self._queue) == 0
```

```
def BFS(nodelist, start, goal):
    to_visit = queue()
    nodelist[start].setSeen()
    to_visit.enqueue(start)
```

› The next thing to do is create a loop where we pull out the next node and visit its neighbors. At the beginning, we need a variable to keep track of whether we've found the goal—that'll be "False" at first. We'll then have to have our loop, which will be a while loop with two conditions: the goal was not found, and the queue is not empty.

› Now we have a loop, and the first thing we need to do is get an item out of our queue, and then get its neighbors. We'll call "dequeue" to get the next node out of the queue—the index of the next node. We'll call the index that we pulled out "current." For that node, we need to pull out the neighbors, which we can do with a single function call. We just call "nodelist[current]" and then call the "getNeighbors" method on that, which returns a list of neighbors to visit.

```
def BFS(nodelist, start, goal):
    to_visit = queue()
    nodelist[start].setSeen()
    to_visit.enqueue(start)
    found = False
    while (not found) and (not to_visit.isEmpty()):
        current = to_visit.dequeue()
        neighbors = nodelist[current].getNeighbors()
```

› First, we have a for loop, which will go through all of the neighbors, so we write "for neighbor in neighbors." We next check to see if the node is an unseen one. If it's not, we don't need to think about it. If it is unseen, we change that. Using the "setSeen" command and the "discover" command, we mark the node as "seen" and with a prior node of the current one.

› Finally, we check to see if we have found the goal by directly comparing the neighbor with the goal. If so, we mark "found" as "True," which will stop this loop the next time around. If not, we add this neighbor onto our queue of nodes to check.

```
def BFS(nodelist, start, goal):
    to_visit = queue()
    nodelist[start].setSeen()
    to_visit.enqueue(start)
    found = False
    while (not found) and (not to_visit.isEmpty()):
        current = to_visit.dequeue()
        neighbors = nodelist[current].getNeighbors()
        for neighbor in neighbors:
            if nodelist[neighbor].isUnseen():
                nodelist[neighbor].setSeen()
                nodelist[neighbor].discover(current)
                if neighbor == goal:
                    found = True
                else:
                    to_visit.enqueue(neighbor)
```

› If the goal was found, we need to find the list of nodes that got us to the goal. Because each node along the way from the start to the goal has a reference of who discovered the node, we can just read this list backward to get our result.

› Let's assume that we do this through a function call—to a function named "retrievePath." There are several ways to write this.

```
def retrievePath(nodelist, start, goal):
    #Return the path from start to goal
def BFS(nodelist, start, goal):
    to_visit = queue()
    nodelist[start].setSeen()
    to_visit.enqueue(start)
    found = False
```

```
    while (not found) and (not to_visit.isEmpty()):
        current = to_visit.dequeue()
        neighbors = nodelist[current].getNeighbors()
        for neighbor in neighbors:
            if nodelist[neighbor].isUnseen():
                nodelist[neighbor].setSeen()
                nodelist[neighbor].discover(current)
                if neighbor == goal:
                    found = True
                else:
                    to_visit.enqueue(neighbor)
    return retrievePath(noswliar, start, goal)
```

› One way to implement this is to use a recursive approach. We start out seeing if we need a path for just one node—that is, if the start and the goal are the same. In that case, we create a list containing just the start. We set up an empty list, called "path," append the start value on, and return it. Otherwise, we will find the previous node that comes right before the goal.

› We recursively get the path from the start node to that previous node—that is, we call our retreivePath function using that previous node as the goal. Then, we just append the goal onto the end of that, and return.

```
def retrievePath(nodelist, start, goal):
    #Return the path from start to goal
    if start == goal:
        path = []
        path.append(start)
        return path
    else:
        previous = nodelist[goal].discovered()
        previous_path = retrievePath(nodelist, start, previous)
        previous_path.append(goal)
        return previous_path
```

› Once we've finished our breadth-first search algorithm, we can test it, using a small graph for five people who have several friend relationships. When we run this, we get a list—John, Sue, Fred, Kathy—which is indeed a path connecting the two friends.

```
people = []
person = node('John')
people.append(person)
person = node('Joe')
people.append(person)
person = node('Sue')
people.append(person)
person = node('Fred')
people.append(person)
person = node('Kathy')
people.append(person)
makeFriends('John', 'Joe')
makeFriends('John', 'Sue')
makeFriends('Joe', 'Sue')
makeFriends('Sue', 'Fred')
makeFriends('Fred', 'Kathy')
pathlist = BFS(people, 0, 4)
for index in pathlist:
    print(people[index].getName())

