THE GREAT COURSES®

Topic
Science & Mathematics

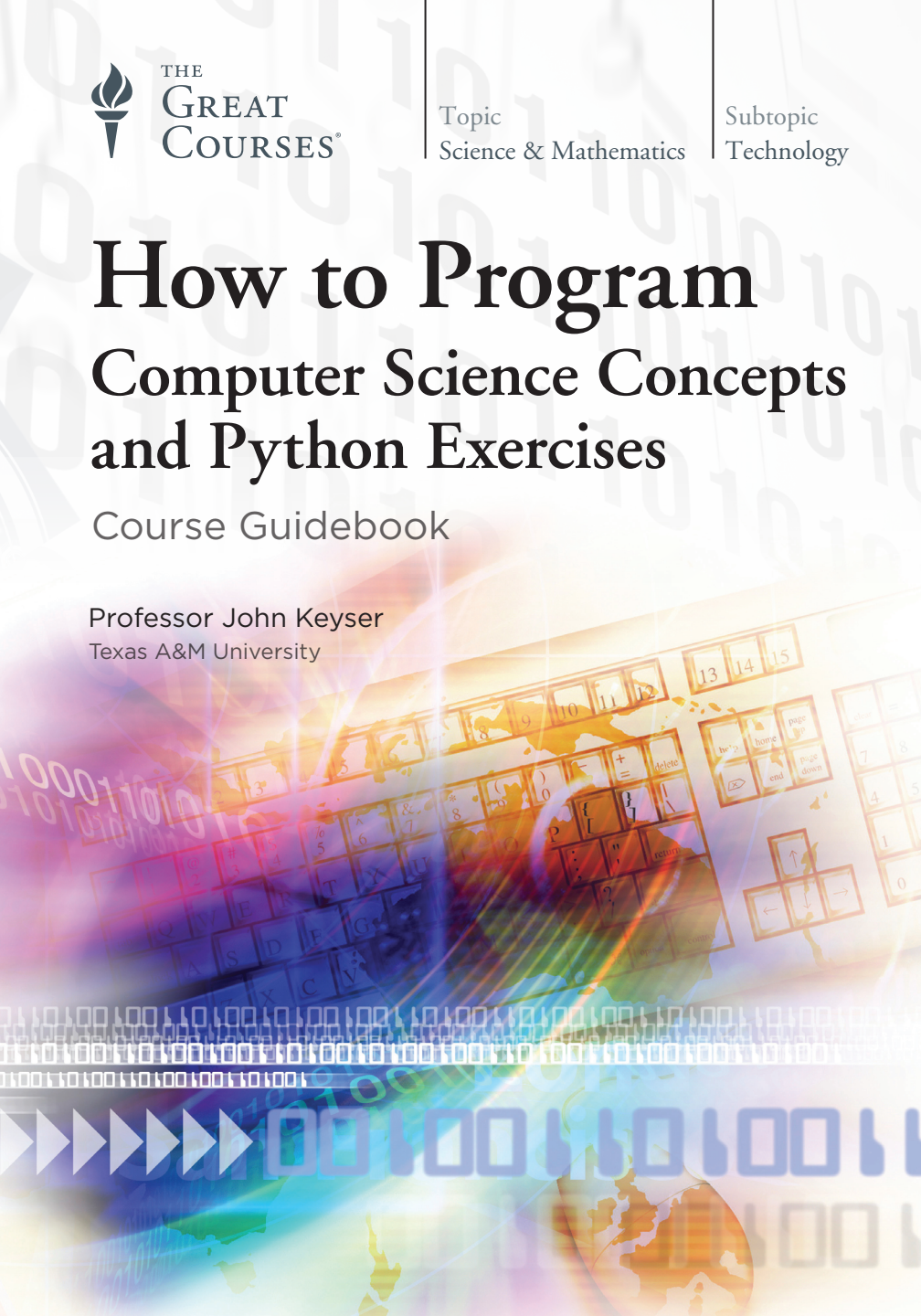
Subtopic
Technology

How to Program

Computer Science Concepts and Python Exercises

Course Guidebook

Professor John Keyser
Texas A&M University



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]

Professor Biography i
Course Scope 1
Installing Python and PyCharm 4

[LECTURE GUIDES]

