Computing Science 120 • Study Guide

Introduction to Computing Science and Programming I

Summer 2024 Edition

by

Greg Baker

Faculty of Applied Sciences
Simon Fraser University

Contents

List of Figures

Course Introduction

[Most of the material in this introduction will apply to all offerings of CMPT 120, but some details will vary depending on the semester and instructor. Please check your course web site for details.]

Welcome to CMPT 120, "Introduction to Computing Science and Programming I". This course is an introduction to the core ideas of computing science and the basics of programming. This course is intended for students who do not have programming experience. If you have done a significant amount of programming, you should take CMPT 126 instead.

This course isn't intended to teach you how to use your computer. You are expected to know how to use your computer: you should be comfortable using it for simple tasks such as running programs, finding and opening files, and so forth. You should also be able to use the Internet.

Here are some of the goals set out for students in this course. By the end of the course, you should be able to:

- Explain some of the basic ideas of computing science.
- Explain what computer programming is.
- Create algorithms to solve simple problems.
- Implement computer programs in the Python programming language.
- Apply the core features of a programming language to solve problems.
- Design programs that are easy to understand and maintain.

Keep these goals in mind as you progress through the course.

Learning Resources

Study Guide

The *Study Guide* is intended to guide you through this course. It will help you learn the content and determine what information to focus on in the texts.

Some suggested readings for each section are listed at the beginning of the units. You should also look at the key terms listed at the end of each unit.

In some places, there are references to other sections. A reference to "Topic 3.4," for example, means Topic 4 in Unit 3.

In the *Study Guide*, side-notes are indicated with this symbol. They are meant to replace the asides that usually happen in lectures when students ask questions. They aren't strictly part of the "course."

Web Materials

The web materials for your section of the course will contain assignments and further administrative details. Please refer to them for more information.

Requirements

Computer Requirements

You need to have access to a computer for this course. The labs on campus or your own computer can be used. Instructions on using the labs can be found on the course web site.

Software Requirements

All of the software that you need for this course can be downloaded for free. Links to obtain the course software are on the course web site. Appendix A contains instructions for installing and working with the Python software.

There may be some other pieces of software that you'll need to use when submitting your work. See the course web site for instructions on installing and using these.

Academic Dishonesty

We take academic dishonesty very seriously in the School of Computing Science. Academic dishonesty includes (but is not limited to) the following: copying another student's assignment; allowing others to complete work for you; allowing others to use your work; copying part of an assignment from an outside source; cheating in any way on a test or examination.

If you are unclear on what academic dishonesty is, please read Policy 10.02. It can be found in the "Policies & Procedures" section in the SFU web site.

Cheating on a lab or assignment will result in a mark of 0 on the piece of work. At the instructor's option, further penalties may be requested. Any academic dishonesty will also be recorded in your file, as is required by University policy.

Any academic dishonesty on the midterm or final will result in a recommendation that an F be given for the course.

Part I

Computer Science and Programming

Unit 1

Computing Science Basics

Learning Outcomes

- Define the basic terms of computing science.
- Explain simple algorithms using pseudocode.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapter 1 in *How to Think Like a Computer Scientist*.

Before we start programming, we need to know a little about what *computing science* really is.

Topic 1.1 What is an Algorithm?

The concept of an "algorithm" is fundamental to all of computing science and programming. Stated simply, an algorithm is a set of instructions that can be used to solve a problem.

Figure 1.1 contains one simple algorithm that you might use in everyday life. This algorithm is used in baking and it is written in a way that most people can understand and follow. It is used to make cookies, cakes, muffins, and many other baked goods.

Of course, we aren't going to spend this whole course talking about cooking. (It might be more fun, but the University would get cranky if we started

1. Combine the room-temperature butter and the sugar. Mix until light and fluffy.
2. Add the eggs to the creamed butter and mix to combine.
3. In another bowl, combine the liquid ingredients and mix to combine.
4. Sift together the flour and other dry ingredients.
5. Alternately add the dry and liquid ingredients to the butter-egg mixture. Mix just enough to combine.

Figure 1.1: The "creaming method": an everyday algorithm.

giving cooking lessons in CMPT courses.) Still, the algorithm in Figure 1.1 has a lot in common with the algorithms we will be looking at during this course.

We are more interested in the kinds of algorithms that can be completed by computers. We will spend a lot of time in this course designing algorithms and having the computer complete them for us.

Here's a definition of "algorithm" that most computer scientists can live with: [Anany Levitin, *Introduction to The Design & Analysis of Algorithms*, p. 3]

> An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

There are a few words you should notice about the definition:

unambiguous: When you read an algorithm, there should be no question about what should be done. Is this the case in Figure 1.1?

If you understand cooking terms like "light and fluffy" and "sift together", then you can probably follow most of this recipe. You might have some problem with the last step: you're supposed to "alternately" add the dry and wet ingredients. Does that mean you should do dry-wet-dry? Dry-wet-dry-wet-dry-wet? How many additions should you make?

Recipes in cookbooks are often written with small ambiguities like this either because it doesn't matter what you do or the author is assuming that the reader will know what to do. For the record, the right thing in this case is probably dry-wet-dry-wet-dry.

problem: An algorithm should always present a solution to a particular problem. Each algorithm is designed with a particular group of problems in mind.

In Figure 1.1, the problem must have been something like “Using these ingredients, make muffins.”

legitimate input: An algorithm might need some kind of input to do its job. In the example problem, the inputs are the ingredients; you have to have the correct ingredients before you can start the algorithm.

In addition to having the inputs, they have to be “legitimate”. Suppose we start the instructions in Figure 1.1 with these ingredients: 1 can of baby corn, 1 cup orange juice; 1 telephone. We aren’t going to get very far. In this example, “legitimate” ingredients include sugar, eggs, flour and butter.

If you put the wrong inputs into the algorithm, it might not be able to deal with them.

finite amount of time: This means that if we start the algorithm, we had better finish it eventually.

A recipe that leaves us in the kitchen until the end of time isn’t much good. Suppose we added this step to Figure 1.1:

6. Stir with a fork until the mixture turns red.

No amount of stirring is going to make that happen. If you followed the recipe literally, you’d be standing there stirring forever. Not good.

Many later computing science courses cover algorithms for various problems. For example, CMPT 354 (Databases) discusses algorithms for efficiently storing database information.

Data Structures

When discussing algorithms, it also becomes necessary to talk about *data structures*. A data structure describes how a program stores the data it’s working with.

To carry on with the cooking example, suppose you're trying to find a recipe for muffins. Most people have their recipes in cookbooks on a shelf. To find the recipe, you'd probably select a likely looking book or two and check the index of each one for the recipe you want—that's an algorithm for finding a recipe.

On the other hand, if you have recipes on index cards in a box (because you've just copied the good recipes out of all of your books), you might have to shuffle through the whole pile to find the one you want. If you keep the pile in some kind of order, e.g. alphabetical by the name of the dish it makes, you might be able to find the recipe much faster.

The point? The way you choose to store information can have a big effect on the algorithm you need to work with it. There are many data structures that represent different ways of storing information. We will explore a variety of data structures later in the course.

◈ Courses that discuss algorithms for particular problems generally the corresponding data structures too.

Topic 1.2 What is Computing Science?

Why all this talk of algorithms? This is supposed to be a computing science course: we should be talking about computers. Consider this quote: [Anany Levitin, *Computing Research News*, January 1993, p. 7]

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools, it is about how we use them and what we find out when we do.

Computing science (also known as *computer science*) isn't all about computers. Still, there are certainly a lot of computers around. You will be using computers in this course when you program; most computing science courses involve using computers in one way or another.

Computing science is often defined as: [G. Michael Schneider and Judith L. Gersting, *An Invitation to Computer Science*]

> The study of algorithms, including
>
> 1. Their formal and mathematical properties.

2. Their hardware realizations.
3. Their linguistic realizations.
4. Their applications.

So, computing science is really about algorithms. We will spend a lot of time in this course talking about algorithms. We will look at how to create them, how to implement them, and how to use them to solve problems.

Here is a little more on those four aspects:

1. Their formal and mathematical properties: This includes asking questions like "what problems can be solved with algorithms," "for what problems can we find solutions in a reasonable amount of time," and "is it possible to build computers with different properties that would be able to solve more problems?"
2. Their hardware realizations: One of the goals when building computers is to make them fast. That is, they should be able to execute algorithms specified by the programmer quickly. They should also make good use of their memory and be able to access other systems (disks, networks, printers, and so on). There are many choices that are made when designing a computer; all of the choices have some effect on the capabilities of the final product.
3. Their linguistic realizations: There are many ways to express algorithms so a computer can understand them. These descriptions must be written by a person and then followed by a computer. This requires some "language" that can be understood by both people and computers. Again, there are many choices here that affect how easily both the person and computer can work with the description.
4. Their applications: Finally, there are questions of what actual useful things can be done algorithmically. Is it possible for a computer to understand a conversation? Can it drive a car? Can the small computers in cell phones be made more useful? If the answer to any of these is "yes," then how?

Most of our algorithms won't look much like Figure 1.1. We will focus on algorithms that computers can follow. See Figure 1.2 for an algorithm that is more relevant to a computing science course.

1. Tell the user to pick a secret number between 1 and 100.
2. The smallest possible number is 1; the largest possible is 100.
3. Make a guess that is halfway between the smallest and largest (round down if necessary).
4. Ask the user if your guess is too large, too small or correct.
5. If they say you're correct, the game is over.
6. If they say your guess is too small, the smallest possible number is now the guess plus one.
7. If they say your guess is too large, the largest possible number is now the guess minus one.
8. Unless you guessed correctly, go back to step 3.

Figure 1.2: An algorithm that guesses a secret number between 1 and 100.

The algorithm in Figure 1.2 is designed to solve the problem "guess a secret number between 1 and 100." It meets all of the criteria of the definition of "algorithm" from Topic 1.1.

You may have to spend a few minutes to convince yourself that this algorithm will always eventually guess the correct number, thus finishing in a "finite amount of time". It does. Try a few examples.

This algorithm works by keeping track of the smallest and largest possibilities for the user's secret number. At the start of the algorithm, the number could be anywhere from 1 to 100. If you guess 50 and are told that it's too large, you can now limit yourself to the numbers from 1 to 49—if 50 if too large then the numbers from 51 to 100 must also be too large. This process continues until you guess the right number.