OUTPUT:
John
Sue
Fred
Kathy
```

› If you've ever heard of a "Bacon number," in which you try to connect actors who have acted in movies together, all the way to a connection to Kevin Bacon, this is the algorithm used to determine that. If we could form a graph of everyone in the world, with a link between people who know each other, we could use this algorithm to check the claim that any two people are separated by only six degrees of separation.

## Reading

Lambert, *Fundamentals of Python*, chap. 12.

## Exercise

There are many other graph algorithms besides breadth-first search (BFS). One of these is depth-first search (DFS), which aims to explore as far as possible.

Imagine that the queue used to keep track of nodes to visit in the BFS algorithm is instead replaced by a stack. Assume that you had the following graph and were starting at node E, trying to find node G. Assume that neighbors are listed in alphabetical order in each node. What is the order in which the nodes are visited?

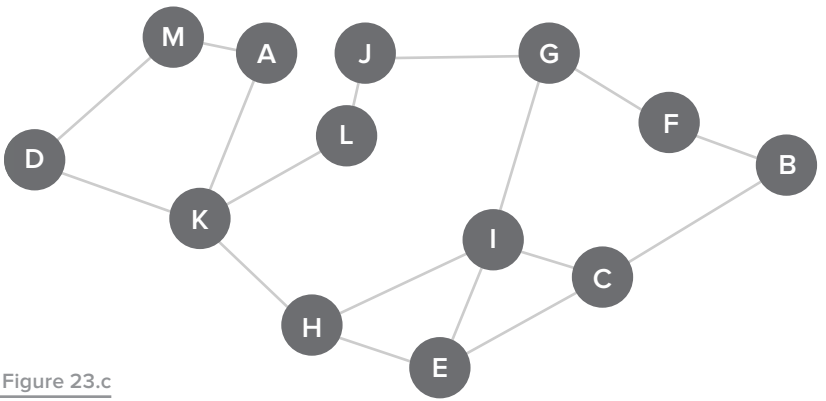



Figure 23.c

# Graph Search and a Word Game

Let's learn an algorithm and build a game. Pulling together several of the topics that we've discussed previously, we're going to build a word game based on a graph. The fundamental graph algorithm we're going to use is called the breadth-first search.

Searching through a graph to connect a starting node with another node can take one of two approaches. You can either follow edges as far as you can go until you can go no farther, or gradually spread out from the starting point. Breadth-first search, as the name suggests, takes the latter approach, which is more balanced. Let's imagine we have a social network, where people form the nodes, and we have an edge connecting people if they're friends with each other. If I want to know what's the shortest way to connect two people through a sequence of friends, I can find that out through a breadth-first search.

Look at this graph, since it will illustrate the process. Let's say that the green node is the node that we're starting at, and the yellow node is the one that we're trying to find. In our social network, the green node is the first person, and the yellow is the person we want to connect to. People who are friends in the social network are represented as neighbor nodes in the graph.

We'll number the nodes as we find them, and we'll keep a list of nodes, visiting them in order. Once we've finished a node, we'll color it black. It'll be a whole lot easier to understand if you see it, so let's do that now. We start with node 0, our starting node. The first thing we do is we look at all of its neighbors. It has three neighbors, so we number them 1, 2, and 3. We didn't find the goal among them, so we are done with node 0.

Now we go on to the next node, node 1. We will check its neighbors. It has 4 neighbors, but one of those, node 0, already has been seen, so

we don't number it. We check nodes 4, 5, and 6, giving them numbers. Then, we're done with node 1.

On to node 2. Node 2 also has four neighbors, but two of them, nodes 0 and 3, have already been seen. So, we look at the 2 remaining neighbors and give them numbers. Next is node 3. All of node 3's neighbors have been seen, so we have nothing else to do there. The process will continue with node 4 and with node 5, and finally, when we get to node 6, we have found our result. It turns out that there is a path from 0 to 1 to 6 to our goal node. In other words, the green person has a friend who is friends with yellow's friend.

So, that's the general idea. We visit a node, check all of its neighbors, and we put them into a list to visit next if they haven't been visited already. If we want to get that path out at the end, we'd need to keep track of each node along the way. Let's see how we would implement that algorithm in code. First, we'll write some pseudocode for the algorithm.

To run a breadth-first search, we will need some additional information at each node. We'll want to classify each node as either unseen or seen. It will start as unseen, and when we first discover it, we'll mark it as seen. Also, we'll want to keep track of the previous node that was used to see this node for the first time, so that we can get a list of edges back out at the end.

Next, we'll keep a queue of which nodes to visit. A queue is first in, first out. We can implement a queue with just a list, and we have an enqueue to add something on to the end of the queue and a dequeue to take the next one from the front. Initially, the only node in our queue will be the starting node.

Then, we'll go through, and we'll visit the nodes. Each time, we'll take whichever one is next in the queue. We can then check its neighbors. For each neighbor, we'll ignore it if we've already seen it—that means we've already discovered that node, and we don't need to worry about it again. However, if it's unseen, we will mark that neighbor as seen, and

then we'll make sure it knows that it was discovered from whichever node we are on right now. We need to check to see if we've found the goal node and if so, we can stop our loop. Otherwise, we need to add this node to the queue. At the very end, we will need to go back over the path that we found, by following the list of which node discovered the goal, then which one discovered that one, and so on until we're all the way back at the start.

OK, given that pseudocode, let's see how we would implement that in actual code. Let's assume that we're working with the social network. We're going to start out with pretty much the same operations that we had in the adjacency list way of storing graphs that I showed you earlier when we were using cities and roads. That is, we'll have nodes for people, and each person will have a list of friends, that is, its neighboring nodes. We'll assume an unweighted graph, so our edge is just going to be a number of another node. That means that we don't even need an edge class—each node can just keep a list of neighbors, which are just the indices for its neighbors. And, we can have our makeFriends routine that lets us link two people together as friends. It will go through the people list to find the index of each of the names of the two people, and then it'll add the corresponding index to the friend list of each.

In order to run breadth-first search we'll need to augment our node structure, to have the additional features I mentioned earlier—being able to mark a node as seen or unseen, and noting the previous node on the BFS path. We'll use a number to designate unseen and seen—just 0 or 1. We'll keep this in a local variable, status, that we set to 0 on initialization. We'll also provide routines that let us set the status to Unseen or Seen, and we'll provide routines that give us a true or false as to whether it's seen or not. We also need to augment the node so that it has some information about which node discovered it during the search. We'll add a routine that lets us set that node. Later, we'll need to do some more with this when we print out our result, but that's enough to be able to write our basic breadth-first search routine.

So, at this point, we've augmented our nodes to hold the required information. We'll now actually write a function to do this search for

us. We need to take in the graph, which is going to be our node list, a starting node, and a goal node. The nodes are just the index of the node. So, let's see how we start our routine off. We'll first have our queue routine—exactly the same as the one we developed a while back. Then, we have the beginning of our breadth-first search routine or BFS. The BFS function will take in a nodelist, a start, and an end.

Now, the first thing to do was to mark the starting node seen and add it to the queue. So, within our routine, we created a new, empty queue, called to_visit. We then marked the starting node as visited by going to nodelist[start]—that's the starting node, and calling setSeen, which will mark it as seen. We then add the start node to the queue, by calling enqueue.

The next thing to do is to create a loop where we will pull out the next node, and we'll visit its neighbors. For this part, see how much of this you can write, before watching, then again after watching. I'll go through this slowly so you can stop and see a little bit, for when you get stuck.

OK, at the beginning, we need a variable to keep track of whether we've found the goal—that'll be False at first. We'll then have to have our loop, which will be a while loop with two conditions. One is that the goal was not found, and one is that the queue is not empty. So, now we have a loop, and the first thing we need to do is get an item out of our queue, and then get its neighbors. See if you can continue writing from here.

OK, we have our loop, and we need to get the next node out of the queue. So, we'll call dequeue to get the next node, and by that I mean the index of the next node. We'll call that index that we pulled out current. Now, for that node, we need to pull out the neighbors, which we can do with a single function call. We just call nodelist[current], and then call the getNeighbors method on that. That method returns a list of neighbors to visit. Remember the steps to do with the neighbors? Let's try coding what needs to be done at this point.

OK, first, we have a for loop, The for loop will go through all of the neighbors, so we write: for neighbor in neighbors. We next check to

see if the node is an unseen one - if it's not, we don't even need to think about it. If it is unseen, we change that—using the setSeen command and the discovered command, to mark the node as seen and with a prior node of the current one. Finally, we check to see if we have found the goal, by directly comparing the neighbor with the goal. If so, we mark found as true, which will stop this loop the next time around. If not, we add this neighbor onto our queue of nodes to check.

OK, there's one more part to code. If the goal was found, we need to go ahead and find the list of nodes that got us to the goal. Since each node along the way from the start to the goal has a reference of who discovered the node, we can just read this list backward from the goal back to the start to get our result. Let's assume we do this through a function called retrievePath. We need to write retrievePath. There are several ways we could write it—see what you can come up with.

OK, one way to implement that is to use a recursive approach. We start out seeing if we need a path for just one node—that is, if the start and the goal are the same. In that case, we create a list containing just the start. We set up an empty list, called path, and we append the start value on, and we return it. Otherwise, we will find the previous node that comes right before the goal. We recursively get the path from the start node to that previous node. That is, we call our retreivePath function using that previous node as the goal. Then, we just append the goal onto the end of that, and we return.

So, we've finished our breadth-first search algorithm. We can test this out, using a small graph that I created for five people who have friend relationships. In this case, we call BFS with a start of 0 and a goal of 4. That's the first person, John, and the last person, Kathy. We'll take that result, and we'll print the names of the friends in order. So, when we run this, we get a list, John, Sue, Fred, Kathy—which is indeed a path connecting the friends.

If you've ever heard of a Bacon Number in which you try to connect actors who have acted in movies together, all the way to a connection to Kevin Bacon—this is the algorithm used to determine that. If we could

form a graph of everyone on earth, with a link between people who know each other, we could use this algorithm to check the claim that any two people are separated by only six degrees of separation.

So, given this graph algorithm, let's see how we can use it for a completely different type of application, now for something completely different. There are lots of fun brain teaser puzzles out there, and I bet you've played with many of them. Let me tell you about one of these puzzles that's always been fun for me, even though I haven't always been that good at it. One of the things that I think is so interesting is that this is a problem that can be quite challenging for people, but it's a very straightforward puzzle for a computer to solve.

The game is a word game where we try to transform one word into another one. The key is that you can change only one letter at a time, and with every change, you still need to have a valid word. For example, say that you wanted to change the word car to the word let. Can you come up with a sequence of words, each one different from the previous by only one letter, that gets you from car to let?

Well, here's one option. We could change the *r* in car to a *t*, ending up with cat. Then, we can change the *c* in cat to a *b*, ending up with bat. From there, we can change the *a* to an *e*, ending up with bet. And finally, we change the *b* to an *l*, and we have gotten to our goal, let. Now, there were other ways to get there, but this was one way. What we're going to develop is a way for the computer to play this game. We'll give the computer a starting word and an ending word, and the challenge will be to let it come up with a chain of words to get from one to the other. This is also going to help us see ways to tie together lots of the things that we've encountered so far. Let's start with the overall design.

We can assume that we'll be given a starting word and an ending word, and we'll produce a chain of words. This should sound a whole lot like the social network example that we just had. We have a graph, with a starting node, and an ending node, and a chain between them. That's what we have here, also, so we should think about how this problem

could be modeled as a graph. And then, all we'll need to do is run our breadth-first search algorithm on that graph.

Obviously, the nodes of the graph need to be words. For there to be an edge between two words, there should be exactly one letter different between them. If the words differ by more than one letter, then there's no edge. The graph will be unweighted, and undirected.

So, we have a rough outline of the program. We'll need to set up the graph itself. Then, we can ask the user for a starting and ending word, run our breadth-first-search to find a chain, and output the results.

To set up the graph, we'll first need to read in a set of words that are valid words. We'll then need to form a graph from those words. To form a graph, we'll compare words, and if they differ by exactly one letter, we'll add an edge between them.

Now, we need to stop and think for a minute before we write any code. We're going to read in a long list of words, and then we're going to need to compare every pair of words to see if they differ by exactly one letter. Let's think about what the running time for that will be. If we read in *n* words, we compare the first one to n-1 other words. Then we compare the next one to n-2 other words. The net result of all of this is going to be something that's order n-squared.

So, if we have about 1000 words, which is actually pretty small, then we'll have on the order of a million, well actually half a million, comparisons. That's a lot of comparisons. And, to make it worse, 1000 is not a lot of words. The English language has hundreds of thousands of words. If we were to have a word list containing a regular dictionary, we might never finish building the graph.

So, we need to limit the number of pairwise comparisons by limiting the set of words, staying within a number that our n-squared process can find in a reasonable amount of time. Computers these days are fast, and doing a million comparisons, or even a few million, is not unreasonable. But, to do tens of millions can start becoming noticeably slower. So, let's

limit our word list to no more than a few thousand words. If we have too many words to compare, we could spend our whole day just waiting for the graph to build.

The first thing that we need to do is define the data structures and functions needed for our graph. A node in the graph should represent a word, so we'll store a word in each node. Since we're going to run a breadth-first search, we'll need to keep track of a discovered by link, and a seen/unseen value, as well. The code will look a whole lot like the code we had in our earlier breadth-first search on the social network. So, we're going to start with that code, and just modify it as necessary.

Here's our class. It's identical to the node class that we were just using in the breadth-first search example, with one difference. We're storing the value for each node using the term Word instead of Name since each node is a word rather than a person. Again, everything else—getting and adding neighbors, marking things as seen and unseen is identical.

Our addLink routine will be similar to the previous one, but will have some differences. Remember that when we joined friends before we would form an edge an edge between a pair of nodes. We'll create two versions. One, much like the previous one, will take in two words we want to connect. We'll call this addWordLink. AddWordLink will take in the list of words and the two words that we want to link. We'll go through the list of words to find the index for each of the ones that we're looking for. An alternative will be a routine that we just call addLink that we assume takes in the index of the two things we are linking. The parameters passed in will be the word list, as well as the indices of the two that we need to form the edge between. In this case, we don't need to search for the words to find the indices, so the code is significantly shorter.

The remainder of our breadth-first search is unchanged. Our queue structure is no different than when we first introduced it. Retrievepath is exactly the same. And, BFS, the breadth-first search algorithm, is also exactly the same. None of those rely on the actual values in the nodes, just their connectivity. So, there's not a single thing that needs to change for our graph of words versus our social network graph. That's great.

So, let's go through the process of setting up the game. We'll first need to read in a set of words, and we'll be reading this in from a file. As we mentioned before, we need to be careful about the size of the file—that is, the number of words—in this list. We could limit ourselves to only certain categories of words, for example, words with 3 letters, or 4 letters or something like that. We could limit ourselves to the most common English words, or any of a variety of choices. You can do a quick search on the web and find many different lists of words. I'd recommend finding one that's of reasonable size—say no more than about 3000 words, and saving it in a file.

I've found a list of common English words, about 2000 of them, in order of frequency, and saved it into a file, called dictionary.txt. There's one word per line in the file. So, our first task is to read in the file and create our word list in memory. We'll create one node per word, and store it in a list of nodes. What would the code for this look like?

Well, first, we'll have our list of nodes that we initialize to an empty list. In this code, we'll call that node list words. Next, we open up our file that contains all the words in it for reading. I'll call that dict, short for dictionary, and open the file dictionary.txt for reading. Next, I'll go through every line of dict using a for loop. Remember that there's one word per line of the file, so for each line, I want to get that word, create a node, and add that node to the list of nodes. The way we do this here is to create a new node, called word, where we take the line, we strip away any whitespace—that's the strip() command. Once we've created that node, we append it onto words list—that's our list of nodes.

So, at this point, we've read in a bunch of words and we've turned them into nodes. Now, we'll need to actually form the graph. Remember, we'll want to check every pair of words, and form a link if there is just one letter different between them. To do this, we probably need a function that will let us compare two words, and return true if, and only if, they are different by just one letter.

Let's write our code for that function. The function needs to take in two strings, that is, the two words, and return true if they're different

by only one letter. What do you think this code should look like? Here, we've written a function called compareWords. It first checks to see if the words are the same length, and if not, it returns False, since they can't be different by only one letter. Otherwise, it will go through a for loop, letting the index go from 0 to the length of one of the two words. It keeps track all along the way of how many letters are different through a numdifferent variable. For each index, it compares the letters at that index and then it increments numdifferent if they're different by one. After doing this, if the number of different letters is exactly one, we return True; otherwise, we return False.

Now, given that code, how would we form the graph itself? Remember, we need to compare every word to every other word, and add a link if they're different by just a single letter.

OK, what we're going to do is loop through all of the words in the list, and compare to each of the other words in the list. So, we will have nested for loops. The outer for loop will use an index, *i*, to go through each word in the list. The inner for loop will then go from *i* plus one to the end of the list. Notice that the inner loop doesn't go over every item, just the following ones, since the earlier items would have already been compared. For each of those pairs, we will compare the two words, calling compareWords that we just wrote. If the comparison is True, that is, if they differ by just one letter, then we add an edge to the graph by calling addLink, with the *i* and *j* indices passed in.

At this stage, we've built our entire graph. Yay. The next thing in our program is to get the words that we want to find a chain between—the starting and the ending words. This can be done with just a couple of lines of code. So, now comes the main event—actually finding a chain between these two words. This is where our breadth-first search comes in.

First, remember how our BFS routine was defined. The BFS takes in the list of nodes, the index of the starting node, and the index of the ending node. It's the index of each node that we pass in as a parameter. We don't have that right now—we just have the word itself. So, we need to find the index of each of those words in the list of all words.

So, given that we have two strings, word1 and word2, how would we find the indices in the list of nodes, that is the list of nodes, that is the list words? We'll loop through all of the nodes in our nodelist and find indices that matches our starting and ending words. So, we loop with an index, *i* that goes from 0 to the length of the list, words, and for each element, it checks to see whether that node matches the starting or ending word. If it does, we save that index. Notice that we also keep track of the index, and if it turns out that one word or the other is not in the list, we will give the user a message that the word was not in the dictionary and exit. There are other ways we could deal with this if we wanted to, like as throwing an exception, or looping until the user gives us a valid word. But, in this case, we're just exiting the program, and the user can run it again to try a new combination.

So, now we have the index of the starting word and the index of the ending word. We can call the breadth-first search. We just call the function BFS, passing in the list words, and the starting and ending indices. BFS will return the path, which is a series of indices showing the path from the start to the goal. We store this in the variable path. It might seem weird that this main part of the code comes down to a single function call, but this is an advantage that we get from using the data structures and algorithms that we know of—the actual tasks we want can sometimes be very simple.

There's one more thing to be done, and that's to output the path. Remember, we have a list of node indices, giving the path from the starting word to the ending word. How would we print out this list to the screen? Well, this is pretty straightforward. We will loop through the elements of the path. Then, we just use that index to get the node from the words list and call getWord on that node to get the word. This is printed out to the screen and voila. we have printed out a chain of words from the start to the goal.

Let's run this one time to see how it works. Let's go back to our example, and enter car and let as our start and goal words. When we do this, we do indeed find a chain from car to bar to bat to bet to let. That's slightly different from the one I came up with in my head, where instead of bar I

had the word cat. Now, realize that if you try this yourself, you might get a different list—it'll depend on the dictionary you use, and on the order that the words appear in that dictionary.

So, we can say we've achieved success. But, there's one big problem with our program. If you've got this whole thing typed in and working, try testing it out for a while, and I bet you'll pretty quickly come across the error. Try testing it now.

Well, if you tested it out, you probably found out that the program will sometimes crash. And, if you did enough investigating, you'd figure out that the problem is occurring when there's not actually a chain between the two words. What's happened is that we've run BFS, and instead of finding a chain, it runs out of edges, and there's no path.

To fix this, we're going to need to modify the BFS routine. First, remember the BFS routine? We have a loop that keeps pulling off nodes from the queue until we've either found the goal or we run out of nodes. We then return the results of retrievePath. If you ran the code in the debugger or looked closely at the error messages when your program crashed, you'd find that BFS was crashing in retrievePath. See if you can figure out a simple fix to the BFS routine, so that it won't crash if there's not a path. Here's one simple fix. Instead of just returning retrievePath, we first check to see if a path was found. If so, we call retrievePath, just like before, and return it. If not, we return an empty list. That's a really simple fix.

Now we probably want to put a check into our output, also. We can check to see if we have an empty path and if so, we print that no chain was found. Otherwise, we print the chain like before. There are a lot more variations that you could add on to this routine, and if you find this program fun, I'd encourage you to try. For example, try using a much bigger dictionary, or try turning the program around—where the computer finds a chain of at least length 3 and then challenges the user to come up with a chain.

This word game has helped to demonstrate the power of data structures and algorithms. We took what was a challenging problem, and because we could represent it as a graph, and use a basic graph algorithm to solve it, a relatively complicated program became easier to write. Whenever we can find data structures and algorithms to fit our problem, the whole task of programming becomes much simpler.

The program we developed in this lecture would perform poorly if there's a really large word list provided at the beginning, due to the order n-squared number of checks between pairs of words. And that means that it could benefit from the techniques that we'll explore in our next, and final, lecture, where we will take up the subject of parallel computing.

I'll see you then.

24

# Parallel Computing Is Here

Parallel computing is both the future and present of computer programming. One of the biggest challenges that programmers will face in the long term is how to make effective use of the increasing parallelism that is being provided. To the extent that this can be solved, we will be able to see actual performance benefits that keep pace with processor improvements. In this lecture, you will learn about parallel computing.

## [ PARALLEL COMPUTING ]

- In the 1960s, Gordon Moore, one of the founders of Intel, made a famous prediction: that the number of transistors on an integrated circuit would follow an exponential growth rate, doubling every so many years. This idea came to be known as "Moore's law," and it has driven the computer chip design industry for many years. And though the rate of growth may have slowed, we're still seeing big improvements as our computers become smaller and faster.

- But there's a big difference between simply having more transistors and being able to use them effectively. As chip designers have had more and more to work with, it's been increasingly difficult to figure out how to use those additional transistors to get bigger and more powerful processors.

- Instead, designers have increasingly turned to parallelism to make use of the resources available. Instead of one bigger processor, they've used the transistors to make two processors—or four processors—on the same chip. This has led to dual-core, quad-core, and multicore home computers.

› Parallel computing is not a new idea. IBM researchers began exploring it in detail in the 1950s, and the first supercomputers in the 1970s were parallel machines. All of the supercomputers you've heard about are massively parallel machines. But parallelism is becoming increasingly widespread, as individual processors become multicore processors, a feature that has even migrated to smartphones.

› When a single processor becomes faster, we could expect everything running on it to run faster. But putting in a dual-core processor—in other words, using parallelism—doesn't necessarily cause our programs to run faster. The particular computation we're doing will determine whether we can actually make use of the parallelism provided.

## [ PARALLELISM IN PRACTICE ]

› In the following code, there are four computations getting assigned to variables *a* through *d*. The order we execute these statements doesn't really matter. In any order, we end up with the exact same variables having the exact same values. We would say that this code is easily parallelizable. We could do all four of these statements at the exact same time, and we would come out with the exact same answer.