LECTURE 1
What Is Programming? Why Python? 9

LECTURE 2
Variables: Operations and Input/Output 16

LECTURE 3
Conditionals and Boolean Expressions 26

LECTURE 4
Basic Program Development and Testing 39

LECTURE 5
Loops and Iterations 48

LECTURE 6
Files and Strings 59

LECTURE 7
Operations with Lists 66

LECTURE 8
Top-Down Design of a Data Analysis Program 79

LECTURE 9
Functions and Abstraction 92

LECTURE 10
Parameter Passing, Scope, and Mutable Data 103

LECTURE 11
Error Types, Systematic Debugging, Exceptions 113

LECTURE 12
Python Standard Library, Modules, Packages. 124

LECTURE 13
Game Design with Functions 132

LECTURE 14
Bottom-Up Design, Turtle Graphics, Robotics 146

LECTURE 15
Event-Driven Programming. 157

LECTURE 16
Visualizing Data and Creating Simulations. 170

LECTURE 17
Classes and Object-Oriented Programming 182

LECTURE 18
Objects with Inheritance and Polymorphism. 193

LECTURE 19
Data Structures: Stack, Queue, Dictionary, Set. 205

LECTURE 20
Algorithms: Searching and Sorting 216

LECTURE 21
Recursion and Running Times 228

LECTURE 22
Graphs and Trees 238

LECTURE 23
Graph Search and a Word Game 248

LECTURE 24
Parallel Computing Is Here 260

[SUPPLEMENTAL MATERIAL]

Answers. 269
Glossary 293
Python Commands 310
Python Modules and Packages Used 312
Bibliography 314

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

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.”

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
```

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

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

```
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")
```

```
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

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.

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.

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.)

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.

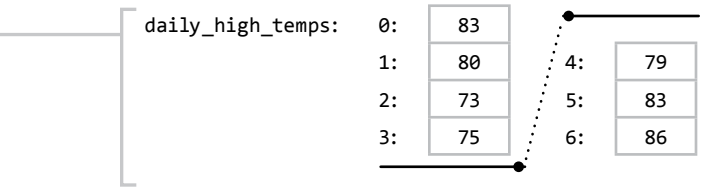
› A variable always corresponds to a box in memory. When we think of a list or array, we have a whole stack of those boxes. We give one name to that whole stack of boxes. This is going to let us organize our data—to keep common things together.

› 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.

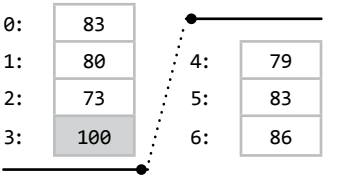
```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
print (daily_high_temps[4])

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:
daily_high_temps:   0:  83        5:  83
                    1:  80        6:  86
                    2:  73
                    3:  75
                    4:  79        x:   7
```

› 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.

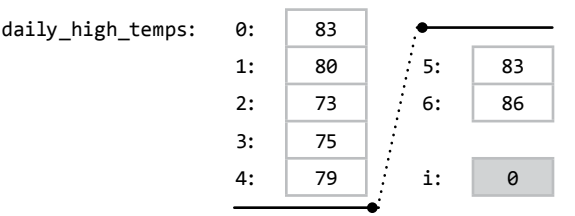
› We can use a for loop, as follows. We don't use a "range" command—we just say for all the *i* in the list. This code is going to print off all the values in the list, just like before.

```
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.

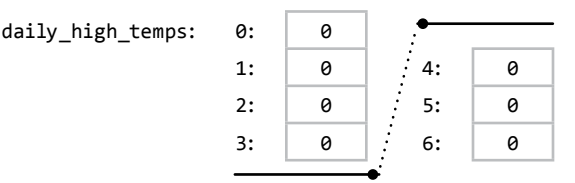
› Note that *i* is getting assigned the value from the list. So, if we modify the value of *i*, it does not change the value in the list.

```
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).

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.

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 = input("Type in your credit card number:")
print("Warning! Use program at your own risk.")
expiration = input("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 = input("Type in your credit card number:")
warn()
expiration = input("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 izs 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
```

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)
```

testscore

numcorrect:	17
total:	20

a:	12
b:	20

› 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)
```

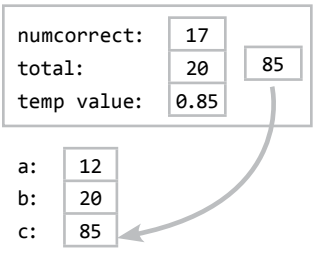
testscore

numcorrect:	17
total:	20
temp value:	0.85

a:	12
b:	20

› 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.

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.

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.

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.

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*.

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.

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]

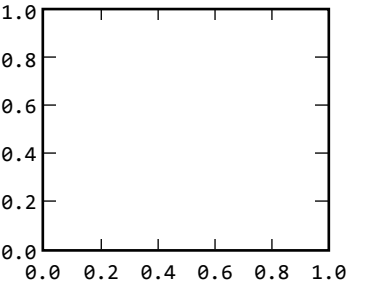
› One of the best packages to create visualizations of data is matplotlib. It has a very wide range of capabilities and is probably the most well-known and popular Python package for creating plots, graphs, and charts from data.

› 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.

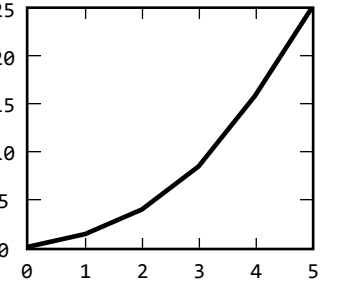
› We learn how to use graphical tools like this by looking at the documentation. In particular, the API is the application programming interface. The API documentation lists the various commands that are provided and the details of how each is used.

› 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.

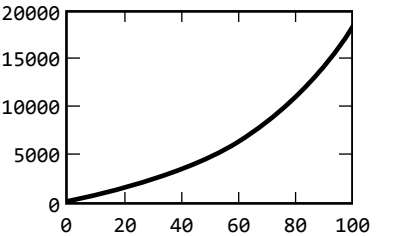
```
#Set initial conditions
time = 0
balance = 1000
#Set list to store data
timelist=[time]
balancelist=[balance]
while (time < 10):
    #Increase balance and time
    balance += balance*0.03
    time += 1
    #Store time and balance in lists
    timelist.append(time)
    balancelist.append(balance)
#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.