By the end of this course, you should be able to create algorithms like this (and more complicated ones too). You will also be able to implement them so they can be completed by a computer.

Check-Up Questions

- ▶ Can you think of a number where the algorithm in Figure 1.2 will make 7 guesses? 8?
- ▶ What is "legitimate input" for the algorithm in Figure 1.2? What happens if the user enters something else?

Topic 1.3 What is Programming?

Much of this course will focus on *computer programming*. What is programming?

A *computer program* is an algorithm that a computer can understand. This program is often referred to as an *implementation*. Not all algorithms can be implemented with a computer: Figure 1.1 can't. We're interested in the ones that can.

A *programming language* is a particular way of expressing algorithms to a computer. There are many programming languages and they all have different methods of specifying the parts of an algorithm. What you type in a particular programming language to specify an algorithm is often referred to as *code*.

Each programming language is designed for different reasons and they all have strengths and weaknesses, but they share many of the same concepts. Because of this, once you have learned one or two programming languages, learning others becomes much easier.

Why Python?

In this course, we will be using the *Python* programming language. Python is an excellent programming language for people who are learning to program.

You may be wondering why this course doesn't teach programming in C++ or Java. These are the languages that you probably hear about most often.

In this course, we want you to focus on the basic concepts of programming. This is much harder to do in Java and C++: there are too many other things to worry about when programming in those languages. Students often get overwhelmed by the details of the language and can't concentrate on the concepts behind the programs they are writing.

```
write "Think of a number between 1 and 100."
set smallest to 1
set largest to 100
until the user answers "equal", do this:
    set guess to $\lfloor (smallest + largest)/2 \rfloor$
    write "Is your number more, less or equal to guess?"
    read answer
    if answer is "more", then
        set smallest to $guess + 1$
    if answer is "less", then
        set largest to $guess - 1$
```

Figure 1.3: Figure 1.2 written in pseudocode.

C++ and Java are very useful for creating desktop applications and other big projects. You aren't doing that in this course. Languages like Python are a lot easier to work with and are well suited for smaller projects and for learning to program.

Topic 1.4 Pseudocode

Before you start writing programs, you need a way to describe the algorithms that you are going to implement.

This is often done with *pseudocode*. The prefix "pseudo-" means "almost" or "nearly". Pseudocode is almost code. It's close enough to being a real program in a programming language that it's easy to translate, but not so close that you have to worry about the technical details. The natural language (English) algorithm descriptions in Figures 1.1 and 1.2 might be accurate, but they aren't generally written in a way that's easy to transform to a program.

Figure 1.4 is an example of the way we'll write pseudocode in this course. It is a translation of Figure 1.2.

Figure 1.4 uses the notation $\lfloor x \rfloor$, which you might not have seen before. It means "round down", so $\lfloor 3.8 \rfloor = 3$ and $\lfloor -3.8 \rfloor = -4$. It is usually pronounced "*floor*".

```
set $hour$ to 0
set $minute$ to 0
set $second$ to 0
repeat forever:
    set $second$ to $second + 1$
    if $second$ is more than 59, then
        set $second$ to 0
        set $minute$ to $minute + 1$
    if $minute$ is more than 59, then
        set $minute$ to 0
        set $hour$ to $hour + 1$
    if $hour$ is more than 23, then
        set $hour$ to 0
    write "$hour$:$minute$:$second$"
    wait for 1 second
```

Figure 1.4: Another pseudocode example: a digital clock

It is usually helpful to express an algorithm in pseudocode before you start programming. Especially as you're starting to program, just expressing yourself in a programming language is challenging. You don't want to be worrying about what you're trying to say and how to say it at the same time.

Writing good pseudocode will get you to the point that you at least know *what you're trying to do* with your program. Then you can worry about *how to say it* in Python.

There are several computing science courses where no programming language is used and you don't write any code at all. If you know how to program, it is assumed that you know how to convert pseudocode to a program. So, the courses concentrate on pseudocode and algorithms. The rest is easy (once you learn how to program).

There is another example of pseudocode in Figure 1.4. This algorithm could be used to manage the display of a digital clock. It keeps track of the current hour, minute, and second (starting at exactly midnight).

Summary

This unit introduced you to some of the fundamental ideas in computing science. The ideas here are key to all of computing science.

If you're a little fuzzy on what exactly a data structure or pseudocode is, you don't need to panic (yet). After you've written a few programs in later units, come back to these terms and see if they make a little more sense then.

Key Terms

- algorithm
- data structure
- computing science
- computer programming
- programming language
- Python
- pseudocode

Unit 2
Programming Basics

Learning Outcomes

- Use the Python software to get programs running.
- Create programs that perform simple calculations.
- Use variables to store information in a program.
- Create programs that take input from the user.
- Explain how computers store information in binary.
- Take a simple problem and create an algorithm that solves it.
- Implement that algorithm in Python.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapter 2 in *How to Think Like a Computer Scientist.*

Topic 2.1 Starting with Python

In this course, you will be using the *Python* programming language. You can download Python for free or use it in the lab. See Appendix A for more instructions on how to use the software.

One nice feature of Python is its *interactive interpreter*. You can start up Python and start typing in Python code. It will be executed immediately, and you will see the results.

You can also type Python code into a file and save it. Then, you can run it all at once. The interactive interpreter is generally used for exploring the language or testing ideas. Python code in a file can be run as an application and even double-clicked to run your program.

You will start by working with the Python interpreter. See Appendix A for instructions on getting Python running. When you start the Python interpreter, you'll see something like this:

```
Python 3.12.2 (main, Feb 6 2024, 21:26:36)
Type "help", "copyright", "credits" or "license()"
for more information.
>>>
```

The `>>>` is the *prompt*. Whenever you see it in the interpreter, you can type Python commands. When you press return, the command will be executed and you will be shown the result. Whenever you see the `>>>` prompt in examples, it's an example of what you'd see in the interpreter if you typed the code after the `>>>`.

For some reason, when people are taught to program, the first program they see is one that prints the words "Hello world" on the screen. Not wanting to rock the boat, you will do that too. Here's what it looks like in the Python interpreter:

```
>>> print("Hello world")
Hello world
```

The stuff after the prompt is the first line of Python code you have seen. You could have also typed it into a text editor, named the file hello.py and run it.

The `print` function in Python is used to put text on the screen. Whatever is in the parentheses will be printed on the screen.

Any text in quotes, like `"Hello world"` in the example, is called a *string*. Strings are just a bunch of *characters*. Characters are letters, numbers, spaces, and punctuation. Strings have to be placed in quotes to be distinguished from Python commands. If we had left out the quotes, Python would have complained that it didn't know what "`Hello`" meant, since there is no built-in command called `Hello`.

The Interpreter vs. the Editor

When you use Python's *IDLE* (Integrated DeveLopment Environment), the first window that you see is the interactive interpreter. That's the window with the >>> prompt where you can type Python code, press return, and see its result. You can use this to test small sections of Python code to make sure they do what you expect.

If you create a "New Window", the window you create is a file editing window. It doesn't have a prompt or anything else. You can type Python code here and *save* it as a .py file. You can run a Python .py file by double clicking it. You can also press F5 while editing it in IDLE to run it. You should use an editor window to write whole programs.

Check-Up Questions

- ▶ Type `print("Hello world!")` into an editor window in IDLE and save it as hello.py file. Run it with Python.

  If you're using Windows and you run the program by double-clicking the file, the output window might disappear before you can see the results. You can stop this from happening by running the program in IDLE or by waiting for the user to press return before ending the program. We'll talk about how to do that in the next topic.

- ▶ Add a few more `print` statements to your hello.py program (one per line). Run it and see what happens.

Statements

If you did the "Check-Up Questions" above, you would have created a file containing one line:

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

This line is a Python *statement*.

Statements are the basic building blocks of Python programs. Each statement expresses a part of the overall algorithm that you're implementing. The `print` statement is the first one you have seen so far, but there are many others. Each one lets you express a different idea in such a way that the computer can complete it.

When you run a Python program (i.e., code you typed in a `.py` file and saved), the statements are executed in the order they appear in the file. So, the Python program

```
print("Hello world!")
print("I'm a Python program that prints stuff.")
```

. . . will produce this output:

```
Hello world!
I'm a Python program that prints stuff.
```

Topic 2.2 Doing Calculations

In order to implement the algorithm in Figure 1.4, you will need to be able to calculate $guess + 1$ and $\lfloor(smallest + largest)/2\rfloor$.

Python can do calculations like this. An *expression* is any kind of calculation that produces a result. Here are some examples of using expressions in `print` statements in the Python interpreter:

```
>>> print(10 - 2)
8
>>> print(15/3)
5.0
>>> print(25+19*5)
120
>>> print(10.2 / 2 / 2)
2.55
```

The Python *operators* `+`, `-`, `*`, and `/` perform addition, subtraction, multiplication, and division, as you might expect. Note that they do order-of-operations they way you'd expect, too:

```
>>> print(25+19*5)
120
>>> print(25+(19*5))
120
>>> print((25+19)*5)
220
```

Parentheses do the same thing they do when you're writing math: they wrap up part of a calculation so it's done first. Note that a number by itself is an expression too.

```
>>> print(18)
18
```

Now, in Figure 1.4, suppose that the current value of *smallest* is 76 and *largest* is 100. Then, we can at least do the right calculation:

```
>>> print((76+100)/2)
88
```

Python can do calculations on strings too.

```
>>> print("An" + "Expression")
AnExpression
>>> print("An " + 'Expression')
An Expression
>>> print('ABC' * 4)
ABCABCABCABC
```

Note that when you enter a string, it has to be wrapped up in quotes. This is the only way Python can distinguish between characters that are part of a string or part of the expression itself. In Python, single quotes (') and double quotes (") can be used interchangeably.

If you forget the quotes around a string, you'll probably get an error message:

```
>>> print(An + 'Expression')
NameError: name 'An' is not defined
```

Here, Python is telling us that it doesn't know the word "`An`". It does know words like `print` and a few others. If you want to talk about a bunch of characters as part of a string, they have to be surrounded by quotes. So, even when a number, or anything else is in quotes, it is treated like a string (which makes sense, since strings go in quotes):

```
>>> print(120 * 3)
360
>>> print("120" * 3)
120120120
>>> print("120 * 3")
120 * 3
```

Functions

Python can also use *functions* as part of expressions. These work like functions in mathematics: you give the function some *arguments*, and something is done to calculate the result. The result that the function gives back is called its *return value*.