```
a = 3*8
b = 7/12
c = 3.14159*4.0*4.0
d = (12+3)*(5-8)
```

› On the other hand, in the following lines of code, we have to compute *a* before we can compute *b*, because *b* needs to know the value of *a* to do its computation. Likewise, we need to compute *b* before *c* and *c* before *d*. There's no way we could compute these statements in a different order; they have to go in a particular sequence. This code could not be parallelized—there's no way to execute two or more of these statements at the same time.

```
a = 3.3+8.5
b = a*4.0*4.0
c = 16 - b
d = c/4.0
```

- Different applications will have different levels of parallelism, and a typical computation will have some pieces that can be parallelized and some that can't.

- Parallel computers are arranged in many different ways, and the ways you can use parallelism can change depending on how the processors are set up and how they can communicate with each other. Some parallel processing is done automatically and is hidden from you. The graphics processor, for example, automatically processes all the graphical elements that need to be drawn in parallel.

- But to make full use of parallel processing, we need to do it explicitly. We'll focus just on Python on a standard home computer. And, for this case, the main way we take advantage of parallelism is through **threading**, or **multiprocessing**.

- With threading, we'll be creating functions that can run in a different **process**, or "thread," than the main program. The main program will **spawn** these other processes, each of which will be running their respective functions. Those spawned processes will execute separately from the main program, and if there are multiple processors available (such as on a multicore computer), they'll run on these different cores at the same time—that is, in parallel. If there's just a single core central processing unit, these processes will still run just fine; there just won't be any overall improvement in the performance.

- The following is a very simple example of a multiprocess "Hello, World" program.

```
from multiprocessing import Process
def print_function():
    print("Hello, World!")
```

```
if __name__ == '__main__':
    p = Process(target=print_function)
    p.start()
```

- First, we'll be using the multiprocessing module. It's part of the Python standard library, so we just import it to get it. The main thing we're going to want from this module is a class definition called "Process." We will be creating instances of Processes.

- Next, we'll define a function that is the thing we're going to run in parallel. This function is going to be the thing spawned by the main program. In our case, our function is just called "print_function," and all it does is print "Hello, World!"

- In the main part of the program, there's a particular line of code we need to include that checks the name of the process: It's an if statement, and your code should all be indented from there. This line is where the Python multiprocessing module separates which code is part of the main, "primary" program as opposed to all the other spawned processes. Remember to include this line so that your code can work correctly.

- Next, you'll actually create an instance of the Process object. When we initialize the Process object, we have to pass in the function that we want the process to run, as the "target" parameter. In this case, our function is print_function, so we pass in "target=print_function" as our parameter.

- Finally, we spawn the process by calling the "start" method on the process.

- If we run this code, the interpreter comes along, creates the Process object, and then spawns it. That's all that happens in the main program, in this case. In the separate process that was spawned off, we have "Hello, World!" printed out, and that's what we see as the output.

```
from multiprocessing import Process
def print_function():
    print("Hello, World!")
```

```
if __name__ == '__main__':
    p = Process(target=print_function)
    p.start()

OUTPUT:
Hello, World!
```

› What if the function that will form our process takes in some parameters? In the following code, we've modified the print function to take in a name so that we can print "Hello" to that name. We can set up the arguments to be passed into the function through the use of an "args" (short for "arguments") parameter when we create the process.

```
from multiprocessing import Process
def print_function(name):
    print("Hello,", name)
if __name__ == '__main__':
    p = Process(target=print_function, args=("John",))
    p.start()
```

› The term "arguments" is another way of referring to the parameters being passed in to a function. In this case, we set the args to be "John," followed by a comma. The comma is used because the arguments list is expecting at least two args—two arguments—and that's because the arguments list gets turned into a tuple, which makes it non-mutable. A tuple of one is not possible, but simply adding a comma gives the tuple the appearance of two arguments to work with. If you don't put in the comma, you can get assignment errors. But set up with two arguments, this will print out "Hello, John."

› Now let's look at how we could spawn multiple processes in practice. The following is a variation on our earlier program. Notice that our function just takes in a number and prints a message "Printing from process" and then the number that was passed in.

```
from multiprocessing import Process
def print_process(number):
    print("Printing from process", number)
if __name__ == '__main__':
    process_list = []
    for i in range(20):
        p = Process(target=print_process, args=(i,))
        process_list.append(p)
    for p in process_list:
        p.start()
```

› In our main routine, we'll create a list of 20 processes. We'll have a loop, with *i* ranging up to 20, and for each one, we'll create a process in which "print_process" is the function and *i* is used as the parameter. After that, we'll go through and actually start each process, in a separate loop.

› When we run this, the following is the output. Notice that every process number, 0 through 19, gets printed once. But the order that these are printed seems pretty random; the earlier numbers seem to be getting printed before the later ones, but they're certainly not in the same order.

```
Printing from process 1
Printing from process 0
Printing from process 2
Printing from process 4
Printing from process 3
Printing from process 12
Printing from process 7
Printing from process 13
Printing from process 5
Printing from process 11
Printing from process 9
Printing from process 8
Printing from process 17
Printing from process 14
Printing from process 6
```