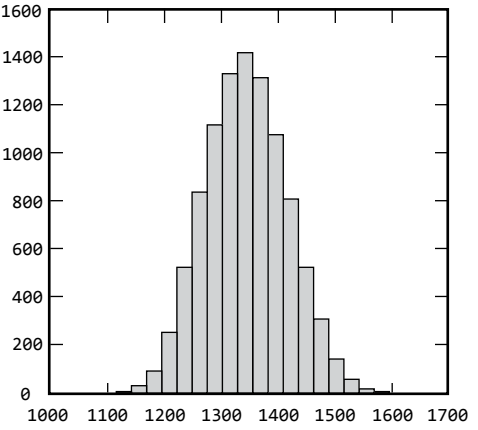
```
import random
def ChangeInBalance(initial_balance):
    rate = random.uniform(0.0, 0.06)
    return initial_balance*rate
number_years = 10
number_sims = 100
final_balances = []
for i in range(number_sims):
    #Set initial conditions
    time = 0
    balance = 1000
    while (time < number_years):
        #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.

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.

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”?

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), 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)
```

20 Algorithms: Searching and Sorting

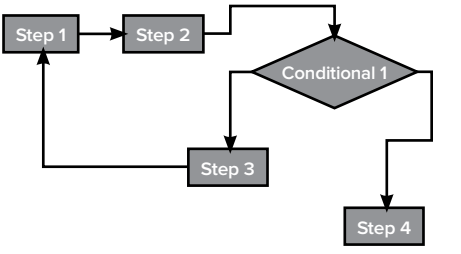
Algorithms form the core of computer science. Algorithms are how we describe what we actually want the computer to do. Computer programming is really just the process of taking an algorithm and converting it into a program that the computer understands. The program is the concrete incarnation of the 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
```

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).

22 Graphs and Trees

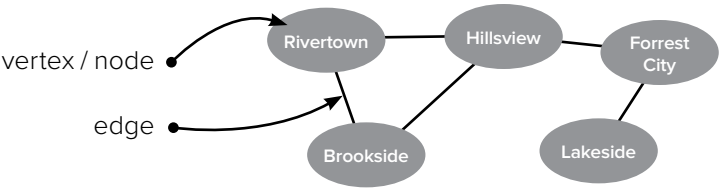
Graphs offer a structure for capturing an incredibly wide diversity of relationships and the connections that are formed all over. They represent connections between locations, between people, among species, and among data. These and many other relations are well represented on a graph. Once relationships 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.

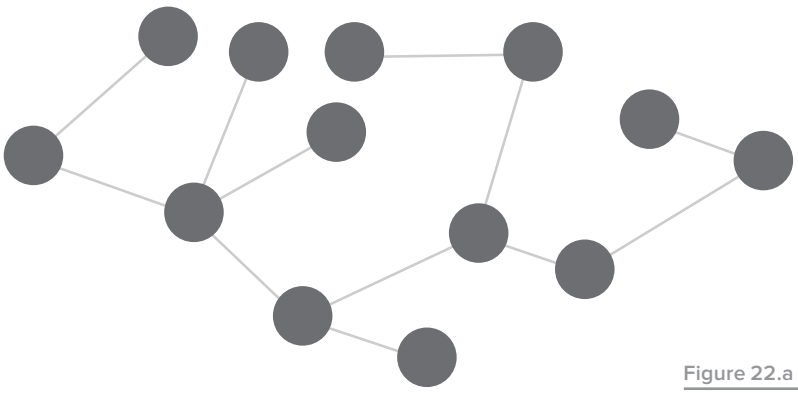


Figure 22.a

- 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.

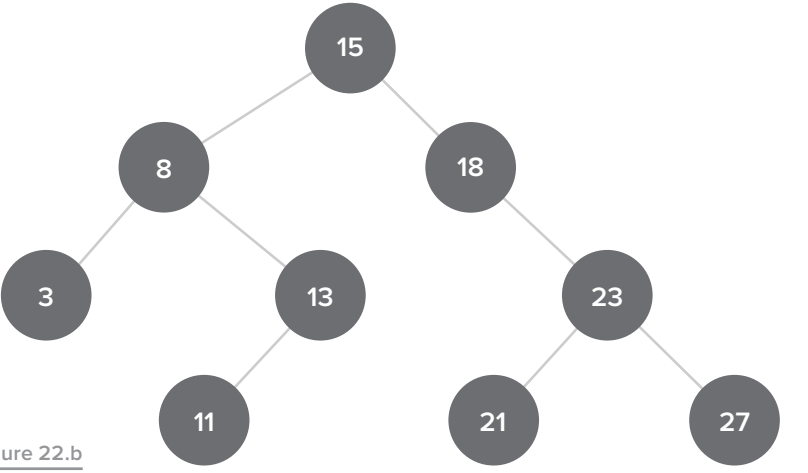


Figure 22.b

› 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()
```