For example, in Python, there is a built-in function `round` that is used to round off a number to the nearest integer:

```
>>> print(round(13.89))
14.0
>>> print(round(-4.3))
-4.0
>>> print(round(1000.5))
1001.0
```

Functions can take more than one argument. The `round` function can take a second argument (an *optional argument*) that indicates the number of decimal places it should round to. For example,

```
>>> print(round(12.3456, 1))
12.3
>>> print(round(12.3456, 2))
12.35
>>> print(round(12.3456, 5))
12.3456
```

In these examples, the value is rounded to the nearest 0.1, 0.01, and 0.00001. For this function, if you don't indicate the optional argument, its default is 0. The *default value* for optional arguments depends on the function.

Functions can take any type of information as their argument and can return any type. For example, Python's `len` function will return the length of a string, i.e. how many characters it has:

```
>>> print(len("hello"))
5
>>> print(len("-<()>-"))
6
>>> print(len(""))
0
```

There are many other ways to do calculations on numbers and strings than we have seen here. You will see more as you learn more about programming. You will see some more functions as you need them.

Actually, `print` is also a function in Python: we're calling it by putting an argument in parentheses after the function name, just like `round` or `len`. It doesn't give back any result ("return" anything), so we aren't using it as part of an expression.

Check-Up Questions

- Try `print`ing the results of some other expressions. Check the calculations by hand and make sure the result is what you expect.
- Try some of the above string expressions, swapping the single quotes for double quotes and vice-versa. Convince yourself that they really do the same thing.
- Some of the examples above "multiply" a string by a number (like `"cow"*3`). The result is repetition of the string. What happens if you multiply a number by a string (`3*"cow"`)? What about a string by a string (`"abc"*"def"`)?

Topic 2.3 Storing Information

You aren't going to want to always print out the result of a calculation like we did in Topic 2.2. Sometimes, you need to perform a calculation to be used later, without needing to display the results right away. You might also want to ask the user a question and remember their answer until you need it.

For example, in the algorithm in Figure 1.2, you want to calculate values for *smallest*, *largest*, and *guess* and store those results. You also need to ask the user for their answer and store the result. You need to keep all of those in the computer's memory.

Whenever we need the computer to temporarily remember some information in a program, we will use a *variable*. A variable is a way for you to reserve a little bit of the computer's memory to *store* the information you need.

You will give variables names that you will use to refer to them later. For example, if you ask the user for their age and want to store their input, you might use a variable named “`age`”. The name of the variable should describe and somehow indicate what it represents.

To put a value in a variable, a *variable assignment statement* is used. For example, to put the result of the calculation `14/2` into a variable named `quotient`,

```
quotient = 14/2
```

In a variable assignment statement, put the name of the variable you want to change on the left, an equals sign, and the new value on the right.

You can use any expression to calculate the value that will be stored in the variable. Variables can store any kind of information that Python can manipulate. So far we have seen numbers and strings.

◈ Be careful: Only the *result* of the calculation is stored, not the whole calculation.

To use the value that's stored in a variable, you just have to use its name. If a variable name is used in an expression, it is replaced with the stored value.

```
>>> num = 7
>>> word = "yes"
>>> print(num - 3)
4
>>> print(word + word)
yesyes
>>> num = 4
>>> print(num - 3)
1
```

Note that you can change the value in a variable. In the above example, `num` was first set to 7 and then changed to 4. Notice that the variable `num` was holding a number and `word` was holding a string. You can change the kind of information a variable holds by doing a variable assignment as well.

Topic 2.4 Types

As noted above and in Topic 2.2, Python treats numbers (like `2`, `-10`, and `3.14`) differently than strings (like `"abc"`, `"-10"`, and `""`). For example, you can divide two numbers, but it doesn't make sense to divide strings.

```
>>> print(10/2)
5.0
>>> print("abc" / 2)
TypeError: unsupported operand type(s) for /: 'str' and
'int'
```

Numbers and strings are two different *types* of information that Python can manipulate. *String* variables are used to hold text or collections of characters.

In Python, a `TypeError` indicates that you've used values whose types can't be used with the given operation. The type of the values given to an operator can change the way it works. In Topic 2.2, you saw that the `+` operator does different things on numbers (addition) and strings (joining).

In fact, the numeric values that Python stores aren't as simple as just "numbers". There are separate types for *integers* and *floating point* values.

Integers are numbers without any fraction part. So, `10`, `0`, and `-100` are all integers. Numbers with fractional parts, like `3.14`, `-0.201`, and `10.0`, are stored as floating point values. These two types are represented differently in the computer's memory, as we will discuss in Topic 2.6.

When dividing integers in Python, the result is always a floating point value, so the result can be represented as closely as possible. There is a separate division operator `//` that is *integer division*: it divides and rounds down, giving the integer below the "true" result.

Sometimes the `/` is called *true division* to distinguish it from `//`. Have a look at some examples in the Python interpreter:

```
>>> print(10 / 2)
5.0
>>> print(10 / 3)
3.3333333333333335
>>> print(11 / 3)
3.6666666666666665
>>> print(10 // 2)
5
>>> print(10 // 3)
3
>>> print(11 // 3)
3
```

◈ You may have noticed that in Python, the calculation `10/3` is close to $\frac{1}{3}$ but not exactly equal. This is a limitation of the way floating point numbers are represented by computers in general (not just in Python). Details will have to wait for another course.

◈ There is a built-in function called `type` that will tell you the type of an object in Python. Try `type(10/3)` and `type(10//3)`.

When implementing the pseudocode in Figure 1.4, you can actually use this to make sure the calculation *guess* rounds down to the next integer.

Note that you can trick Python into treating a whole number like a floating point number by giving it a fractional part with you type it. So `10` is an integer (or "*int*" for short), but `10.0` is a floating point value (or "*float*").

Type Conversion

Sometimes, you'll find you have information of one type, but you need to convert it to another.

For example, suppose you want to calculate the average of several integers. You would do the same thing you would do by hand: add up the numbers and divide by the number of numbers. Suppose you had found the sum of 10 numbers to be 46, leaving the values 46 in `sum` and 10 in `num`. If you try to divide these numbers in Python, you'll get the result 4, since you're dividing two integers. Really, you want the result 4.6, which you would get if at least one of the values being divided was a float.

There are Python functions that can be used to change a value from one type to another. You can use these in an expression to get the type you want. The function `int()` converts to an integer, `float()` converts to a floating point value, and `str()` converts to a string. For example,

```
>>> float(10)
10.0
>>> str(10)
'10'
>>> int('10')
10
>>> int(83.7)
83
>>> str(123.321)
```

```
'123.321'
>>> int("uhoh")
ValueError: invalid literal for int(): uhoh
```

As you can see, these functions will do their best to convert whatever you give them to the appropriate type. Sometimes, that's just not possible: there's no way to turn `"uhoh"` into an integer, so it causes an error.

Converting numbers to strings is often handy when printing. Again, suppose you have the integer 46 in the variable `total` and you want to print out a line like

```
The sum was 46.
```

You can print out multiple values with the comma, but they are separated by spaces:

```
>>> print("The sum was", total, ".")
The sum was 46 .
```

Note that there's a space between the `46` and the period. You can remove this by combining strings to get the result we want:

```
>>> print("The sum was " + str(total) + ".")
The sum was 46.
```

When Python joins strings, it doesn't add any extra spaces. You have to convert `total` to a string here since Python doesn't know how to add a string and a number:

```
>>> print("The sum was " + total + ".")
TypeError: can only concatenate str (not "int") to str
```

The word *concatenate* means "join together". When you use the `+` on strings, it's not really adding them, it's joining them. That's called concatenation.

Topic 2.5 User Input

Something else you will need to do to implement the algorithm from Figure 1.4 is to get input from the user. You need to ask them if the number they're thinking of is larger, smaller or equal.

To do this in Python, use the `input` function. This function will give the user whatever message you tell it to, wait for them to type a response and press enter, and return their response to your expression.

For example, this program will ask the user for their name and then say hello:

```
name = input("What is your name? ")
print("Hello, " + name + ".")
```

If you run this program, it will display "`What is your name? `" on the screen and wait for the user to respond. Their response will be stored in the variable `name`. For example,

```
What is your name? Julius
Hello, Julius.
```

If the user enters something else, that's what will go in the `name` variable,

```
What is your name? Joey Jo-Jo
Hello, Joey Jo-Jo.
```

◈ In this guide, any input that the user types will be **set in bold, like this**.

Whenever you use the `input` function, it will return a string. That's because as far as the interpreter is concerned, the user just typed a bunch of characters and that's exactly what a string is.

If you want to treat the user's input as an integer or floating point number, you have to use one of the type conversion functions described above. For example, if you ask the user for their height, you really want a floating point value, but we get a string. So, it must be converted:

```
m = float(input("Enter your height (in metres): "))
inches = 39.37 * m
print("You are " + str(inches) + " inches tall.")
```

When you run this program,

```
Enter your height (in metres): 1.8
You are 70.866 inches tall.
```

In this example, the user enters the string `"1.8"`, which is returned by the `input` function. That is converted to the floating point number `1.8` by the `float` function. This is stored in the variable `m` (for "metres"). Once there is a floating point value in `m`, your program can do numeric calculations with it. The number of inches is calculated and the corresponding floating point number is stored in `inches`. To print this out, it is converted back to a string with the `str` function. Sometimes `print` will do the conversion for you, but it was done explicitly in this program.

Topic 2.6 How Computers Represent Information

You may be wondering why you have to care about all of the different types of values that Python can handle. Why should `25` be different from `25.0`? For that matter, how is the number `25` different from the string `"25"`?

The real difference here is in the way the computer stores these different kinds of information. To understand that, you need to know a little about how computers store information.

Binary

All information that is stored and manipulated with a computer is represented in *binary*, i.e. with zeros and ones. So, no matter what kind of information you work with, it has to be turned into a string of zeros and ones if you want to manipulate it with a computer.

Why just zeros and ones?

A computer's memory is basically a whole bunch of tiny rechargeable batteries (*capacitors*). These can either be discharged (0) or charged (1). It's fairly easy for the computer to look at one of these capacitors and decide if it's charged or not.