```
Printing from process 15
Printing from process 19
Printing from process 10
Printing from process 16
Printing from process 18
```

› Remember that each of those print statements was getting printed from a totally separate process. You can think of it as though it's a totally separate program that's running completely independently of the others, possibly on a different processor. And those different processors might have other things running on them—for example, some operating system commands or other applications running in the background. If you run this code a few times, you should see a different result every time—there are 20 factorial possible orderings.

## [ PARALLEL PROCESSORS ]

› The effectiveness of parallelism is measured by how effectively parallel processors can be applied to a particular problem. If you had two processors, the best situation would be if you could use both of them all the time with no time wasted. In this case, your overall running time would be cut in half. Four processors could cut running time in a quarter.

› However, in reality, we can't fully utilize the processors we have. Most problems are only partly parallelizable. In fact, there's a law called **Amdahl's law** that helps us calculate a limit for how much any given level of parallelism could speed up a computation.

› According to this law, a problem that's only 50% parallelizable cannot attain better than double the speedup time, no matter how many processors we throw at it. For a problem where 75% could be parallel, the maximum speedup is 4 times. For 90%, it's as large as 10 times, and if 95% can be parallel, speedup can be at most 20 times. However, these are theoretical upper limits; in practice, there can be other constraints, too.

- A follow-up to Amdahl's law, called **Gustafson's law**, helps us determine how much larger of a problem we can handle given more processors. Both these laws relate how much of the program needs to be done sequentially versus how much could be done in parallel.

- For example, graphics applications tend to be highly parallelizable. In a three-dimensional game, there are often hundreds of thousands of triangles being drawn, but the order they're drawn is not so important. Many scientific computations are also very parallelizable, with calculations taking place over a large grid, each piece of which can be handled separately from the others.

- Addressing these types of problems has contributed to the rise of another form of parallelism called grid computing—or, more generally, **distributed computing**—in which physically distinct, often dispersed, computers are loosely coupled with one another to handle distinct pieces of a single problem. Distributed computing can be thought of as a type of parallelism, because computation is being done on various computers at remote locations at the same time.

## [ PITFALLS AND ALTERNATIVE WAYS TO USE PARALLELISM ]

- There are many pitfalls along the way to becoming a good parallel programmer. For example, having to pass information only via things like queues can take some getting used to. Also, you basically can't use the "input" command as part of a parallel program—all the processes will stop while waiting for input. But you now know what's needed to write some simple parallel programs and how to parallelize existing slow code to get a speedup.

- In addition to creating parallel applications directly, there are some less direct ways that we can make use of parallelism in our code.

- If you've run multiple programs on your computer at one time, you're probably taking advantage of parallelism. Each program is running as a separate process, so if there are multiple programs running, they can potentially be running on separate processors. More commonly, when there are multiple processes running on one processor, the operating system takes care of "time sharing" the processor—basically, making sure that each process gets some fraction of time so that they all make progress together.

- In any case, we can initiate parallel computation by simply spawning new applications on our computer. And, fortunately, it's really easy to do this. If we use the "subprocess" module, we can spawn a new process in the operating system. Unlike the processes that we were using earlier, these don't remain tied to the Python program, so once they're spawned, they will continue to run on the computer, even if the Python program ends.

## Reading

Lubanovic, *Introducing Python*, chaps. 10–11.

## Exercise

What code would you write to spawn a new process, running a program named "myProgram.exe"?

24 TRANSCRIPT

# Parallel Computing Is Here

Back in the 1960s, Gordon Moore, one of the founders of Intel, made a famous prediction. He said that the number of transistors on an integrated circuit would follow an exponential growth rate, doubling every so many years. This idea came to be known as Moore's Law, and it's driven the computer chip design industry for many years. And although the rate of growth may have slowed, we're still seeing big improvements as our computers become smaller and faster.

But, there's a big difference between simply having more transistors, and being able to use them effectively. As chip designers have had more and more to work with, it's been increasingly tough to figure out how to use those additional transistors to get bigger and more powerful processors. Instead, designers have increasingly turned to parallelism to make use of the resources available. Instead of just one bigger processor, they've used the transistors to make two processors on the same chip. Or four processors. This has led to home computers that are dual-core, quad-core, and multi-core.

Parallel computing is not a new idea. IBM researchers began exploring it in detail in the 1950s, and the first Cray supercomputers in the 1970s were parallel machines. All of the supercomputers you've heard about are massively parallel machines. But, parallelism is becoming increasingly widespread, as individual processors become multicore processors, a feature that has even migrated to smartphones.

Now, when a single processor gets faster, we could expect everything running on it to run faster. But, putting in a dual-core processor, in other words, using parallelism, doesn't necessarily cause our programs to run faster. The particular computation that we're doing will determine whether we can actually make use of the parallelism that's provided.

Let's get look at some code just to illustrate. In this code, we have four computations getting assigned to variables *a*–*d*. Notice that the order we execute these statements doesn't really matter. We could compute *c* first, then *b*, then *d*, then *a*. In any order, we end up with the exact same variables having the exact same values. We would say that this code is easily parallelizable. We could actually do all four of these statements at the exact same time, and we come out with the exact same answer.

On the other hand, look at these lines of code. Notice that we have to compute *a* before we can compute *b* since *b* needs to know the value of a to do its computation. Likewise, we need to compute *b* before *c*, and *c* before *d*. There's no way we could compute these statements in a different order—they have to go in a particular sequence. This code could not be parallelized—there's no way to execute two or more of those statements at the same time.

Now, different applications will have different levels of parallelism. And, a typical computation will have some pieces that can be parallelized, and some that can't. You can think of this same thing in non-computing terms. If you're making a meal, you can imagine that it's easy to parallelize the making of two different dishes. But, it's not possible to parallelize everything. For example, you can't put the frosting on a cake until you've first baked the cake and mixed the frosting. You can also see why you might get a benefit from having 2 or 4 people to cook dinner, but that doesn't improve indefinitely.

So, how can we actually make use of parallel computing? Parallel computers are arranged in lots of different ways, and the ways you can use parallelism can change depending on how the processors are set up and how they can communicate with each other. Some parallel processing is done automatically and is hidden from you. The graphics processor does this, for instance—it automatically processes all the graphical elements that need to be drawn in parallel. But, to make full use of parallel processing, we need to do it explicitly. We'll focus just on Python on a standard home computer. And, for this case, the main way we take advantage of parallelism is through threading or multiprocessing.

I need to first explain the idea of threading. We're going to be creating functions that can run in a different process or thread than the main program. The main program will spawn these other processes, each of which will be running their respective functions. Those spawned processes will execute separately from the main program, and if there are multiple processors available like on a multicore computer, they'll run on these different cores at the same time—that is, in parallel. Now, if there's just a single core CPU, these processes will still run just fine. There just won't be any overall improvement in the performance.

Let's see what some multiprocessor code will look like. This a super-simple example, of a multi-process Hello, World style program. Let's look at the different parts of this. First, we'll be using the multiprocessing module. It's part of the Python standard library, so we just import it to get it. The main thing that we're going to want from this module is a class definition called Process. We will be creating instances of Process. Next, we'll define a function that is the thing we're going to run in parallel. This function is going to be the thing spawned by the main program. In our case here, our function is just called print_function, and all it does is print Hello World.

In the main part of the program, there's a particular line of code we need to include that checks that the name of the process. It's an if statement, and your code should go indented in from there. This line is where the Python multiprocessing module separates which code is part of the main or primary program as opposed to all the other spawned processes. Just remember to include this line so your code can work correctly.

Next, you'll actually create an instance of the Process object. When we initialize the Process object, we have to pass in the function that we want the process to run, as the target parameter. In this case, our function is print_function, so we pass in target = print_function as our parameter. And finally, we spawn the process by calling the start method on the process.

Now, if we run this code, what happens is that the interpreter comes along, creates the Process object, and then spawns it. That's all that happens in the main program. Now, in the separate process that was

spawned off, we have Hello, World printed out, and that's what we see as the output.

Let's look at a variation. What if the function that will form our process takes in some parameters? Here, we've modified the print function to take in a name, so that we can print hello to that name. We can set the arguments to be passed into the function through the use of an args parameter when we create the process. args is short for arguments, which is another way of referring to the parameters being passed in to a function. In this case, we set the args to be John comma. Why the comma? Well, the arguments list is expecting at least two args, two arguments, and that's because the arguments list gets turned into a tuple, which makes it non-mutable. A tuple of one is not possible, but simply adding a comma gives the tuple the appearance of 2 arguments to work with. If you don't put in the comma, you can get amusing assignment errors—for instance, the string John will get turned into a tuple of four individual letters. And then, the print statement will give an output error for getting called with four arguments. But set up with two arguments, this will print out Hello, John.

OK, now let's look at how we could spawn multiple processes in practice. Here's a variation on our earlier program. Notice that our function just takes in a number, and prints a message Printing from process and then the number that was passed in.

Now, in our main routine, we'll create a list of 20 processes. We'll have a loop, with *i* ranging up to 20, and for each one, we'll create a process in which print_process is the function, and *i* is used as the parameter. After that, we'll go through and actually start each process, in a separate loop. Now, I wonder if you can guess what will happen here?

Well, here's the output I got when running it. Notice that every process number, 0 through 19, gets printed once. But, the order that these are printed seems pretty random. Not totally random—the earlier numbers come before the later ones, but they're certainly not in the same order. What's going on?

Well, remember that each of those print statements was getting printed from a totally separate process. You can think of it as though it's a totally separate program that's running completely independently of all the others, possibly on a different processor. And, those different processors might have other things running on them, like the operating system commands or applications just running in the background. Try running this code yourself a few times and you should see a different result every time—there are 20 factorial possible orderings. Now, there are some other details to parallel programming, but that's enough to get us started.

The effectiveness of parallelism is measured by how effectively parallel processors can be applied to a particular problem. If I gave you two processors, the absolute best possible situation would be that if you could use both of them all the time with no time wasted. In this case, your overall running time would be cut in half. Four processors could cut running time in a quarter. But, in reality, we can't fully utilize the processors we have. Most problems are only partly parallelizable. In fact, there's a law called Amdahl's law that helps us calculate a limit for how much any given level of parallelism could speed up a computation.

According to this law, a problem that's only 50% parallelizable cannot attain better than a double speed up time, no matter how many processors we throw at it. For a problem where 75% could be parallel, the maximum speed up is four times. For 90%, it's as large as 10 times, and if 95% can be parallel, speed up can be at most 20 times. However, these are still theoretical upper limits—in practice, there can be other constraints, also.

A follow-up to Amdahl called Gustafson's law helps us determine how much larger of a problem we can handle given more processors. Both of these laws relate how much of the program needs to be done sequentially versus how much could be done in parallel. For example, graphics applications tend to be highly parallelizable. In a 3D game, there are often hundreds of thousands of triangles being drawn, but the order they're drawn isn't so important. A lot of scientific computations

are also very parallelizable, with calculations taking place over a large grid, each piece of which can be handled separately from the others.

Addressing these types of problems has contributed to the rise of another form of parallelism called grid computing, or more generally, distributed computing, in which physically distinct, often dispersed, computers are loosely coupled with one another to tackle distinct pieces of a single problem. Distributed computing can be thought of as a type of parallelism, since computation is being done on various computers at remote locations, all at the same time. Let me discuss it for just a minute.

The most basic type of distributed application is the remote procedure call or RPC. A procedure is just a function, and the basic idea is that you're making a typical function call in your program. But, instead of the function executing on your computer, it executes on a remote computer. Often, that remote computer is a server set up to respond to the RPCs. The remote computer might just have more computing power available, or it might have access to data that you don't on your own machine. The Pyro module—that's P-Y-R-O—is one way to write programs that communicate with each other remotely, through these remote function calls.

Also, since people don't often have the ability to set up their own servers at remote locations, several companies will host servers for you. Amazon Web Services is one of the most well-known, and there's a module for that called PiCloud—spelled P-I-Cloud.

Let's reconsider a few of the programs we've developed in this course. See which of these you think could be made parallel: Remember the program where we looked at weather data to predict temperature and rain for a specific date? We went through a large list of data to find days that matched the date we cared about. Could that be parallel, or not? Yes. That could've been parallelized—although dates do come one after one another in the calendar, here, it was ok to check the dates in any order.

How about our financial simulation? There were actually two parts to this simulation. First, how about where we were computing account

balances over many years? Parallel or not? Well, no. Here, each year is dependent on the previous year, so this can not be parallelized. The years are so chained to each other that we're forced to process earlier years before we can go on to later years.

In our financial simulation, we also ran multiple scenarios to get a Monte Carlo simulation of that financial program. So how about multiple scenarios? Parallel or not? Yes. The multiple scenarios part of the code could be parallelized—each run has its own set of random data, and it's independent of every other run. Simulations where we run the same thing many times, just with different random data each time, are some of the best cases for massively parallel processing.

How about our word game, where we were doing paired comparisons between words to form a graph in the word game? Parallel or not? Yes. Each word pair can be compared apart from all the others. This has a big potential of being parallelized. So, let's try this, let's modify our game. After all, the code was rather slow at the beginning, as we tried to set up the graph. We were comparing every word to every other word. Wouldn't it be nice if we could speed that up?

Now, to do this, we need to understand how information can be passed back and forth in a multiprocessor program. When we're running multiple processes, these processes are effectively operating as separate programs. Any changes they make to data within their own process won't be shared with other processes running in parallel or with the main program. Information from one process to another needs to be passed back and forth through special data structures that are designed to work in a multiprocessor environment.

The main tool for passing information like this is the multiprocessor queue. It's part of the multiprocessor module and it's not the same as a standard queue that can be found in other modules or that we would write ourselves. This queue can be passed as a parameter, and any process can add on to it safely, with the information remaining for all of the processes to use.

Let me remind you of the previous version of our code for building up the graph structure words. We have two nested loops to compare all the words in the list of words with all the other words in the list. For each pair, we would call the compareWords function, and if they differ by exactly one letter, we call addLink to connect them.

To parallelize this, we'll break the comparisons into smaller groups. Basically, we can divide up that outer loop, where *i* goes over the whole range, into smaller chunks, where, say, one process will handle about 100 of those *i* values. That's kind of an arbitrary choice, but we want to balance the number of processes with the amount of work each one does. We don't want too many processes or the overhead of starting each of them will take away any gains. But, if we have too few, we won't actually be able to get the benefits of parallelism. We want at least as many processes as we have cores to process them.

Now, I mentioned that separate processes can't actually affect the common data, but they have to pass information back and forth. So, it's not feasible to have each process actually modify our graph and call addLink. Instead, we'll have each process return a list of pairs of words that it finds that should be linked, and then we'll actually do the linking back in the main program. We can get this list of words via that multiprocessor Queue that we just discussed. Each process will add word pairs into the queue, and then the main program will take them out of the queue and form links.

So, first, let's look at the function that we'll want to use in our parallel processes. We'll call this function findlinks. As input, it will take the wordlist, the Queue where it should store any pairs that it finds, and the starting and ending values of i to use in the loop.

The body of the function is very similar to the code we had previously for finding and forming links. We have an outer loop indexed by *i*. We loop here only from the starting *i* value to the ending *i* value that were passed in as parameters. The inner loop will be indexed by *j*, and is identical to the one before, as is the if statement comparing two words by calling compareWords. Finally, where we used to have a command to

add a link, we instead will add an *i, j* tuple to the queue that was passed in as a parameter.

And, that's it for the function that each process will run: each process will perform these checks between pairs of words and stick any that need to be linked into the queue.

Now, let's look at the code to actually set up the processes, which is a little more complicated. Let's take a look. We begin with the special if statement that's needed for multiprocessing. All of our main program will be indented under this. After reading in our word list, we set up a queue, and we create an empty list of processes. We then create a for loop. The For loop will take groups of 100 words, which is why the index ranges from 0 to the length of the wordlist minus 100, in steps of 100. We'll create a process object, using the findlinks function we defined earlier as our target. We then pass in arguments of the word list, the queue, and then the starting and ending values for the outer loop. These are given by our current loop value, *i*, as well as *i* plus 100 for the ending. After that loop, we create one final process to handle any words remaining that didn't fill up a full 100 words from the list. There will be a small number of words, under 100, left over that weren't caught in the for loop. With these processes created, we then start all of them. This spawns completely separate processes that will run on their own on any processors available. Remember that these processes will be putting pairs of indices into the multiprocessor queue.

Now, while this is going on, the main program can start processing the pairs on the queue. As long as the queue is not empty, it can pull off an element using the get command, and assign the tuple to *i* and *j*. Then, an add link between nodes i and j. Remember, we couldn't add the links to the nodes in the process itself since the changes would not have affected the main program. Finally, we take all the processes and join them. The join command is used to make sure that other processes have finished before we move forward. By calling join, we make sure that we don't go too far before the existing processes are finished. Finally, we go through the queue one last time in case any straggler process put words on the queue before we're finished.

So, that's the set of changes needed to turn this code into a parallel version. What do you think will happen if we run the code? To give you an idea, the original code took my laptop about 35 seconds to set up a graph for a set of about 5500 words. I have a dual-core laptop. So, how fast do you think it will be able to set up the graph with the parallel version?

In my case, it took about 25 seconds. Notice that in an ideal world, the two cores would have halved my running time, to about 18 seconds. It didn't do quite that well, since there was some extra time needed to start, and join the processes, and to do the extra work of keeping things in a queue. But, a drop from 35 seconds to 25 seconds is still more than a 25% reduction. So, when we care about time, it's clearly going to be a benefit to use parallelism.

There are more commands available in the multiprocessing module that you might find interesting. For instance, Managers will let you share data between processes, and process Pools will let you set up sets of processes that can be used as needed later on. I'll mention that there's also a Threading module available. Threading works a whole lot like multiprocessing, but it has the additional benefit that memory is shared among all the processes, called threads, so they can affect each other. However, Python does not really let code run in multiple CPU threads, so if we have a process that's CPU-intensive, like our word matching was, threading doesn't actually give a performance increase for many classes of problems. So, if you're really interested in increasing speed, it's better to stick to the multiprocessing module.

Now, there are a lot of pitfalls along the way to becoming a good parallel programmer. For instance, having to pass information only via things like queues can really take some getting used to. Also, you basically can't use the input command as part of a parallel program—all the processes will stop while waiting for input.

But, you now know what's needed to write some simple parallel programs on your own and how we can parallelize existing slow code to get a speedup. You should also have a much better idea of how the

whole threading process is working when you encounter threaded programs, and you can form judgments for yourself about how your parallel computer will or won't perform better on some application.

In addition to creating parallel applications directly, there are some less direct ways that we can make use of parallelism in our code. I'll show you one example. It'll be our final program in this course, and it's one that you might find useful for optimizing the way you open applications on your desktop.

If you've run multiple programs on your computer at one time, you're probably taking advantage of parallelism right there. Each program is running as a separate process, and so if there are multiple programs running, they can potentially be running on separate processors. More commonly, when there are multiple processes running on one processor, the operating system takes care of time sharing the processor—basically making sure that each process gets some fraction of time, so they all make progress together.

In any case, we can initiate parallel computation by simply spawning new applications on our computer. And, fortunately, it's really easy to do this. We saw an example midway through the course when we demonstrated code to open a browser window. With just a few lines of code, you could pop open a browser window to some website. Starting a different application isn't that much tougher.

If we use the subprocess module, we can spawn a new process in the operating system. Unlike the processes that we were using earlier, these don't remain tied to the Python program, so once they're spawned, they will continue to run on the computer, even if the Python program ends.

Let's look at a quick example. Say we wanted to open up the notepad program on a Windows machine. The program is located in the Windows directory on the C drive. We could import the subprocess module. Then, we call subprocess.run and give a string with the application to run. In this case, that would be C:\Windows\notepad.exe. Remember that for a backslash in a string, we need to use a double backslash. If you have an

older installation of Python, you might need to use the Popen command instead of run, but the effect is the same. Running this will launch the notepad program.

You can do the same thing for any executable file on your computer. Just spawn a process for that file by using the subprocess.run command, and put in the exact location of the file on your computer. Even if you leave the Python program, those processes will stick around. You can specify command-line arguments to files, by including that within the string. A command-line argument is when you pass a parameter to a routine by specifying it on the command line when you start the function. For example, notepad will take a command line argument of the name of the file to open.

So, the code you see here will spawn two processes. The first will open notepad with a file dictionary.txt located one directory up from where the Python file that we're running is. The second will open Microsoft Excel, using a spreadsheet in my Documents folder. Obviously, you would need to modify the file names and locations to match your own machine, but the basic format is the same.

So, try implementing a file like this yourself. Do you ever find yourself opening more than one application at the same time? Maybe you open a file explorer and a photo editor together, so you can drag and drop files from the explorer into the window? Or, maybe you open a spreadsheet for trip information at the same time you open up a web browser to a travel site? Whatever the case, you can easily create a small Python script—in other words, a Python program, that'll do this. Then, save it to your desktop or some other location where it'll be easy to find, and you can spawn those programs by running that single file.

While these might seem like really simple operations and programs, they are spawning additional processes, the same way that the Process spawning that we saw earlier was working. And we're potentially taking advantage of parallel computing power on the processor itself.

Parallel computing is the both the future and the present of computer programming. One of the biggest challenges that we'll face as programmers in the long term is how to make effective use of the increasing parallelism that's being provided. To the extent we can solve this, we'll be able to see actual performance benefits that keep pace with processor improvements.

Well, we've reached the end of our course. And, there was much rejoicing. Congratulations.

Looking back over the course, we've really come quite a ways. If you've made it this far, you've covered not only the topics commonly covered in a first programming course, but with the data structures and algorithms topics, you've seen material often found in a second course. And, our discussion here about parallelism brings you to the frontiers of current-day practice.

Following this course actively, means that you now have the skills to do a good amount of programming on your own. Just remember that programming is a skill like any other, and it can be developed through practice.

One great way to develop your Python skills further would be to take any of the programs from the course and try to add your own improvements. Another would be to pick a topic that's interesting to you, and select a module that deals with that topic. For graphics, you might start with the turtle and pyglet modules. For simulations and data visualization, there's random and matplotlib. For working with files and applications on your own computer, you might start with os and shutil. There are thousands of modules out there, so find one that does something you find interesting. Then, using that module just try writing some programs for fun.

You never need to understand everything about a module to start trying to work with it—that 's the beauty of abstraction. And, everyone needs to look up syntax sometimes, there's nothing wrong with getting help from a book, an online source, or another person.

Programming can be an almost uniquely fun and satisfying endeavor—there's really nothing quite like it. Now that you've seen programming, from lots of different angles, let me tell you why programming has held my interest for decades now. I think that now you can appreciate each of these points for yourself.

First, programming is a problem-solving exercise. When we have a larger goal in mind, we have to figure out how to turn that idea into code. Figuring out just how to apply all the different tools available to solve a problem can be both challenging and fun. Even debugging, as painful as it can be at the time, repeatedly offers a mystery to solve. And, when everything finally comes together and works, it's an incredible feeling that you overcame all the obstacles and turned some mental model into real working software.

Second, programming lets you express your creativity. You can dream up all sorts of ideas and use programming to actually see them come to life. If you noticed, most of our methods for software development were more about design than they were about actual specific code. Programming gives you a different way to exercise your design skills.

And third, programming affects the way you think. Programming teaches you to approach problems in a logical, orderly fashion, to develop plans that can be followed sequentially, and to recognize how to break a complex problem into more manageable pieces. Abstraction can be a useful approach in the real world, just like it is in the computer.

So, as you go out from here, I'll be happy if you've been able to experience and take with you the spark of enjoyment I get from programming. Even if programming is just a side hobby in your life, it can be a source of satisfying challenges and fun activity. It can become a life-long passion.

Thanks for spending this course with me. Always look on the bright side of life, and Happy Programming.

# Answers

## LECTURE 1

1 `15`

2 `25` (** raises to a power, so 3**2 is 9 and 4**2 is 16.)

3 `21`

4 Nothing (Notice that this is a comment.)

5 `1.0`

6 `0` (// is integer division, so 1//2 is 0.)

7 `1` (% is modulus, or the remainder when divided by the number. So, odd numbers %2 will be 1, while even numbers %2 will be 0.)

8
```
print (12*8)
```

9
```
print(180/7)
```

10
```
print("I love Python")
```

11
```
#This is my first program
```

**REMEMBER**

There is more than one way to write code correctly.

## LECTURE 2

1. `25`
2. `15`
3. `115`
4. `250`
5. `10.0`
6. `10`
7. `Welcome Home` (The comma causes a space to be printed.)
8. `WelcomeHome` (Concatenation eliminates the space.)
9. `1015` (The + indicates concatenation. The conversion to an integer comes after the concatenation.)
10. `bread_price = 2.0`
11. `total_price = 2*bread_price + 3*cheese_price`
12. `age = int(input("What is your age?"))` (Remember: Convert a number that's input to an integer.)
13. 
```
knight_name = input("What is the knight's name?")
knight_trait = input("What is a characteristic of the knight?")
print("Sir "+knight_name+" the "+knight_trait)
```

## LECTURE 3