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.

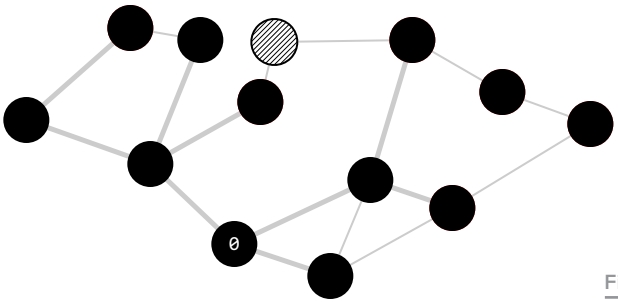


Figure 23.a

- 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.

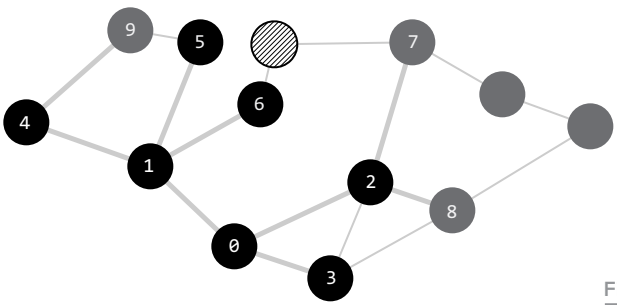


Figure 23.b

- 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?

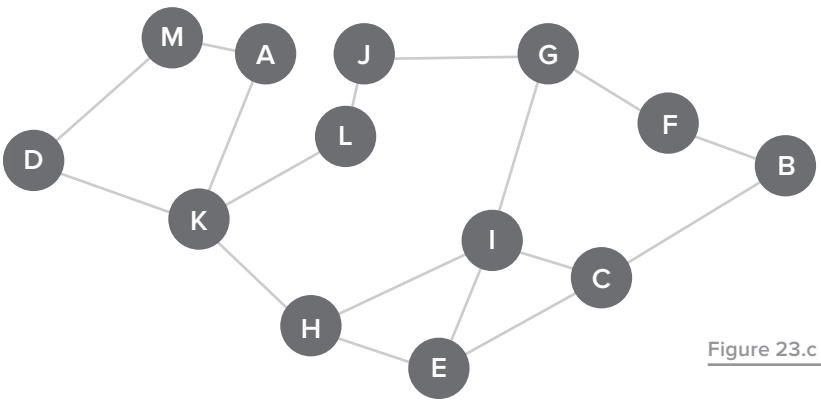


Figure 23.c

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"?

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

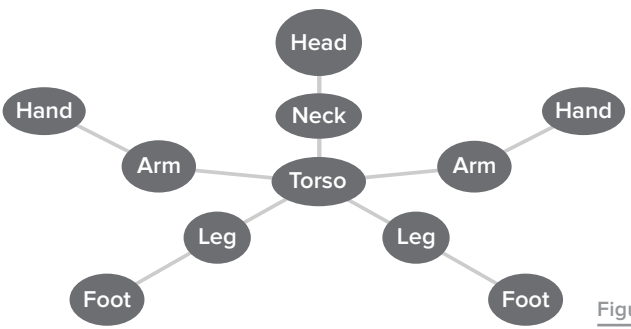


Figure A22.a

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

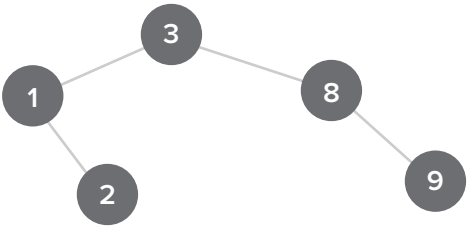


Figure A22.b

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.