> It's possible to use the same technology to represent digits from 0 to 9, but it's very difficult to distinguish ten different levels of charge in a capacitor. It's also very hard to make sure a capacitor doesn't discharge a little to drop from a 7 to a 6 without noticing. So, modern computers don't do this. They just use a simpler system with two levels of charge and end up with zeros and ones.

Hard disks and other storage devices also use binary for similar reasons. Computer networks do as well.

Prefix	**Symbol**	**Factor**
(no prefix)		$2^0 = 1$
kilo-	k	$2^{10} = 1024 \approx 10^3$
mega-	M	$2^{20} = 1048576 \approx 10^6$
giga-	G	$2^{30} = 1073741824 \approx 10^9$
tera-	T	$2^{40} = 1099511627776 \approx 10^{12}$

Figure 2.1: Prefixes for storage units.

A single piece of storage that can store a zero or one is called a *bit*. Since a bit is a very small piece of information to worry about, bits are often grouped. It's common to divide a computer's memory into eight-bit groups called *bytes*. So, 00100111 and 11110110 are examples of bytes.

When measuring storage capacity, the number of bits or bytes quickly becomes large. Figure 2.1 show the prefixes that are used for storage units and what they mean.

For example, "12 megabytes" is

$$12 \times 2^{20} \text{ bytes} = 12582912 \text{ bytes} = 12582912 \times 8 \text{ bits} = 100663296 \text{ bits}\,.$$

Note that the values in Figure 2.1 are slightly different than the usual meaning of the metric prefixes. One kilometre is exactly 1000 metres, not 1024 metres. When measuring storage capacities in computers, the 1024 version of the metric prefixes is usually used.

That statement isn't entirely true. Hard drive makers, for instance, generally use units of 1000 because people would generally prefer a "60 gigabyte" drive to a "55.88 gigabyte" drive ($60 \times 10^{12} = 55.88 \times 2^{30}$).

Unsigned Integers

Once you have a bunch of bits, you can use them to represent numbers.

First, think about the way you count with regular numbers: 1, 2, 3, 4, 5. . . . Consider the number 157. What does each of the digits in that number mean? The "1" is one hundred, "5" is five tens, and "7" is seven ones: $157 = (1 \times 10^2) + (5 \times 10) + (7 \times 1)$.

As you go left from one place to the next, the value it represents is multiplied by 10. Each digit represents the number of 1s, 10s, 100s, 1000s. . . .

The reason the values increase by a factor of 10 is that there are ten possible digits in each place: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This is called *decimal* or *base 10 arithmetic*. (The "dec-" prefix in latin means 10.)

Applying the same logic, there is a counting system with bits, *binary* or *base 2 arithmetic* ("bi-" means 2). The rightmost bit will be the number of 1s, the next will be the number of 2s, then 4s, 8s, 16s, and so on. Binary values are often written with a little 2 (a subscript), to indicate that they are base 2 values: 101_2. If there's any possibility for confusion, base 10 values are written with a subscript 10: 34_{10}.

To convert binary values to decimal, do the same thing you did above, substituting 2s for the 10s:

$$\begin{aligned}1001_2 &= (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 8 + 1 \\ &= 9_{10} \,.\end{aligned}$$

The base 2 value 1001_2 is equal to 9_{10}. Another example with a larger number:

$$\begin{aligned}10011101_2 = &\ (1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + \\ &\ (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ = &\ 128 + 16 + 8 + 4 + 1 \\ = &\ 157_{10} \,.\end{aligned}$$

So, 10011101 is the base 2 representation of the number 157. Any positive whole number can be represented this way, given enough bits. All of the values that can be represented with four bits are listed in Figure 2.2.

You should be able to convince yourself that for any group of n bits, there are 2^n different possible values that can be stored in those bits. So, n bits can represent any number from 0 to $2^n - 1$. Other common groupings are of 16 bits (which can represent numbers 0 to $2^{16} - 1 = 65535$) and 32 bits (which can represent numbers 0 to $2^{32} - 1 = 4294967295$).

The computer can do operations like addition and subtraction on binary integers the same way you do with decimal numbers. You just have to keep in mind that $1 + 1 = 2_{10} = 10_2$, so if you add two 1's together, there is a carry.

There are a few examples of binary addition in Figure 2.3. These correspond to the decimal operations $10 + 4 = 14$, $11 + 2 = 13$, and $13 + 5 = 18$.

binary	**decimal**	**binary**	**decimal**
1111	15	0111	7
1110	14	0110	6
1101	13	0101	5
1100	12	0100	4
1011	11	0011	3
1010	10	0010	2
1001	9	0001	1
1000	8	0000	0

Figure 2.2: The four-bit unsigned integer values.

$$\begin{array}{cccccc} & & & & & \\ & & 1 & 0 & 1 & 0 \\ + & & 0 & 1 & 0 & 0 \\ \hline & & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{cccccc} & & & {\scriptstyle 1} & & \\ & & 1 & 0 & 1 & 1 \\ + & & 0 & 0 & 1 & 0 \\ \hline & & 1 & 1 & 0 & 1 \end{array} \qquad \begin{array}{cccccc} & & {\scriptstyle 1} & & {\scriptstyle 1} & \\ & & 1 & 1 & 0 & 1 \\ + & & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 \end{array}$$

Figure 2.3: Some examples of binary addition

You can use the familiar algorithms you know for subtraction, multiplication, and division as well.

Positive and Negative Integers

The method described above will let us represent any *positive* integer in the computer's memory. What about negative numbers?

The bits that make up the computer's memory must be used to represent both positive and negative numbers. The typical method is called *two's complement notation.* (The previous method, which can't represent negative values, is generally called *unsigned integer* representation.)

To convert a positive value to a negative value in two's complement, you first flip all of the bits (convert 0s to 1s and 1s to 0s) and then add one. So, the four-bit two's complement representation for -5 is:

start with the positive version:	0101
flip all of the bits:	1010
add one:	1011 .

binary	**decimal**	**binary**	**decimal**
1111	−1	0111	7
1110	−2	0110	6
1101	−3	0101	5
1100	−4	0100	4
1011	−5	0011	3
1010	−6	0010	2
1001	−7	0001	1
1000	−8	0000	0

Figure 2.4: The four-bit two's complement values

All of the four-bit two's complement values are shown in Figure 2.4. If we use four bits, we can represent values from −8 to 7.

Here are a few other reasons computers use two's complement notation:

- It's easy to tell if the value is negative: if the first bit is 1, it's negative.
- For positive numbers (values with the first bit 0), the unsigned and two's complement representations are identical. The values 0–7 have the same representations in Figures 2.2 and 2.4.
- Addition and subtraction work the same way as for unsigned numbers. Look back at Figure 2.3. If you instead interpret at the numbers as two's complement values, the corresponding decimal calculations are $-6 + 4 = -2$, $-5 + 2 = -3$, and $-3 + 5 = 2$. (You have to ignore the last 1 that was carried in the last example—the computer will.) They are still correct. That means that the parts of the computer that do calculations don't have to know whether they have unsigned or two's complement values to work with.
- No number has more than one two's complement representation. If instead the first bit was used for the sign (0 = positive, 1 = negative), then there would be two versions of zero: 0000 and 1000. This is a waste of one representation, which wastes storage space, not to mention that the computer has to deal with the special case that 0000 and 1000 are actually the same value. That makes it difficult to compare two values.

Most modern computers and programming languages use 32 bits to store integers. With this many bits, it is possible to store integers from -2^{31} to $2^{31} - 1$ or −2147483648 to 2147483647.

So, in many programming languages, you will get an error if you try to add one to 2147483647. In other languages, you will get -2147483648. The analogous calculation with four bits is $7 + 1$:

$$\begin{array}{r} {\scriptstyle 1\,1\,1} \\ 0\,1\,1\,1 \\ +\;0\,0\,0\,1 \\ \hline 1\,0\,0\,0 \end{array}$$

If these were unsigned values, this is the right answer. But, if you look in Figure 2.4, you'll see that 1000 represents -8. If this *overflow* isn't caught when doing two's complement, there's a "wraparound" that means you can suddenly go from a large positive number to a large negative one, or vice-versa.

In Python, you don't generally see any of this. Python will automatically adjust how it represents the numbers internally and can represent any integer. But, if you go on to other languages, you will eventually run into an integer overflow.

Another type of numbers is the floating point value. They have to be stored differently because there's no way to store fractional parts with two's complement. Floating point representation is more complicated; it is beyond the scope of this course.

Characters and Strings

The other types of information that you have seen in your Python experience are characters and strings. A *character* is a single letter, digit or punctuation symbol. A *string* is a collection of several characters. So, some characters are `T`, `$`, and `4`. Some strings are `"Jasper"`, `"742"`, and `"bhay-gn-flay-vn"`.

Storing characters is as easy as storing unsigned integers. For a byte (8 bits) in the computer's memory, there are $2^8 = 256$ different unsigned numbers: 0–255. So, just assign each possible character a number and translate the numbers to characters.

For example, the character `T` is represented by the number 84, the character `$` by 36, and `4` by 52. This set of translations from numbers to characters and back again is called a *character set*. The particular character set that is used by almost all modern computers, when dealing with English and other western languages, is called *ASCII*. The course web site contains links to a full list of ASCII characters, if you'd like to see it.

$$\begin{tikzcd}
\text{H} \arrow[d] & \text{i} \arrow[d, "\text{(ASCII chart lookup)}"] \\
72 \arrow[d] & 105 \arrow[d, "\text{(conversion to binary)}"] \\
01001000 & 01101001
\end{tikzcd}$$

Figure 2.5: Conversion of the string “Hi” to binary.

So, in order to store the character `T` in the computer’s memory, first look up its number in the character set and get 84. Then, use the method described for unsigned integers to convert the number 84 to an 8-bit value: 01010100. This can then be stored in the computer’s memory.

With only one byte per character, we can only store 256 different characters in our strings. This is enough to represent English text, but it starts to get pretty hard to represent languages with accents (like á or ü). It’s just not enough characters to represent languages like Chinese or Japanese.

The *Unicode* character set was created to overcome this limitation. Unicode can represent up to 2^{32} characters. This is enough to represent all of the written languages that people use. Because of the number of possible characters, Unicode requires more than one byte to store each character.

In ASCII, storing strings with several characters, can be done by using a sequence of several bytes and storing one character in each one. For example, in Figure 2.5, the string “Hi” is converted to binary.