```
1  False

2  False                                    (== tests for equality.)

3  True                                     (!= tests for inequality.)

4  True

5  True                                     (Strings are compared letter by letter.)

6  False

7  True                                     (Capital letters always come before
                                             lowercase letters, so "O" < "o.")
8  False                                    (All capital letters come before
                                             lowercase letters, so "o" > "T.")
9  True

10 False

11 True

12 True

13 Reasonable cost

14 Eligible for reduced benefits
```

15
```
age = 67
income = 10000
if (income < 15000) and (age >= 70):
    print("Eligible for benefits")
elif (income < 20000):
    print("Eligible for reduced benefits")
else:
    print("Not eligible for benefits")
```

16
```
if user_guess < hidden_answer:
    print("Too low")
elif user_guess > hidden_answer:
    print("Too high")
else:
    print("You guessed it!")
```

17 a)
```
if year % 4 == 0:
      if year % 100 == 0:
          if year % 400 == 0:
              print("Leap year")
          else:
              print("Not a leap year")
      else:
          print("Leap year")
  else:
      print("Not a leap year")
```

b)
```
if year % 400 == 0:
      print("Leap year")
  elif year % 100 == 0:
      print("Not a leap year")
  elif year % 4 == 0:
      print("Leap year")
  else:
      print("Not a leap year")
```

c)
```
if (year%4 != 0) or ((year%4 == 0) and (year%100 == 0)
  and (year%400 != 0)):
      print("Not a leap year")
  else:
      print("Leap year")
```

## LECTURE 4

The following is one example of the modified code.

For the first part, notice that we added a variable, "cost," and then created another variable, "saved," which we read in from the user. We then compute "balance" as the difference.

For the second part, notice that we have a new input line that gets the period being saved for and that this variable is used in the later input and output statements.

```
#Get information from user
print("I'll help you determine how long you will need to save.")
name = input("What's your name? ")
item = input("What is it you are saving up for? ")
cost = float(input("OK, "+name+".  Please enter the cost of the
  "+item+": "))
saved = float(input("How much have you already saved? "))
balance = cost-saved
period = input("How often will you save (day, week, month)? ")
if (balance<0):
    print("Looks like you already saved enough!")
    balance = 0
    payment = 1
else:
    payment = float(input("Enter how much you will save each
      "+period+": "))
```

```
    if (payment <= 0):
        payment = float(input("Savings must be positive.  Please
          enter a positive value:"))
        if (payment <=0):
            print(name+", you still didn't enter a positive
              number!  I am going to just assume you save 1 per
              "+period+".")
            payment = 1
#Calculate number of payments that will be needed
num_remaining_payments = int(balance/payment)
if (num_remaining_payments < balance/payment):
    num_remaining_payments = num_remaining_payments + 1
#Present information to user
print(name+", if you save", payment, "each "+period+", you must
  make", num_remaining_payments, "more payments, and then you'll
  have your "+item+"!")
```

## LECTURE 5

1
```
10
5.0
2.5
1.25
```

2
```
0
1
3
6
10
15
21
```

(Notice that the value can exceed 20 inside the loop, and the check to see if it is less than 20 is only after the loop body is completed.)

3
```
0
1
2
3
```
(Remember that the range starts at 0, and only continues while less than the number given in parentheses.)

4
```
3
4
```
(The range starts at 3 and continues while *i* is less than 5.)

5
```
1
4
7
```

6 Nothing printed

(Notice that the increment is negative, so we are counting down. The starting value, 1, is less than the minimum, 10.)

7
```
10
7
4
```

8 The following are two versions: one with a while loop and one with a for loop. Notice that you need to start at 1, not 0, and include the final number in the list, either by using "<=," as in the while loop, or "num+1," as in the for loop.

```
num = int(input("Enter a number to count to: "))
i = 1
while (i<=num):
    print(i)
    i += 1
```

---

```
num = int(input("Enter a number to count to: "))
for i in range(1,num+1):
    print(i)
```

9
```
for i in range(2,7,3):
    print(i)
```

**10** The following are two possible versions.

```
secret_number = 7
while True:
    guess = int(input("Enter your guess from 1 to 10: "))
    if guess == secret_number:
        break
    else:
        print("No!  Try again.")
print("You guessed it!")
```

---

```
secret_number = 7
guess = 0
while guess != secret_number:
    guess = int(input("Enter your guess from 1 to 10: "))
    if guess != secret_number:
        print("No!  Try again.")
print("You guessed it!")
```

## LECTURE 6

**1**
```
infile = open("data.txt", 'r')
```

**2**
```
outfile = open("../data.txt", 'w')
```

**3**
```
infile.close()
outfile.close()
```

**4**
```
for l in infile.readlines():
    print(l ,end='')
```

(Note: By adding the ",end=''" to the print statement, we eliminate the final newline that is printed at the end of each print statement. This was not asked for in the problem but can be a useful technique.)

5
```
filename = input("What file should we write to? ")
outfile = open(filename, 'w')
for i in range(1,11):
    outfile.write(str(i)+'\n')
outfile.close()
```

6
```
infile = open("data.txt", 'r')
i = 0
sum = 0
for l in infile.readlines():
    num = int(l)
    sum += num
    i+=1
average = (sum)/i
infile.close()
print(average)
```

## LECTURE 7

1
```
4
```

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

3
```
[2, 4, 6]
```

4
```
[18, 20]
```

5
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

6
```
20
```

7
```
[16, 18, 20]
```

8
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

9
```
2
4
6
8
10
12
14
16
18
20
```

10
```
[2, 4, 6, 100, 10, 12, 14, 16, 18, 20]
```

11
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

12
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 100]
```

13
```
[2, 12, 14, 16, 18, 20]
```

14
```
[2, 4, 100, 200, 18, 20]
```

15
```
[2, 4, 100, 6, 8, 10, 12, 14, 16, 18, 20]
```

16
```
minor_ages = []
for age in ages:
    if age < 18:
        minor_ages.append(age)
```

17
```
mylist = [["John", "James", "Joel"], [25, 28, 30]]
```

## LECTURE 8

The following are changes to the relevant parts of the program. The new lines are in bold.

```
# Perform analysis
minsofar = 120
maxsofar = -100
numgooddates = 0
sumofmin=0
sumofmax=0
raindays = 0
for singleday in gooddata:
    numgooddates += 1
    sumofmin += singleday[1]
    sumofmax += singleday[2]
    if singleday[1] < minsofar:
        minsofar = singleday[1]
    if singleday[2] > maxsofar:
        maxsofar = singleday[2]
    if singleday[3] > 0:
        raindays += 1
avglow = sumofmin / numgooddates
avghigh = sumofmax / numgooddates
rainpercent = raindays / numgooddates * 100
########## Present Results ##########
print("There were", numgooddates,"days")
print("The lowest temperature on record was", minsofar)
print("The highest temperature on record was", maxsofar)
print("The average low has been", avglow)
print("The average high has been", avghigh)
print("The chance of rain is", rainpercent, "%")
```

## LECTURE 9

1 XXXXX

2 256

3 4 6

4 21

5 9 12

6
```
def print_string(numtimes, str):
    for i in range(numtimes):
        print(str)
```

7
```
def small_list(listA, listB):
    newlist = []
    for i in range(len(listA)):
        if listA[i] < listB[i]:
            newlist.append(listA[i])
        else:
            newlist.append(listB[i])
    return newlist
```

8
```
def middle(a, b, c):
    if ((a >= b) and (b >= c)) or ((a <= b) and (b <= c)):
        return b
    elif ((a >= c) and (c >= b)) or ((a <= c) and (c <= b)):
        return c
    else:
        return a
```

```
9   def findincrease(val1, val2):
        if val2 > val1:
            return val2 - val1
        else:
            return 0
    salary1 = float(input("Enter previous salary"))
    benefits1 = float(input("Enter previous benefits"))
    bonus1 = float(input("Enter previous bonus"))
    salary2 = float(input("Enter new salary"))
    benefits2 = float(input("Enter new benefits"))
    bonus2 = float(input("Enter new bonus"))
    salaryincrease = findincrease(salary1, salary2)
    benefitsincrease = findincrease(benefits1, benefits2)
    bonusincrease = findincrease(bonus1, bonus2)
```

## LECTURE 10

```
1   3

2   [0, 2, 3]

3   21

4   2

5   3

6   [0, 2, 3]

7   2 1

8   def increment_list(a):
        for i in range(len(a)):
            a[i] += 1
```

9
```
def multiply4(a, b=1, c=1, d=1):
    return a*b*c*d
```

## LECTURE 11

1 The four cases on either side of a "boundary": 2, 3, 12, 13.

Some "middle" values, such as 1, 7, 25.