In Figure 2.5, the binary string 0100100001101001 represents “Hi” in ASCII. But, if you look at this chunk of binary as representing an integer, it’s the same as 18537. How does the computer know whether these two bytes in memory are representing the string “Hi” or the number 18537?

There actually isn’t any difference as far as the computer itself is concerned. Its only job is to store the bits its given and do whatever calculations it’s asked to do. The programming language must keep track of what kind of information the different parts of the memory are holding. This is why the concept of types is so important in Python. If Python didn’t keep track of the type of each variable, there would be no way to tell.

In some programming languages, C in particular, you can work around the type information that the programming language is storing. For example, you could store the string “Hi” and then later convince the computer that you wanted to treat that piece of memory like a number and get 18537. This is almost always a bad idea.

How computers represent various types of information is sometimes quite important when programming. It is also discussed in CMPT 295 (Intro to Computer Systems) and courses that cover how programming languages work like CMPT 379 (Compiler Design).

Topic 2.7 Example Problem Solving: Feet and Inches

Back in Topic 2.5, there was a program that converted someone’s height in metres to inches:

```
Enter your height (in metres): 1.6
You are 62.992 inches tall.
```

But, people don’t usually think of their height in terms of the number of inches. It’s much more common to think of feet and inches. It would be better if the program worked like this:

```
Enter your height (in metres): 1.6
You are 5’ 3" tall.
```

The notation $5'\,3''$ is used to indicate “5 feet and 3 inches”, which is $5 \times 12 + 3 = 63$ inches.

To do this conversion, convert the number of metres to inches, as done in Topic 2.5, by multiplying by 39.37. Then, determine how many feet and inches there are in the total number of inches. The pseudocode is shown in Figure 2.6.

When you’re converting an idea for an algorithm to code, you shouldn’t try to do it all at once, especially when you’re first learning to program. Implement part of the algorithm first, then test the program to make sure it does what you expect before you move on. Trying to find problems in a large chunk of code is very hard: start small.

```
**write** "Enter your height (in metres):"
**read** $metres$
**set** $totalinches$ to $39.37 \times metres$
**set** $feet$ to $\lfloor totalinches/12 \rfloor$
**set** $inches$ to $totalinches - feet \times 12$
round $inches$ to the nearest integer
**write** "You are $feet'$ $inches''$ tall."
```

Figure 2.6: Meters to feet-inches conversion pseudocode.

```
metres = float(input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = total_inches/12
print("You are " + str(feet) + " feet tall.")
```

Figure 2.7: Converting to feet and inches: number of feet.

Start writing a Python program to implement the pseudocode in Figure 2.6. You can grab the first few lines from the program in Topic 2.5. Then, try to calculate the number of feet. This has been done in Figure 2.7.

Note that when you run this program, it calculates the number of feet as a floating point number:

```
Enter your height (in metres): 1.6
You are 5.24933333333 feet tall.
```

This makes sense, given what we know about types: when Python divides a floating point value (`metres`), it returns a floating point value. But in the algorithm, you need an integer and it needs to be rounded *down* to the next integer. This is what the `int` function does when it converts floating point numbers to integers, so you can use that to get the correct value in `feet`.

◈ If you have a statement in Python that you want to split across multiple lines, so it's easier to read, you can end the line with a backslash, "\". This was done in Figures 2.7 and 2.8, so the code would fit on the page.

Once you have the correct number of feet as an integer, you can calculate the number of inches too. This is done in Figure 2.8.

```
metres = float(input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = int(total_inches/12)
inches = total_inches - feet*12
print("You are " + str(feet) + " feet and " \
        + str(inches) + " inches tall.")
```

Figure 2.8: Converting to feet and inches: feet and inches.

```
metres = float(input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = int(total_inches/12)
inches = int(round(total_inches - feet*12))
print("You are " + str(feet) + " feet and " \
        + str(inches) + " inches tall.")
```

Figure 2.9: Converting to feet and inches: rounding inches.

This program does the right calculation, but leaves the number of inches as a floating point number:

```
Enter your height (in metres): 1.6
You are 5 feet and 2.992 inches tall.
```

To convert the number of inches to an integer, you can't use the `int` function, which would always round down. You shouldn't get $5' \ 2''$ in the above example; you should round to the *nearest* integer and get $5' \ 3''$.

You can use the `round` function for this. Note that `round` does the rounding, but leaves the result as a floating point value. You will have to use the `int` function to change the type, but the value will already be correct.

See Figure 2.9 for the details. When you run this program, the output is almost correct:

```
Enter your height (in metres): 1.6
You are 5 feet and 3 inches tall.
```

The last thing you have to do to get the program working exactly as specified at the start of the topic is to print out the feet and inches in the proper format: $5' \, 3''$. This presents one last problem. You can't just print double quotes, since they are used to indicate where the string literal begins and ends. Code like this will generate an error:

```
print(str(feet) + "' " + str(inches) + "" tall")
```

The interpreter will see the `""` and think it's an empty string (a string with no characters in it). Then, it will be very confused by the word "`tall`". The solution is to somehow indicate that the quote is something that it should print, not something that's ending the string. There are several ways to do this in Python:

- Put a backslash before the quote. This is called *escaping a character*. It's used in a lot of languages to indicate that you mean the character itself, not its special use.

  ```
  print(str(feet) + "' " + str(inches) + "\" tall")
  ```

- Use a single quote to wrap up the string. In Python, you can use either single quotes (`'`) or double quotes (`"`) to indicate a string literal. There's no confusion if you have a double quote inside a single-quoted string.

  ```
  print(str(feet) + "' " + str(inches) + '" tall')
  ```

  Of course, you have to use double quotes for the string that contains a single quote.

- A final trick that can be used is Python's *triple-quoted string*. If you wrap a string in *three* sets of double quotes, you can put anything inside (even line breaks). This can be a handy trick if you have a lot of stuff to print and don't want to have to worry about escaping characters.

  ```
  print(str(feet) + """' """ + str(inches) \
          + """" tall""")
  ```

  This can be very cumbersome and hard to read for short strings like this. (As you can see, it made the whole thing long enough it wouldn't fit on one line.) It's more useful for long strings.

So, finally, the quotes can be printed to produce the desired output. See Figure 2.10. When the program runs, it produces output like this:

```
metres = float(input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = int(total_inches/12)
inches = int(round(total_inches - feet*12))
print("You are " + str(feet) + "' " \
        + str(inches) + '" tall.')
```

Figure 2.10: Converting to feet and inches: printing quotes

```
metres = float(input( \
        "Enter your height (in metres): "))
total_inches = int(round(39.37 * metres))
feet = total_inches/12
inches = total_inches - feet*12
print("You are " + str(feet) + "' " \
        + str(inches) + '" tall.')
```

Figure 2.11: Converting to feet and inches: fixed rounding error

```
Enter your height (in metres): 1.6
You are 5' 3" tall.
```

But, there is still one problem with this program that is a little hard to notice. What happens when somebody comes along who is 182 cm tall?

```
Enter your height (in metres): 1.82
You are 5' 12" tall.
```

That's not right: five feet and twelve inches should be displayed as six feet and zero inches. The problem is with the rounding-off in the calculation. For this input, `total_inches` becomes 71.6534, which is *just under* six feet (72 inches). Then the division to calculate `feet` gives a result of 5, which we should think of as an error.

The problem isn't hard to fix: we are just doing the rounding-off too late. If instead of `total_inches` being the floating-point value 71.6534, we could round it off immediately to 72. That would correct this problem and it has been done in Figure 2.11.

Now we get the right output:

```
Enter your height (in metres): 1.82
You are 6' 0" tall.
```

This is a good lesson for you to see at this point: it's important to test your program carefully, since bugs can hide in unexpected places.

Check-Up Questions

▶ Download the code from this topic from the course web site and test it with some other inputs. Do the conversion by hand and make sure the program is working correctly.

▶ Try some "bad" inputs and see what the program does. For example, what if the user types in a negative height? What if they type something that isn't a number?

Summary

There's a lot in this unit. You should be writing your first programs and figuring out how computers work. The example developed in Topic 2.7 is intended to give you some idea of how the process of creating a program might look.

When you're learning to program, you should be writing programs. Reading this Guide over and over won't help. You should actually spend some time at a computer, experimenting with the ideas presented here, learning how to decipher error messages, and dealing with all of the other problems that come with writing your first programs.

Key Terms

- interactive interpreter
- statement
- expression
- operator
- function
- argument
- variable
- variable assignment
- type
- conversion
- integer
- unsigned integer

- string
- ASCII
- floating point
- binary
- bit
- byte
- two's complement
- character set
- escaping a character

Unit 3

Control Structures

Learning Outcomes

- Design algorithms that use decision making and implement them in Python.
- Design algorithms that use iteration and implement them in Python.
- Given an algorithm, approximate its running time.
- Find and fix bugs in small programs.
- Create algorithms for more complex problems.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Sections 4.2–4.7, 6.2–6.4, 1.3, Appendix A in *How to Think Like a Computer Scientist*.

Topic 3.1 Making Decisions

All of the code we have written so far has been pretty simple. It all runs from top to bottom, and every line executes once as it goes by. The process soon becomes boring. It's also not very useful.

```
write "Think of a number between 1 and 10."
set guess to 6.
write "Is your number equal to guess?"
read answer
if answer is "yes", then
    write "I got it right!"
if answer isn't "yes", then
    write "Nuts."
write "That's the end of the game."
```

Figure 3.1: Simplified guessing game

In the guessing game example from Figure 1.4, we need to decide if the user has guessed correctly or not and then take the appropriate action. Almost every program you write to solve a real problem is going to need to do this kind of thing.

There are a few ways to make decisions in Python. We'll only explore one of them here.

The `if` statement

The most common way to make decisions in Python is by using the `if` statement. The `if` statement lets you ask if some condition is true. If it is, the *body* of the `if` will be executed.

For example, let's simplify the guessing game example from Figure 1.4. In the simplified version, the user thinks of a number from 1 to 10 and the computer only takes one guess. Pseudocode for this game is shown in Figure 3.1.

Some Python code that implements Figure 3.1 can be found in Figure 3.2.

Here are two example executions of the program:

```
Think of a number between 1 and 10.
Is your number equal to 6? no
Nuts.
That's the end of the game.

Think of a number between 1 and 10.
Is your number equal to 6? yes
```

```
print("Think of a number between 1 and 10.")
guess = 6
answer = input("Is your number equal to " \
        + str(guess) + "? ")
if answer == "yes":
    print("I got it right!")
if answer != "yes":
    print("Nuts.")
print("That's the end of the game.")
```

Figure 3.2: Implementation of Figure 3.1

```
I got it right!
That's the end of the game.
```

As you can see from these examples, only one of the `print` statements inside of the `if` is executed. The `if` statement is used to make a decision about whether or not some code should be executed.

The *condition* is used to decide what to do. The two conditions in Figure 3.2 are `answer == "yes"` and `answer != "yes"`. These mean "`answer` is equal to `yes`" and "`answer` is not equal to `yes`," respectively. We will look more at how to construct conditions later.

The indented `print` statements are not executed when the `if` condition is false. These statements make up the *body* of each `if` statement. The last `print` is executed no matter what: it isn't part of the `if`.

In Python (unlike many programming languages), the amount of space you use is important. The only way you can indicate what statements are part of the `if` body is by indenting, which means you'll have to be careful about spacing in your program.

All block statements in Python (we'll be seeing more later) are indented the same way. You start the block and then everything that's indented after it is the body of the block. When you stop indenting, the block is over.

How much you indent is up to you, but you have to be consistent. Most Python programmers indent 4 spaces and all of the example code for this course is written that way.

Boolean Expressions

The expressions that are used for `if` conditions must be either true or false. In Figure 3.2, the first condition is `answer == "yes"`, and it evaluates to true when the value stored in `answer` is `"yes"`.

These conditions are called *boolean expressions*. The two *boolean values* are `True` and `False`. A boolean expression is any expression that evaluates to true or false.

To check to see if two values are equal, the `==` operator is used and `!=` is the not equal operator.

```
>>> if 4-1==3:
        print "Yes"

Yes
```

◈ You have to press an extra Enter in this example after the `print` statement. In the Python interpreter, the extra blank line is used to tell it you're done the block.

The less than sign (`<`) and greater than sign (`>`) do just what you'd expect. For `<`, if the left operand is less than the right operand, it returns true. There are also boolean operators less than or equal (`<=`), and greater than or equal (`>=`).

Note the difference between `=` and `==`. The `=` is used for variable assignment; you're telling Python to put a value into a variable. The `==` is used for comparison—you're asking Python a question about the two operands. Python won't let you accidentally use a `=` as part of a boolean expression, for this reason.

Functions and methods can also return `True` or `False`. For example, strings have a method `islower` that returns `True` if all of the characters in the string are lower case (or not letters):

```
>>> s = "Hans"
>>> s.islower()
False
>>> s = "hans"
>>> s.islower()
True
```

The else clause

In Figure 3.2, we wanted to take one action if the user answered "yes" and another if they answered anything else. It could also have been written in the following way:

```
if answer == "yes":
    print("I got it right!")
else:
    print("Nuts.")
```

The else clause is executed if the if condition is *not* true.

In the if statement, you can specify an else clause. The purpose of the else is to give an "if not" block of code. The else code is executed if the condition in the if is false.

It is also possible to allow more possibilities with elif blocks. The elif is used as an "else if" option. In Figure 3.2, we could have done something like this:

```
if answer == "yes":
    print("I got it right!")
elif answer == "no":
    print("Nuts.")
else:
    print("You must answer 'yes' or 'no'.")
```

Here, the logic of the program changes a little. They have to answer "yes" or "no". If they answer anything else, they get an error message.

Any number of elifs can be inserted to allow for many possibilities. Whenever an if...elif...elif...else structure is used, only one of the code bodies will be executed. The else will only execute if *none* of the conditions are true.

Topic 3.2 Definite Iteration: for loops

We are still missing one major concept in computer programming. We need to be able to execute the same code several times (iterate). There are several ways to iterate in most programming languages. We will discuss two ways you can use in Python: for and while loops.

```
write "Enter a nonnegative integer:"
read $n$
set $factorial$ to 1
do this for $i$ equal to each number from 1 to $n$:
    set $factorial$ to $factorial \times i$
write $factorial$
```

Figure 3.3: Pseudocode to calculate factorials

```
num = int( input("How high should I count? ") )
for i in range(num):
    print(i, end=' ')
```

Figure 3.4: Using a `for` loop in Python

The `for` loop

In some problems, you know ahead of time how many times you want to execute some code. For example, suppose we want to calculate *factorials*. If you haven't run across factorials before, "n factorial" is written "$n!$" and is the product of all of the number from 1 to n:

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n\,.$$

We can write a program to calculate factorials and we'll know that we have to do n multiplications. Figure 3.3 contains pseudocode to calculate factorials.

In Python, if you have a problem like this where you know before you start iterating how many times you'll have to loop, you can use a `for` loop.

Loops where you know how many times you're going to loop when you start are called *definite loops*. Well, they are in textbooks. Everybody else just calls them "`for` loops". Isn't Computing Science fun?

Before we try to implement a factorial program from Figure 3.3, let's explore the `for` loop a little. The program in Figure 3.4 uses a `for` loop to count as high as the user asks.

The easiest way to construct a `for` loop is with the `range` function. When a `for` loop is given `range(x)`, the loop body will execute `x` times. Figure 3.4 will look like this when it's executed:

```
n = int( input("Enter a nonnegative integer: ") )
factorial = 1

for i in range(n):
    factorial = factorial * (i+1)

print(factorial)
```

Figure 3.5: Calculating factorials with Python

```
How high should I count? 12
0 1 2 3 4 5 6 7 8 9 10 11
```

Notice that the `range` starts from zero and counts up to `num` − 1. If we wanted to count from one, we could have written the loop like this:

```
for i in range(num):
    print(i+1, end=' ')
```

We will have to do this when implementing the factorial algorithm since we need the values from 1 to n, not from 0 to $n-1$.

Here are the details of what happens when that loop runs: The given range, `range(num)`, will cause the loop to repeat `num` times; the range represents the integers `0`, `1`, `2`, . . . , `num-1`. The loop variable (`i`) will be set to the first value in the range (`0`), and the loop body (the `print` statement) is executed. The body is then repeated for each of the other values in the range (`1`, `2`, . . . , `num-1`).

Figure 3.5 contains a program program that calculates $n!$.

For some strange historical reason, `i` is a common choice for the `for` loop variable if nothing else is appropriate. You should choose a descriptive variable name where possible, but for quick, short loops, `i` is a good default.

Topic 3.3 Indefinite Iteration: `while` loops

If you don't know how many times you want the loop body to execute, the `for` loop is hard to work with. For example, in the guessing game in Figure 1.4, we just have to keep guessing until we get it right. That could be

```
name = input("What is your name? ")

while name == "":
    name = input("Please enter your name: ")

print("Hello, " + name)
```

Figure 3.6: Using a `while` loop in Python

anywhere from 1 to 7 guesses. We will finally implement the guessing game in Topic 3.5.

In Python, you can do this with a `while` loop. To construct a `while` loop, you use a condition as you did in a `if` statement. The body of the loop will execute as many times as necessary until the condition becomes false.

For example, the program in Figure 3.6 will ask the user to enter his or her name. If a user just presses enter, the program will keep asking until the user provides a response. It looks like this when it's executed:

```
What is your name?
Please enter your name:
Please enter your name:
Please enter your name: Sherri
Hello, Sherri
```

When the `while` loop runs, these steps are repeated:

1. Check the value of the loop condition. If it's `False`, the loop exits, and the program moves on to the following code.
2. Run the loop body.

Basically, the `while` loop uses its condition to ask "should I keep going?" If so, it runs the loop once more and asks again.

When you use an indefinite loop, you have to make sure that the loop condition eventually becomes false. If not, your program will just sit there looping forever. This is called an *infinite loop*.

You'll write an infinite loop sooner or later. Press control-C to stop the Python interpreter when you get tired of waiting for infinity.

Topic 3.4 Choosing Control Structures

When you're starting to program, you may find it difficult to decide which control structure(s) to use to get a particular result. This is something that takes practice and experience programming. Once you figure out how to *really* work with these control structures, it becomes easier.

There aren't any rigid rules here. There are often many ways to do the same thing in a program, especially as things get more complicated. Below are a few guidelines.

Before you can choose a control structure, you need to have a pretty good idea what you want the computer to do: you need to have an algorithm in mind. Once you do, you can then think about how to get a program to do what you want.

- Just do it. Remember that most statements in Python (and most other programming languages) are executed in the order that they appear. So, a variable assignment, or `print` statement will execute right after the statement before (unless a control structure changes how things run).

  These statements tell the computer *what* you want to do. The control structures let you express *when* it will happen. (Sort of. Don't take that too literally.)

- Maybe do it. If you have some code that you want to execute only in a particular situation, then a conditional (`if` statement) is appropriate. An `if` will run its body *zero times or one time*. If you need to do similar things more than once, you should be looking at a loop.

  If the logic you need is "either this or that," then you should use `if` with an `else` clause. If you need to do "one of these things," then use the `if`…`elif`…`elif`…`else` form.

  In any of these cases, you need to come up with a boolean expression that describes when it's appropriate to do each case. This again takes practice. The goal is to use the variables you have and the boolean operators to come up with something that is true *exactly* when you want the code to run.

- Do it several times. When you need to do something several times, you need a loop. Remember that the loops can execute zero, one, or more times, depending on exactly how you've expressed things (and the values of variables, and what the user types, and so on).

Note that the task done in the body of the loop doesn't have to be *exactly* the same every time through. You can use the loop variable (in a `for` loop) and any other variables in your program to keep track of what should be done *this* time through the loop. For example, you might want to examine the loop variable to see if it has a particular property and print it to the screen if it does. To do this, you would use a `if` statement in the loop, and write a condition that expresses the property you're looking for.

- Do it this many times. If you know when you start the loop how many times it will run (for example, count up to a certain number, or run once for every item in a collection), you can use a `for` loop.

  For now, all of our `for` loops will loop over a `range` of numbers. In Topic 5.2, you will see that `for` can be used to loop over other items.

  Note that you don't need to know how many times you'll loop when you're *writing* the program. The size of the `range` can be an expression that's calculated from user input, or anything else. You just need to be able to figure this out when the program gets to the `for` loop.