"Extreme" cases, such as 0 or 100.

2 a)
```
def findmiddle(a):
    if len(a) < 3:
        raise TypeError  # This could be a different
          exception
    if ((a[0] >= a[1]) and (a[1] >= a[2])) or ((a[0] <=
      a[1]) and (a[1] <= a[2])):
        return a[1]
    elif ((a[0] >= a[2]) and (a[2] >= a[1])) or ((a[0] <=
      a[2]) and (a[2] <= a[1])):
        return a[2]
    else:
        return a[0]
```

b)
```
try:
      middle = findmiddle(a)   #a is some
  except TypeError:
      print("Problem: You need a list of at least length
        3!")
```

## LECTURE 12

1 a) gzip (or zlib, zipfile)
b) fractions
c) smtplib
d) urllib

2
```
import os
os.mkdir("DataDir")
```

## LECTURE 13

Note: If you use a different board definition, your other functions would change, too.

1
```
board = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

2
```
def make_move (board, position, value):
    position -= 1           # change position to be 0-8
    pos1 = position // 3    # integer division gets the row
      number
    pos2 = position % 3     # modulus gives the column number
    board[pos1][pos2] = value
```

3
```
def first_row(board):
    if board[0][0] == 'X' and board[0][1] == 'X' and board[0]
      [2] == 'X':
        return 'X'
    elif board[0][0] == 'O' and board[0][1] == 'O' and
      board[0][2] == 'O':
        return 'O'
    else:
        return '.'
```

## LECTURE 14

```
import turtle
def drawA():
    # Draw the left side of the A
    turtle.left(60)
    turtle.forward(20)
    # Draw half the right side of the A
    turtle.right(120)
    turtle.forward(10)
    # Draw the cross-part of the A
    turtle.left(60)
    turtle.backward(10)
    turtle.forward(10)
    # Draw remainder of the right side
    turtle.right(60)
    turtle.forward(10)
    # Return the turtle to original orientation
    turtle.left(60)
    # Move turtle over a small amount
    turtle.up()
    turtle.forward(5)
    turtle.down()
```

## LECTURE 15

```
1   import pyglet
    window = pyglet.window.Window(width=400, height=300,
      caption="ExerciseWindow")
    Im1 = pyglet.image.load('BlueTri.jpg')
    @window.event
    def on_mouse_press(x, y, button, modifiers):
        window.clear()
        Im1.blit(x,y)
    pyglet.app.run()
```

```
2   import tkinter
    class Application(tkinter.Frame):
        def __init__(self, master=None):
            tkinter.Frame.__init__(self, master)
            self.pack()
            self.hello_button = tkinter.Button(self)
            self.hello_button["text"] = "Print repeatedly"
            self.hello_button["command"] = self.printtimes
            self.hello_button.pack(side="bottom")
        def printtimes(self):
            global times
            for i in range(times):
                print("Hello!")
            times += 1
    times = 1
    root = tkinter.Tk()
    app = Application(master=root)
    app.mainloop()
```

## LECTURE 16

```
import random
from matplotlib.pyplot import show, hist
rolls = []
for i in range(10000):
    roll = (random.randrange(6)+1) + (random.randrange(6)+1) +
      (random.randrange(6)+1)
    rolls.append(roll)
hist(rolls, bins=16)
show()
```

## LECTURE 17

1
```
20
100.0
10
300.0
```

2
```
40
1800.0
50
750.0
```

3
```
def print(self):
    print(self.item+"   barcode: "+str(self.barcode))
    print("Price:",self.price)
    print("Current Inventory:", self.quantity)
    print("Sold so far:", self.sales)
```

4
```
class Movie:
    title = ""
    genre = ""
    rating = 0.0
```

5
```
    def __init__(self, t, g, r):
        self._title = t
        self._genre = g
        self._rating = r
```

6
```
movielist = []
rating = 1.0
while rating >= 0.0:
    title = input("Enter the movie title:")
    genre = input("What is the genre of this movie?")
    rating = float(input("How do you rate the movie?"))
    if rating >= 0.0:
        movie = Movie(title, genre, rating)
        movielist.append(movie)
```

## LECTURE 18

**1** a)
```
class Videogame(Game):
    platform = ""
```

b)
```
class Boardgame(Game):
    numpieces = 0
    board = [0,0]
```

**2** In "Videogame" class:

```
def print(self):
    print(self.name)
    print("Up to ", self.numplayers, "players")
    print("Can be played on", self.platform)
```

In "Boardgame" class:

```
def print(self):
    print(self.name)
    print("Up to ", self.numplayers, "players")
    print("Has", self.numpieces, "pieces, and a board of
      size ", self.board[0], "by",self.board[1])
```

**3**
```
tetris = Videogame()
tetris.name = "Tetris"
tetris.numplayers = 1
tetris.platform = "Windows"
tetris.print()
```

**4**
```
import pickle
outfile = open("Game.dat", 'wb')
pickle.dump(tetris,outfile)
outfile.close()
```

**5**
```
import pickle
infile = open("Game.dat", 'rb')
savedgame = pickle.load(infile)
infile.close()
```

## LECTURE 19

1
```
Joseph
James
John
```

2
```
John
James
Joseph
```

3
```
Michael Palin
Michael Palin
Terry Gilliam
```

4
```
{2, 3, 37, 5, 7, 11, 23, 29, 31}
{17, 19, 13}
{2, 3, 37, 5, 7, 11, 13, 14, 15,
  16, 17, 18, 19, 23, 29, 31}
{2, 3, 5, 7, 11, 14, 15, 16, 18, 23, 29, 31, 37}
```

(Note: The order of elements in a set does not matter.)

## LECTURE 20

1
```
def swap(lst, i):
    temp = lst[i]
    lst[i] = lst[i+1]
    lst[i+1] = temp
```

2
```
def one_bubble_pass(lst):
    returnval = False
    for i in range(len(lst)-1):
        if lst[i] > lst[i+1]:
            swap(lst,i)
            returnval = True
    return returnval
```

3
```
def bubblesort(lst):
    keepgoing = True
    while keepgoing:
        keepgoing = one_bubble_pass(lst)
```

## LECTURE 21

1 It computes the product of all the elements in a list. Notice that the recursive call gives 1 for an empty list. For a larger list, it multiplies the first element by the result of the call on the rest of the list.

2 a) "swap" is a constant time function. The time taken to perform a single swap function is independent of the length of the list.

b) "one_bubble_pass" is a linear time function. Each element of the list is visited once, on a single pass through the list.

c) "bubblesort" is a quadratic function. In the worst case, there are a linear number of passes through the while loop, each of which calls "one_bubble_pass," which is again linear.

## LECTURE 22

1

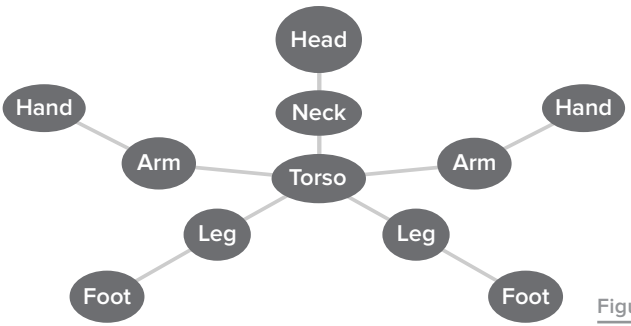



Figure A22.a

2 4. This happens when traveling from one foot, hand, or head to another.

3 Yes. It is connected and does not have any cycles.

4

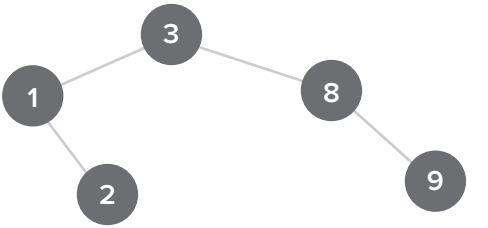



Figure A22.b

5
```
cities = {}
cities['Rivertown'] = 1000
cities['Brookside'] = 1500
cities['Hillsview'] = 500
cities['Forrest City'] = 800
cities['Lakeside'] = 1100
...
road = roads[0]
pop1 = 0
pop2 = 0
pop1 = cities[road.getName1()]
pop2 = cities[road.getName2()]
```

## LECTURE 23

E, H, K, M, L, J, G.

We start with node E as our first node.

If we assume that we put items on the stack in the order they appear in each node, then it will put nodes C, I, and H on the stack in that order.

We will next visit the top node on the stack, which is H. It will put K on the stack (I has already been seen).

We will next visit K. It will put M on the stack.

M will be visited next, and there we will put L on the stack.

When visiting L, we will put J on the stack.

When visiting J, we will find G, our goal.

Notice that nodes I and C are placed on the stack but never are popped off.

The route to G will be E–H–K–M–L–J–G.

Clearly, a stack does not produce the shortest path, unlike the queue.

Alternative answer: E, C, A, B, D, F, G.

If we instead assume that items are put on the stack in the reverse order they appear in each node, then at node E, we will first put H, then I, and then C on the stack.

We would then visit node C. It would put F, then D, and then A on the stack.

A would be visited next. There, we would put B on the stack.

We would next visit B. All of its neighbors would have been seen, so we will return.

Next, we would visit the next node on the stack, D. Again, its neighbors have already been seen.

The next node on the stack is F. When we visit F, we will find G as its neighbor and be done.

Notice that nodes I and H are placed on the stack but never are popped off.

The route to G will be E–C–F–G.

In this case, the stack managed to find the shortest route to G, but this is not guaranteed.

The depth-first search approach will still find some path to the goal, if one exists, but it might not be the shortest path. But a depth-first search is used as a substep in other graph algorithms.

## LECTURE 24

```
import subprocess
subprocess.Popen("myProgram.exe")
```

This mirrors the subprocess spawning as shown in the lecture. We will have the program “myProgram.exe” running in a separate process at the same time as our Python program is running.

# Glossary

**abstract interface**: When a class defines a function that cannot be called. The function must be defined by a child class that inherits from that class. [Lecture 18]

**abstraction**: To simplify the details of something complex and present a simpler view that provides the essence of the complex idea. Abstraction provides much of the power of computer science, where complex functionality is presented as something simpler. In programming, functions provide abstraction, allowing several operations to be described with one function call. [Lecture 9]

**adjacency list**: A list of nodes connected by an edge from a given node. [Lecture 22]

**adjacency matrix**: A matrix describing the edges connecting all pairs of nodes. [Lecture 22]

**algorithm**: An ordered sequence of steps to accomplish some task. [Lecture 20]

**Amdahl's law**: A rule that determines the theoretical limit for how much speedup can be obtained through parallelism. [Lecture 24]

**append**: To add on to the end of a list. [Lecture 7]

**assignment**: A statement that gives a value to a variable, metaphorically like putting a value into a box. In Python (and many other languages, including C, Java, and Fortran), assignment is indicated with an equal sign (=); this is different from a check for equality. In Python, assigning to a variable that has not previously been used will create a new variable with that name. [Lecture 2]

**asymptotic analysis**: Technique for assessing the efficiency of an algorithm. Asymptotic analysis describes the growth in running time as a function of growth of input. [Lecture 21]

**attribute** (also called **field** (Java) and **member variable** (C++)): A variable defined for a particular object. This is usually defined in a class, so that all objects in that class have that attribute. [Lecture 17]

**base case**: A special case in a recursive function that returns without making a recursive call. The base case is what causes a recursive routine to eventually stop. [Lecture 21]

**big-O notation**: A way of describing asymptotic running times. Written as $O(x)$, where $x$ gives the "order" of the running time. From fastest to slowest, running times include $O(1)$: constant; $O(\log_2 n)$: logarithmic; $O(n)$: linear; $O(n \log_2 n)$: linearithmic; $O(n^2)$: quadratic; and $O(2^n)$: exponential. [Lecture 21]

**binary file**: A file stored in binary format, which is typically more efficient but is not able to be read by humans. [Lecture 6]

**binary search**: A process for searching in a sorted list in which you repeatedly check the midpoint of the list and then search in either the upper or lower half of the remaining list. [Lecture 20]

**binary search tree**: A binary tree in which the nodes contain items ordered such that for any node, the left descendants are all smaller items and right descendants are all larger items. [Lecture 22]

**binary tree**: A tree in which each node will have at most two children. [Lecture 22]

**blit** (**block-level image transfer**): A method of combining two images by copying one image onto the other at a particular location. [Lecture 15]

**Boolean**: A type of value that can be true or false. [Lecture 3]

**Boolean operator**: An operator that is applied to Boolean variables. The common operators are "and," "or," and "not." [Lecture 3]

**bottom-up design**: Creating a complex function by tying together existing basic functions. [Lecture 14]

**branching**: The result of having conditionals within code. The code is said to have branching, because any one execution can follow only certain "branches" of the program. [Lecture 3]

**breadth-first search**: Algorithm that searches a graph from a starting node to find a path to another node by examining all nearby nodes before nodes farther away. Can be used to find the shortest path from one node to another in an unweighted graph. [Lecture 23]

**breakpoint**: In a debugger, a point at which the execution of a program will be stopped so that the current values of variables can be examined. [Lecture 11]

**buffer**: Queue of data or events to be handled. For example, user input, such as mouse movements or keyboard key presses, are often stored in an event buffer. [Lecture 19]

**bug** (also called **error**, **fault**, or **defect**): An error created in the process of programming. [Lecture 11]

**call**: To cause a function to execute. A function will be "called" by the main program or by another function, and this will cause the function itself to be executed. [Lecture 9]

**callback function**: In event-driven programming, the function that is called by the event monitor to respond to a particular event. [Lecture 15]

**call stack** (also called **control stack**, **runtime stack**, or **frame stack**): Function activation records that keep track of all the variables and data defined in that part of the program. [Lecture 19]

**central processing unit** (**CPU**): The processor for the computer. This executes the commands and performs operations. [Lecture 2]

**chaining**: In hash tables, refers to making a list of all items that map to the same hash value. [Lecture 19]

**child class** (also called **derived class** or **subclass**): A class that inherits attributes and methods from a parent class. [Lecture 18]

**child node**: A node that is reachable by an edge from a given node and that is one level farther away from the root node. [Lecture 22]

**class**: A way to group both data (defined in attributes) and functions (defined in methods). Can be thought of as a type of variable. Classes form the heart of object-oriented programming. [Lecture 17]. *See also* **object**.

**closing**: To complete work with a file from within a program. Closing the file ensures that it will not be corrupted by the program. [Lecture 6]

**command**: An instruction given to the computer. [Lecture 1]

**comparison operator**: An operator that can compare two values, giving a Boolean result. Common examples are equality and inequality, greater than, and less than. [Lecture 3]

**compiler**: A program to convert code from a programming language into machine instructions. Compilers will take all code in a program together and process it into a set of machine instructions. [Lecture 1]

**concatenation**: Combining two strings to form a new string. [Lecture 2]

**conditional**: A programming construct that checks some condition to determine whether it is true or false and then executes different code depending on the result of that check. [Lecture 3]

**connected graph**: A graph in which there is some sequence of edges connecting every pair of nodes. [Lecture 22]

**constructor**: A function called when an object is instantiated. In Python, this is the "__init__" function. [Lecture 17]

**cycle**: A sequence of edges that, when followed, returns to the starting node. [Lecture 22]

**data structure**: A way of organizing data systematically so that certain operations on that data can be performed easily. Usually used as a way to organize large amounts of similar data. [Lecture 19]

**debugger**: A program that can be used to examine the state of a program at any time. Debuggers are often part of an integrated development environment and can be used to step through a program line by line. [Lecture 11]

**default parameter**: A value to be assigned to a parameter in a function if that parameter is not specified by the user. [Lecture 10]

**dictionary**: A data structure for storing key-value pairs. Implemented using a hash table. [Lecture 19]

**directed edge**: An edge that connects from a source node to a destination node. [Lecture 22]

**distributed computing** (also called **grid computing**): Using physically distinct computers that are loosely coupled with each other (such as over a network) to perform a computation. [Lecture 24]

**divide and conquer**: Taking a large problem and dividing it into several smaller problems that are easier to solve. Typically, this involves dividing a large data set into two or more smaller data sets that can be processed more easily. [Lecture 21]

**docstring**: A (sometimes multiline) string description of a function's behavior; docstrings can be printed when we ask for help about a function. [Lecture 9]

**edge**: A connection between two nodes. Edges can be weighted or unweighted. [Lecture 22]

**edge cases** (also called **corner cases**): Situations that are at the "boundaries" of a range of inputs. These should be a part of any test suite. [Lecture 11]

**encapsulation**: The concept of grouping data and functions to operate on that data together in a package. Encapsulation is provided by classes and objects and is a primary benefit of object-oriented programming. [Lecture 17]

**equality/inequality operator**: An operator to check whether two values are (or are not) equal. In Python, the equality operator is the double equal sign, ==, which is distinct from the single equal sign assignment operator. The inequality operator in Pytyon is !=. [Lecture 3]

**escape character**: In a string, a character used to help specify special non-alphanumeric information, such as line breaks and tabs. In Python, this is the forward slash: \.

**event**: An occurrence, typically coming as input from an external source, that we want the computer to respond to. Common examples are keyboard button presses, mouse motion, and mouse clicks. [Lecture 15]

**event-driven programming**: A programming approach where functions are written to respond to events. Commonly used in interactive graphical programs. [Lecture 15]

**event monitor**: A software control function that takes in events, such as keyboard presses or mouse movements, and makes sure that the appropriate function gets called in response. [Lecture 15]

**exception**: A way of identifying that a runtime error has occurred. Exceptions are raised when a runtime error occurs and are handled later. In Python, the "try...except" commands are used to handle exceptions that occur. [Lecture 11]

**execute** (also called **run**): To have a computer process a set of instructions given in a program. [Lecture 1]

**expression**: A portion of code that, when executed, produces a value from some combination of values, variables, and operations. [Lecture 2]

**file**: A set of data stored in secondary or tertiary memory. Programs must read a file into main memory to use it or can write from main memory to a file. [Lecture 6]

**floating-point number** (also called **float**): A number containing a decimal point, typically with some values specified before and after the decimal. For example, 3.14159 is a floating-point number. [Lecture 2]

**flowchart**: A method for defining an algorithm by creating a graphical layout of the instructions. Shapes are used to describe operations, and arrows are used to indicate the sequence of steps. [Lecture 20]

**for loop**: A loop that repeats a certain number of times, with the number of times controlled by an iterator. [Lecture 5]

**function** (also called **procedure**, **routine**, **subroutine**, or **method**): In programming, a command that (possibly) takes some input, performs some action, and then (possibly) returns a result. [Lecture 9]

**function activation record**: A region of memory set aside for a function to work in, including the function's parameters and any variables defined in the function. The function activation record is destroyed when the function returns. [Lecture 10]

**function body**: The part of the function definition besides the header, describing the actions the function will take, along with when and what to return. [Lecture 9]

**function header**: The initial line of a function definition, giving its name and describing its parameters. [Lecture 9]

**global variable:** A variable that is in scope both outside and within a function. Declaring variables as global is a way to initialize certain types of data without using objects. [Lecture 10]

**graph**: A data structure used to store items and their relationships to each other. Items are stored at nodes, and the relationships between items are stored by edges connecting nodes. [Lecture 22]

**graphical user interface (GUI)**: An interface for a program in which a user interacts with graphical elements such as buttons or locations on the screen. It is implemented using event-driven programming. [Lecture 15]

**Gustafson's law**: A rule that determines how large of a problem can be handled, given more processors. [Lecture 24]

**hardcoding**: When a specific value is set within the code, rather than being read in from a user. Hardcoding tends to be easier to code in the short term but is less flexible in the long term. [Lecture 12]

**hash function**: A function that can take a key phrase and convert it to a number that can be used to index into a list being used for a hash table. [Lecture 19]

**hash table**: Data structure that maps data with indices in a very large range into a smaller set of indices that can be stored more compactly. The index for a data element is called the key. [Lecture 19]

**index:** A number assigned to the position for each variable in a list. By convention, the first index number is usually zero. [Lecture 7]

**indirection**: When an intermediate structure is used to describe a connection between two entities. For example, rather than storing a list of entities, instead there might be a list of indices stored, with the entities stored in a separate structure found by examining each index. [Lecture 22]

**infinite loop**: A loop that repeats without ever ending. [Lecture 5]

**inheritance**: When one class (the child class) is defined to have all the attributes and methods of another class (the parent class). [Lecture 18]

**in-place sort**: A sort in which the original list is modified to put the elements in sorted order. [Lecture 20]

**input/output (I/O)**: The interface between a computer and the outside world. Input can come from many possible sources, including keyboard or mouse input, network connections, sensors, etc. Output can be text or a graphical display that is output to the screen, a printed document, data sent over the network, commands to an attached device, etc. [Lecture 2]

**insertion sort**: A sort in which one new element is repeatedly inserted into an already sorted list. [Lecture 20]

**instantiation**: Creating a new object. This happens when the object is first encountered in a program. [Lecture 17]

**integer**: A number with no fractional component. Integers include -2, -1, 0, 1, 2, etc. [Lecture 1]

**integrated development environment (IDE)**: A software program used to program code. An IDE will include an editor in which code can be written and easily used methods for compiling and executing code. There are typically many more tools that are also included, such as a debugger and hint systems. [Lecture 1]

**interpreter**: Like a compiler, an interpreter is a program to convert code from a programming language into machine instructions. Unlike a compiler, interpreters will convert code one line at a time as it is needed for execution. [Lecture 1]

**iteration**: One pass through a loop. [Lecture 5]

**iterative development**: Developing software by starting with a simple, basic implementation, then adding small amounts to the software, making sure that the software is working before going further. [Lecture 4]

**iterator**: A variable that gets initialized to a starting value and is incremented for each iteration of a loop, until it reaches a maximum value. [Lecture 5]

**key**: In a hash table, the value that is used as an “index” into the table. Keys can be any immutable data type. [Lecture 19]

**keyword**: In a programming language, any of a several special words reserved for exclusive use in commands and not permitted as identifiers for variables, functions, objects, and so on. Python keywords include “print,” “import,” “class,” “global,” “finally,” “True,” and “False.” [Lecture 2]

**keyword argument**: When a parameter value is specified by giving the name of the parameter. While other parameters are processed from left to right, keyword parameters can be specified in a different order. [Lecture 10]

**library**: A collection of functions, classes, and variable definitions that can be imported into another program to extend its capabilities. [Lecture 12]

**list** (also called **array**): Data stored sequentially so that it can be referred to by its index. Typically, the data in a list will be of the same type. [Lecture 7]

**logic error**: An error that causes the program to produce incorrect results, due to incorrect design of the program. [Lecture 11]

**loop**: A programming construct that repeats a set of commands over and over. [Lecture 5]

**loop invariant**: A condition that is true at the beginning of every iteration of a loop. Helpful in algorithm design. [Lecture 20]

**main memory**: Short-term working memory that holds the data currently being used by the computer. This is separate from the CPU but is connected directly. Main memory holds the variables that programs use. [Lecture 2]

**main program:** Part of the computer program that is executed first, apart from any function definitions. [Lecture 10]

**memory**: Part of the computer that can store data. Memory is arranged in a memory hierarchy. [Lecture 2]

**memory hierarchy**: Arrangement of memory in the computer. Higher levels of the hierarchy are much faster and easily accessible to the processor but are more expensive and limited in size. The hierarchy includes registers within the CPU at the highest level, then cache memory (sometimes divided into multiple levels itself) that is near the CPU, then main memory that is connected directly to the CPU on the motherboard, then secondary memory (stored on a hard disk or similar drive), and then tertiary memory (stored remotely). [Lecture 2]

**mergesort**: A recursive sorting routine in which a list is split into two halves, each of which is then sorted recursively. The sorted lists are then merged together. [Lecture 21]

**method** (also called **member function** (C++)): A function defined for a particular object or class. Parallel to how attributes define data. [Lecture 17]. *See also* **attribute**.