  If you do a calculation and end up looping over `range(0)`, the body of the `for` loop will run zero times.

- Do it until it's done. There are often situations when you can't tell how many times to loop until you notice that you're done. These are usually of the form "keep doing this until you're done; you're done when *this* happens."

  In these cases, you probably need a `while` loop. A `while` loop is similar to an `if` in that you need to write a boolean expression that describes when to "go". Unlike a `if`, a `while` loop will execute the body repeatedly, until its condition is false.

  When writing the condition for a `while` loop, you need to figure out how the computer will determine that you still have more work to do before the loop is finished. Often, this is "I need to take another step if…". The body of the `while` is then the code necessary to do a "step".

Once again, finding the control structure to express what you want to do

```
print("Think of a number from 1 to 100.")
smallest = 1
largest = 100
guess = (smallest + largest) // 2
print("My first guess is " + str(guess) + ".")
```

Figure 3.7: Guessing game: first guess

does take some practice, and there aren't really any rules. But, hopefully the above will give you something to start with.

Topic 3.5 Example Problem Solving: Guessing Game

In Unit 1, we introduced the guessing game algorithm. It guessed a number from 1 to 100 that the user was thinking. Working from the pseudocode in Figure 1.4, we now have all of the tools we need to write a program implementing this algorithm.

As we saw in the first problem solving example in Topic 2.7, you shouldn't try to write the whole program at once and just hope it will work. You should test as you write.

The program in Figure 3.7 starts the game and makes the first guess. This will let us test the expression to calculate `guess` first. We can change the initial values for `smallest` and `largest` and make sure `guess` is always halfway between. We are using `//` for division because we want the result to be an integer: halfway between 1 and 100 should be 50, not 50.5.

Now that we can make one guess, we can combine this with an `if` statement to ask the user whether or not we're right. This is similar to Figure 3.2. This is done in Figure 3.8.

We can check this program and make sure our logic is right. Have we accidentally interchanged what should be done in the two cases? Suppose we're thinking of 80:

```
Think of a number from 1 to 100.
Is your number 'more', 'less', or 'equal' to 50? more
51 100
```

```
print("Think of a number from 1 to 100.")
smallest = 1
largest = 100
guess = (smallest + largest) // 2
answer = input( "Is your number 'more', 'less'," \
        " or 'equal' to " + str(guess) + "? " )

if answer == "more":
    smallest = guess + 1
elif answer == "less":
    largest = guess - 1

print(smallest, largest)
```

Figure 3.8: Guessing game: get an answer

Now, the program will be guessing numbers from 51 to 100, which is the right range. On the other side, if we were thinking of 43,

```
Think of a number from 1 to 100.
Is your number 'more', 'less', or 'equal' to 50? less
1 49
```

Since 43 is in the range 1–49, we are still on the right track.

Now, we can put this into a loop and keep guessing until we get it right. By directly translating out pseudocode, we would get something like Figure 3.9.

But, if you try to run this program, you'll see an error like this:

```
Think of a number from 1 to 100.
Traceback (most recent call last):
  File "C:/Python23/guess3.py", line 4, in -toplevel-
    while answer != "equal":
NameError: name 'answer' is not defined
```

This happens because we have tried to use the value in the variable `answer` before ever putting anything into it. In Python, variables don't exist until you put a value in them with a variables assignment, using `=`. So, when we try to use the value in `answer` in the `while` loop's condition, the variable doesn't exist. Python doesn't have anything to use with the name `answer` so it generates a `NameError`.

```
print("Think of a number from 1 to 100.")
smallest = 1
largest = 100
while answer != "equal":
    guess = (smallest + largest) // 2
    answer = input( "Is your number 'more', 'less'," \
            " or 'equal' to " + str(guess) + "? " )
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1

    print(smallest, largest)

print("I got it!")
```

Figure 3.9: Guessing game: trying a loop

This is a good example of problems that can come up when translating pseudocode into a programming language. There isn't always a nice, neat translation; you have to work with the language you're writing your program in. This is also part of the reason it's a good idea to write pseudocode in the first place: it's easier to work out the algorithm without fighting with the programming language.

In order to get around this, we have to get *something* in the variable `answer` before we try to use its value. We could copy the two statements that assign values to `guess` and `answer` outside of the loop, but that could be a little hard to work with later: if we have to fix the code, we have to fix two things instead of one.

The easiest thing to do is just put a dummy value in `answer` so the variable exists, but we're still sure the condition is false. This has been done in Figure 3.10.

Let's try this program. Suppose we're thinking of the number 43 and play the game:

```
Think of a number from 1 to 100.
Is your number 'more', 'less', or 'equal' to 50? less
1 49
```

```
print("Think of a number from 1 to 100.")
smallest = 1
largest = 100
answer = ""
while answer != "equal":
    guess = (smallest + largest) // 2
    answer = input( "Is your number 'more', 'less'," \
            " or 'equal' to " + str(guess) + "? " )
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1

    print(smallest, largest)

print("I got it!")
```

Figure 3.10: Guessing game: a working loop

```
Is your number 'more', 'less', or 'equal' to 25? more
26 49
Is your number 'more', 'less', or 'equal' to 37? more
38 49
Is your number 'more', 'less', or 'equal' to 43? equal
38 49
I got it!
```

There is still some extra output that we can use while testing the program. After each guess, it prints out the current range of numbers it's considering (the lines like `26 49`). Don't be afraid to `print` out extra stuff like this to help you figure out exactly what your program's doing while testing.

Notice that in Figure 3.10, we have a `if` block inside of a `while` loop. If you want to do that, or include a loop in a loop, or an `if` in an `if`, just increase the amount of indenting so the inside block is indented more.

Finally, we have a working guessing game program. A final polished version has been created in Figure 3.11.

You'll notice in Figure 3.11 that we have added some *comments* to the code. In Python, comments are on lines that start with a `#`.

```
print("Think of a number from 1 to 100.")
# start with the range 1-100
smallest = 1
largest = 100
# initialize answer to prevent NameError
answer = ""
while answer != "equal":
    # make a guess
    guess = (smallest + largest) // 2
    answer = input( "Is your number 'more', 'less'," \
            " or 'equal' to " + str(guess) + "? " )

    # update the range of possible numbers
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1

print("I got it!")
```

Figure 3.11: Guessing game: final version

```
write "Think of a number between 1 and 100."
set $guess$ to 1
until the user answers "equal", do this:
    write "Is your number equal to or not equal to $guess$?"
    read $answer$
    set $guess$ to $guess + 1$
```

Figure 3.12: A much worse version of the guessing game

Comments let you include information that is meant only for the programmer. The Python interpreter ignores comments. They are used only to make code easier to understand. You should be in the habit of writing comments on sections of code that briefly describe what the code does.

Topic 3.6 Running Time

In this section, we will explore how long it takes for algorithms to run. The running time of an algorithm will be part of what determines how fast a program runs. Faster algorithms mean faster programs, often much faster, as we will see.

The running time of algorithms is a very important aspect of computing science. We will approach it here by working through some examples and determining their running time.

The Guessing Game

When you were experimenting with the guessing game algorithm and program in Topics 1.4 and 3.5, you probably tried the game with a few different numbers. Hopefully, you noticed that this algorithm focuses in on the number you're guessing quite quickly. It takes at most seven guesses to get your number, no matter which one you're thinking of.

Suppose we had written the program from the pseudocode in Figure 3.12.

This algorithm starts at zero and guesses 1, 2, 3, . . . , until the user finally enters "equal". It does solve the "guess the number between 1 and 100" problem and it meets the other criteria in the definition of an algorithm.

What's different about this algorithm is the amount work it has to do to finish. Instead of at most seven guesses, this algorithm requires up to 100. Obviously, if we're trying to write a fast program, an algorithm that requires seven steps to solve a problem is much faster than one that takes 100.

Suppose we were writing a guessing game that guessed a number from 1 to n. The value of n could be determined when we write the program or by asking the user for the "size" of the game before we start. How many steps would each algorithm take if we modified it to guess a number from 1 to n?

The algorithm in Figure 3.12 would take up to n steps (if the user was thinking of n).

The number of guesses needed by the original algorithm from Figure 1.4 is a little harder to figure out. Each time the algorithm makes a guess, it chops the range from *smallest* to *largest* in half. The number of times we can cut the range 1 to n in half before getting down to one possibility is $\lceil \log_2 n \rceil$.

The notation $\lceil x \rceil$ means "round up". It's the opposite of the floor notation, $\lfloor x \rfloor$ and is usually pronounced "*ceiling*".

The mathematical expression $\log_2 n$ is the "base-2 logarithm of n". It's the power you have to raise 2 to to get n. So, if we let $x = \log_2 n$, then it's always true that $2^x = n$.

Having a running time around $\log_2 n$ steps is good since it grows so slowly when n increases:

$$\begin{aligned} \log_2 1 &= 0\,, \\ \log_2 16 &= 4\,, \\ \log_2 1024 &= 10\,, \\ \log_2 1048576 &= 20\,. \end{aligned}$$

We could give this program inputs with $n = 1000000$ and it would still only take about 20 steps.

Why does this algorithm take about $\log_2 n$ steps? Consider the number of possible values that could still be the value the user is thinking of. Remember that this algorithm cuts the number of possibilities in half with each step.

Step	Possible values
0	$n = n/2^0$
1	$n/2 = n/2^1$
2	$n/4 = n/2^2$
3	$n/8 = n/2^3$
k	$n/2^k$

In the worst case, the game will end when there is only one possibility left. That is, it will end after k steps, where

$$\begin{aligned} 1 &= n/2^k \\ 2^k &= n \\ \log_2 2^k &= \log_2 n \\ k &= \log_2 n\,. \end{aligned}$$

So, it will take $\log_2 n$ steps.

Basically, any time you can create an algorithm like this that cuts the problem in half with every iteration, it's going to be fast.

Remember: any time you have a loop that cuts the problem in half with each iteration, it will loop $\log n$ times. If you understand the above derivation, good. If you'd just like to take it on faith, that's fine too.

Repeated Letters

Now consider the algorithm in Figure 3.13. It will check a word (or any string, really) to see if any character is repeated anywhere. For example, the word "jelly" has a repeated "l"; the word "donuts" has no repeated letters. This algorithm works by taking leach letter in the word, one at a time, and checking each letter to the right to see if its the same.