**model**: The laws and rules that are assumed to govern a particular process. [Lecture 16]

**module**: A Python library, ending with a ".py" extension, just like other Python programs. [Lecture 12]

**Monte Carlo simulation**: A simulation in which multiple random values are used to simulate a range of possible outcomes. [Lecture 16]

**motherboard**: A circuit board in the computer that is used to connect various components, including the processor, main memory, secondary and tertiary storage connections, input/output devices, networks, etc. [Lecture 2]

**multiprocessing**: Using multiple processors simultaneously to execute different computing processes in parallel. [Lecture 24]

**mutable**: Data that can change when it is passed as a parameter to a function. Lists and objects are mutable data types. Mutable data is actually a reference; when passed as a parameter, the reference value will not change, but the values in memory at that reference can change. [Lecture 10]

**nesting**: When one programming construct occurs within another of the same type. For example, if a conditional contains another conditional, or a loop contains another loop, these are said to be nested. [Lecture 3]

**node**: A vertex in a graph that is used to store information about the items or entities. Nodes are connected by edges. [Lecture 22]

**object**: A specific instance of a class. An object is a particular variable, with a type given by the class it belongs to. [Lecture 17]

**opening**: To prepare a file for reading, writing, or appending from within a program. [Lecture 6]

**operation**: A basic action performed on data, such as addition or other basic arithmetic, from applying an operator or calling a function. Operations are performed by the processor. [Lecture 2]

**operator**: A programming construct that computes a new value from some basic values. Operators include addition and other arithmetic, comparisons, and indexing. [Lecture 1]

**out-of-place sort**: A sort in which a new, sorted, list is created while leaving the original list unchanged. [Lecture 20]

**package**: A collection of modules. [Lecture 12]

**parallel computing**: Computing more than one value simultaneously. [Lecture 24]

**parameter**: Values passed into a function. Values for the parameters (sometimes called arguments) are specified when the function is called. Within the function, the parameters are variables that are listed in the header and defined when the function first begins. [Lecture 9]

**parameter passing**: Copying the value from the function call (sometimes called the argument) into the memory set aside for the parameter variable within the function activation record. [Lecture 10]

**parent class** (also called **base class** or **superclass**): A class that defines attributes and methods that are inherited by a child class. [Lecture 18]

**parent node**: A node in a tree that is one edge closer to the root than a given node. [Lecture 22]

**path**: Designation for the location of a file within a computer's storage system. The path tells where to find a file relative to the computer or relative to the current program being executed. [Lecture 6]

**pivot**: In the quicksort algorithm, the value used for separating the list into smaller and larger parts. [Lecture 21]

**polymorphism**: When a single function can take on different implementations. Typically happens when different related classes implement the same function. [Lecture 18]

**procedural programming**: A long-standing method for programming where functions are created to handle all the various tasks that are needed. Programs are built by calling functions in the appropriate order. [Lecture 13]

**process**: A program, or part of a program, that can be executed on a computer. [Lecture 24]

**processor**: The part of the computer that performs computations. The processor has only a limited set of basic operations that it can perform. [Lecture 2]

**program**: A set of commands given to a computer. [Lecture 1]

**programming language**: A language developed for people to be able to easily and precisely give commands to a computer. Examples include Python, Java, C, C++, Fortran, BASIC, COBOL, etc. [Lecture 1]

**pseudocode**: A method for defining an algorithm by writing instructions similar to computer code. The syntax of pseudocode is flexible and generic and does not typically match any particular language. [Lecture 20]

**Python Package Index** (**PyPI**): A repository for thousands of Python modules and packages that are not part of the Python standard library. See https://pypi.python.org/pypi. [Lecture 12]

**Python standard library**: A collection of about 250 modules that is automatically installed as part of Python. See https://docs.python.org/3/library/. [Lecture 12]

**queue**: A data structure that allows storage and retrieval of data, following a first-in, first-out order. Items are added using an enqueue command and removed using a dequeue command. [Lecture 19]

**quicksort**: A recursive sorting routine in which a pivot value is chosen, and then all other values are separated into a larger list and a smaller list, which are then sorted recursively. [Lecture 21]

**recursion**: A process in which a function calls itself, typically with a different set of parameters. [Lecture 21]

**reference** (also called **pointer**): A location in memory at which some larger amount of data is contained. Data in memory can be changed without changing the value of the reference itself. [Lecture 10]

**remote procedure call (RPC)**: Calling and executing a function on a different computer. [Lecture 24]

**root**: A node in a tree designated as the one from which all other nodes will be traced. [Lecture 22]

**runtime error**: An error that occurs when the program is running, causing the program to fail. Runtime errors can be dealt with using exceptions. [Lecture 11]

**scope**: The region of a program in which a variable or function is defined and usable. [Lecture 10]

**search**: The process of finding an element within some larger collection, such as a list. [Lecture 20]

**selection sort**: A sort in which the smallest element is repeatedly selected from the remaining elements. [Lecture 20]

**set**: A data structure for storing items with no fixed order and no duplicate values. It supports the common mathematical set operations. [Lecture 19]

**side effect**: Actions that a function takes that are not obviously part of the function's behavior from its definition. For example, a function might change a value of a variable that is not a parameter. [Lecture 9]

**simulation**: The process of taking a model and set of initial conditions and determining how the process progresses. [Lecture 16]

**slicing**: Generating a subset of a list. [Lecture 7]

**sort**: A basic algorithm for taking a list of values and creating a list in which the values are ordered from smallest to largest. [Lecture 20]

**spawn**: When one process or thread generates another process or thread, to be run in parallel. [Lecture 24]

**stack**: A data structure that allows storage and retrieval of data, following a last-in, first-out order. Items are added using a push command and removed using a pop command. [Lecture 19]

**state**: The particular set of values describing a system at a particular point in time. [Lecture 16]

**statement**: A line of code that gives an instruction to a computer to take some action. [Lecture 1]

**storage**: Alternate term for secondary and tertiary memory. Refers to memory that is not immediately accessible to programs running on the computer; data in storage must be brought into main memory to be used. [Lecture 2]

**string**: A sequence of characters. [Lecture 2]

**stub**: A function inserted during software development that doesn't yet do what it's intended to do, but is just enough that everything around it can run anyway. "Stubbing out" a program means that we are writing stub functions for that program. [Lecture 13]

**syntax error**: A bug that is due to writing code that is not valid. Programs with syntax errors cannot execute. [Lecture 11]

**testing**: The way to debug code, by running code using specific input and determining if output is correct. [Lecture 4]

**test suite**: A set of tests that are run on code to make sure that it is working correctly. As new features are added, the test suite should be continuously verified as working. [Lecture 11]

**threading**: Allowing multiple computer processes to run in parallel. Each separate process is run in a thread. [Lecture 24]

**time step**: The amount of time by which a simulation advances in one round of computation. [Lecture 16]

**top-down design**: Taking a complex task and breaking it into simpler parts, repeatedly, until the basic parts are "obvious." [Lecture 8]

**tree**: A particular type of connected graph that does not contain a cycle. One of the most widely used data structures; many algorithms have been developed just for trees. [Lecture 22]

**tuple**: Like a list, but with fixed length and types. Like a list, index values can be used to access elements of a tuple. Tuples are not mutable. Tuples will often combine different types of data in one tuple. [Lecture 7]

**turtle graphics**: Graphics created by simulating a small robot "turtle" that carries a pen as it moves around, tracing the path it follows. [Lecture 14]

**type**: The way that data stored in a variable should be interpreted by the computer. Each variable and value will have a type, such as an integer, a floating-point number, a string, etc. [Lecture 2]

**undirected edge**: An edge that connects two nodes, with no distinction for a source and destination. [Lecture 22]

**value**: The information stored in a variable. A value can be a number, string, or other type of data. [Lecture 2]

**variable**: A memory location with a given name that can hold a value. [Lecture 2]

**virtual function**: A function that is part of an abstract interface. [Lecture 18]

**weight**: A value stored along an edge, indicating something about the relationship between the nodes it connects. [Lecture 22]

**while loop**: A loop that repeats as long as some condition is true. [Lecture 5]

**widget**: In a graphical user interface, individual items such as sliders or buttons that appear in a window and that the user interacts with to generate events. [Lecture 15]

# Python Commands

**`break`**: Exit a loop or conditional immediately. [Lecture 5]

**`class`**: Define a new class. [Lecture 17]

**`continue`**: Begin the next iteration of the loop. [Lecture 5]

**`def`**: Define a function. [Lecture 9]

**`dict`**: Define a dictionary. [Lecture 19]

**`file.close`**: Close a file. [Lecture 6]

**`file.write`**: Write a string to a file. [Lecture 6]

**`file.read`**: Read the whole file as a string. [Lecture 6]

**`file.readline`**: Read a line from a file as a string. [Lecture 6]

**`float`**: Convert a value to a floating-point number. [Lecture 2]

**`for... in`**: For loop with an iterator proceeding through a given set of values. [Lecture 5]

**`from ... import`**: Import a module or package. [Lecture 12]

**`global`**: Make a variable equivalent to the global variable of the same name. [Lecture 10]

**`if ... elif ... else`**: Conditional statement to execute different code depending on Boolean value(s). [Lecture 3]

**`int`**: Convert a value to an integer. [Lecture 2]

**`input`**: Print text to the screen, and then get input from a user and return. [Lecture 2]

**`len`**: Get length of a list. [Lecture 7]

**`list.append`**: Add an element onto the end of a list. [Lecture 7]

**`list.sum`**: Sum the elements in a list. [Lecture 7]

**`open`**: Open a file for reading, writing, or appending. [Lecture 6]

**`quit`**: Quit the program. [Lecture 9]

**`string.split`**: Split a string into multiple strings based on a character separator. [Lecture 8]

**`range`**: Generate values in a range of numbers. [Lecture 5]

**`return`**: Return from a function, returning a value if specified. [Lecture 9]

**`pass`**: Do nothing. Used when no actual command is wanted. [Lecture 18]

**`print`**: Sends output to screen. [Lecture 1]

**`set`**: Define a set. [Lecture 19]

**`str`**: Convert a value to a string. [Lecture 2]

**`try ... except ... finally`**: Try to execute code, and if an exception is raised, handle it in the except section. [Lecture 11]

**`while`**: While loop continuing while some condition is true. [Lecture 5]

**`with ... as`**: Use to open a file as a given name and close on completion. [Lecture 6]

# Python Modules and Packages Used

The Python standard library includes hundreds of modules that are automatically installed with every version of Python. To use these modules, simply import them into your program. See https://docs.python.org/3/library/.

The Python Package Index (PyPI) includes thousands of Python modules and packages of varying degrees of completeness and support. To use these, you must first download and install them on your computer. This can usually be done through the pip interface, by typing "python –m pip install <package name>." You can also visit the PyPI page for the module or the website devoted to the module (if there is one) to find more details and download it directly.

## Python Standard Library Modules

**math**: Math utilities. [Lecture 12]

**webbrowser**: Open and redirect web browser. [Lecture 12]

**shutil**: Shell utilities. [Lecture 12]

**turtle**: Turtle graphics. [Lectures 12, 14]

**calendar**: Create and display calendars. [Lecture 12]

**time**: Time utilities. [Lecture 12]

**statistics**: Statistical evaluation utilities. [Lecture 12]

**os**: Operating system commands. [Lecture 12]

**random**: Random numbers and data. [Lectures 13, 16]

**tkinter**: Graphical user interface setup and handling. [Lecture 15]

**json**: Converting data to/from JSON format, and then writing and reading JSON strings to files. [Lecture 18]

**pickle**: Converting Python data to a binary format and writing to or reading from a file. [Lecture 18]

**multiprocessing**: Create and run multiple processes in parallel. [Lecture 24]

**subprocess**: Spawn additional processes in the operating system. [Lecture 24]

## Python Package Index Modules

**numpy**: Numerical manipulation (http://www.numpy.org/). [Lecture 12]

**pyglet**: Graphics, mouse input, and game functionality (http://pyglet.readthedocs.org/). [Lecture 15]

**matplotlib**: Graphing and plotting charts (http://matplotlib.org/). [Lecture 16]

# Bibliography

There are a large number of references for Python programming, including several books and websites. The Python tutorial on the official Python website is probably the most useful standard reference: https://docs.python.org/3/tutorial/.

Most Python books will go into much greater detail in some features or applications of the language than others, so the "best" book will often depend on which topic you wish to learn more about. The following are recommended as good books, overall, for further study.

---

Gries, Paul, Jennifer Campbell, and Jason Montojo. *Practical Programming*. 2nd ed. Pragmatic Bookshelf, 2013. This book presents a well-organized introduction to Python.

Lambert, Kenneth. *Fundamentals of Python: Data Structures*. Cengage Learning PTR, 2013. This book uses Python to introduce some of the slightly more advanced ideas in computer science: data structures and algorithms.

Lubanovic, Bill. *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media Inc., 2015. This book provides an overview of Python, including many of the more advanced features of the language.

Matthes, Eric. *Python Crash Course*. No Starch Press, 2015. This book is in two parts: The first provides an introduction to Python, and the second presents three in-depth projects: an arcade-style game, a data visualization, and a web application.

Sweigart, Al. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press, 2015. This book—which is available for free online at https://automatetheboringstuff.com—is in two parts: The first presents an overview and introduction to Python, and the second presents several detailed examples of how to use various modules to build interesting and useful applications.

Zelle, John. *Python Programming: An Introduction to Computer Science*. 2nd ed. Franklin, Beedle & Associates, 2010. This provides a thorough and well-organized introduction to Python and computer science basics. It is organized to support a college-level course in Python.