For example, with the word "Lenny", it will make these comparisons:

L	is compared with	e, n, n, y	(none are equal)
e	is compared with	n, n, y	(none are equal)
n	is compared with	n, y	(equals the "n")
n	is compared with	y	(none are equal)
y	is compared with	nothing	

In the third line, it will compare the first and second "n" and notice that they are equal. So, this word has repeated letters.

You can find a Python implementation of this program in Figure 3.14. The only new thing in this program: you can get character `n` out of a string `str` by using the expression `str[n]`. This is called *string subscripting*. Note that the first character in the string is `str[0]`, not `str[1]`.

This program makes $n(n-1)/2 = n^2/2 - n/2$ comparisons if you enter a string with n characters. When measuring running time of a program,

```
write "Enter the word:"
read word
set counter to 0
for all letters letter_a in the word, do this:
    for all letters letter_b to the right of letter_a, do this:
        if letter_a is equal to letter_b then
            set counter to counter+1
if $counter > 0$ then
    write "There are repetitions"
else
    write "No repetitions"
```

Figure 3.13: Algorithm to check for repeated letters in a word

```
word = input("Enter the word: ")
counter = 0
length = len(word)

for i in range(length):
    # for each letter in the word...
    for j in range(i+1, length):
        # for each letter after that one...
        if word[i]==word[j]:
            counter = counter + 1

if counter > 0:
    print("There are repeated letters")
else:
    print("There are no repeated letters")
```

Figure 3.14: Repeated letters Python implementation

we won't generally be concerned with the smaller terms because they don't change things too much as n gets larger (we'll ignore $n/2$ in the example).

We generally aren't concerned with the constant in front of the term (the $\frac{1}{2}$ on the n^2). So, we will say that the algorithm in Figure 3.13 has a running time of n^2.

You'd be right if you think that throwing away the $\frac{1}{2}$ is losing a lot of information. The difference between an algorithm that finishes in 100 or 200 seconds is significant. The problem is that it's too hard to come up with a factor like this that actually *means* something. The algorithm could easily run twice as fast or half as fast if it was implemented in a different programming language, using different commands in the language, on a different computer, and so on.
Bottom line: is the $\frac{1}{2}$ a big deal? Yes, but we don't worry about it when estimating running times.

If you are taking or have taken MACM 101, you might recognize all of this as the same thing that's done with big-O notation. We would say that the repeated letters algorithm has a running time of $O(n^2)$. If you haven't taken MACM 101, watch for the big-O stuff if you do.

Subset Sum

Let's consider one final example where the best known solution is *very* slow.

Suppose we get a list of integers from the user, and are asked if some of them (a subset) add up to a particular target value. This problem is known as "subset sum", since we are asked to find a *subset* that *sums* to the given value.

For example, we might be asked to find a subset of 6, 14, 127, 7, 2, 8 that add up to 16. In this case, we can. We can take the 6, 2, and 8: $6+2+8 = 14$. In this example, we should answer "yes".

If we used the same list of numbers, but had a target of 12, there is no subset that adds to 12. We should answer "no" in that case.

Some rough pseudocode for the subset-sum problem can be found in Figure 3.15. This algorithm will solve the problem. It simply checks every possible subset of the original list. If one sums to the target, we can answer "yes"; if none do, the answer is "no".

```
for every subset in the list:
    set $sum$ to to the sum of this subset
    if $sum$ is equal to $target$:
        answer “yes” and quit
answer “no”
```

Figure 3.15: Algorithm to solve the subset-sum problem.

n	$2^n \approx$	Approx. time
4	16	16 milliseconds
10	10^3	1 second
20	10^6	17.7 minutes
30	10^9	11.6 days
40	10^{12}	31.7 years

Figure 3.16: Running time of an exponential algorithm

What is the running time of this algorithm? It depends on the number of times the loop runs. If we have a list of n items, there are 2^n subsets, so the running time will be exponential: 2^n.

More accurately, the running time is $n2^n$, since calculating the sum of the subset takes up to n steps.

An exponential running time is *very* slow. It’s so slow that exponential running times are often not even considered “real” solutions. Consider Figure 3.16. It gives running times of a 2^n algorithm, assuming an implementation and computer that can do 1000 iterations of the loop per second.

As you can see from Figure 3.16, this algorithm can only solve subset-sum for the smallest of cases. Even if we find a computer that is much faster, we can only increase the solvable values of n by a few values.

Can we do better than the algorithm in Figure 3.15? Maybe. There are no known sub-exponential algorithms for this problem (or others in a large class of equally-difficult problems), but there is also no proof that they don’t exist.

Number of "Steps"

We have avoided giving the exact definition of a "step" when calculating running times. In general, pick a statement in the *innermost* loop, and count the number of times it runs.

Usually, you can multiply together the number of iterations of the nested loops. For example, consider an algorithm like this one:

```
statement 1
for $i$ from 1 to $\log n$:
    statement 2
    for $j$ from 1 to $n/2$:
        statement 3
```

Here, "statement 3" is in the innermost loop, so we will count the number of times it executes. The first **for** loop runs $\log n$ times, and the second runs $n/2$ times. So, the running time is $(\log n) \cdot (n/2)$. We discard the constant factor and get a running time of $n \log n$.

Remember that when determining running time, we will throw away lower-order terms and leading constants. That means we don't have to count anything in "shallower" loops, since they will contribute lower-order terms. Similarly, we don't have to worry about how many statements are in the loops; that will only create a leading constants, which will be discarded anyway.

Summary

We have now seem algorithms that have running time $\log n$, n, n^2 and 2^n. See Figure 3.17 for a comparison of how quickly these times grow as we increase n. In the graph, 2^n has been excluded, because it grows too fast to see without making a much larger graph.

As you can see, the $\log_2 n$ function is growing *very* slowly and n^2 is growing quite fast.

Coming up with algorithms with good running times for problems can be very hard. We will see a few more examples in this course. A lot of computing scientists spend a lot of time working on efficient algorithms for particular problems.

For a particular algorithm, you can come up with programs that run faster or slower because of the way they are written. It's often possible to decrease the number of iterations slightly or make the calculations more efficient.

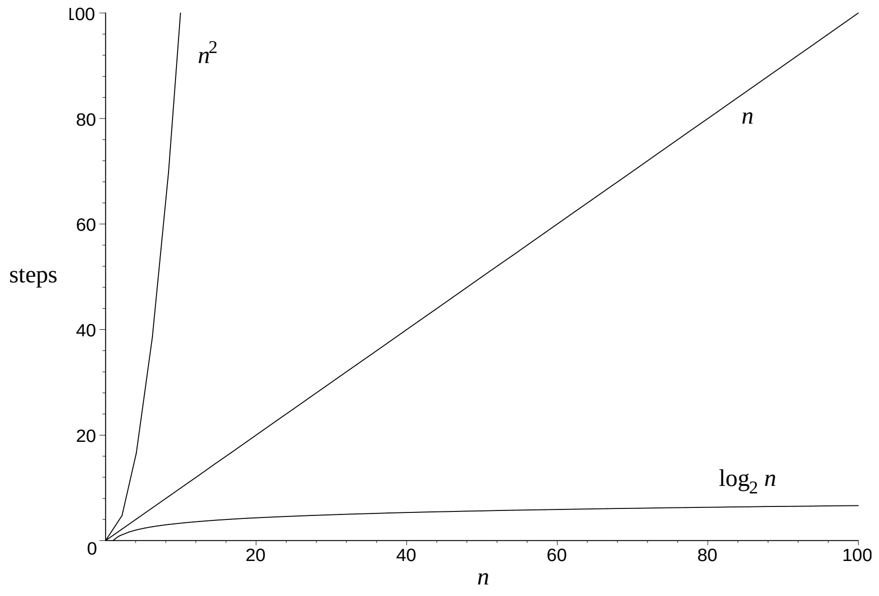


Figure 3.17: Graph of the functions $\log_2 n$, n, and n^2

But no matter how fast the program is you've written, a better algorithm will *always* win for large inputs. This is why most computing science courses spend so much time focusing on algorithms instead of the details of programming. The programming part is easy (once you learn how, anyway). It's coming up with correct, fast algorithms that's hard.

Running time is fundamental when it comes to studying algorithms. It is covered in CMPT 225 (Data Structures and Programming), CMPT 307 (Data Structures and Algorithms), and many other courses. The mathematical details of the analysis are covered in MACM 101 (Discrete Math I).

Topic 3.7 Debugging

Unfortunately, when you write programs, they usually won't work the first time. They will have errors or *bugs*. This is perfectly normal, and you shouldn't get discouraged when your programs don't work the first time. Debugging is as much a part of programming as writing code.

Section 1.3 and Appendix A in *How to Think Like a Computer Scientist* cover the topic of bugs and debugging very well, so we won't repeat too much here. You should read those before you start to write programs on your own.

Beginning programmers often make the mistake of concentrating too much on trying to fix errors in their programs without understanding what causes them. If you start to make random changes to your code in the hopes of getting it to work, you're probably going to introduce more errors and make everything worse.

When you realize there's a problem with your program, you should do things **in this order**:

1. Figure out where the problem is.
2. Figure out what's wrong.
3. Fix it.

Getting it right the first time

The easiest way to get through the first two steps here quickly is to write your programs so you know what parts are working and what parts might not be.

Write small pieces of code and **test them as you go**. As you write your first few programs, it's perfectly reasonable to test your program with every new line or two of code.

It's almost impossible to debug a complete program if you haven't tested any of it. If you get yourself into this situation, it's often easier to remove most of the code and add it back slowly, testing as you do. Obviously, it is much easier to test as you write.

Don't write your whole program without testing and then ask the TAs to fix it. Basically, they would have to rewrite your whole program to fix it, and they aren't going to do that.

As you add code and test, you should temporarily insert some `print` statements. These will let you test the values that are stored in variables so you can confirm that they are holding the correct values. If not, you have a bug somewhere in the code you've written and should fix it before you move on.

In the two example "Problem Solving" topics, 2.7 and 3.5, the program was written in small pieces to illustrate this approach.

Finding bugs

Unfortunately, you won't always catch every problem in your code as you write it, no matter how careful you are. Sooner or later, you'll realize there is a bug *somewhere* in your program that is causing problems.

Again, you should resist the urge to try to fix the problem before you know what's wrong. Appendix A of *How to Think Like a Computer Scientist* talks about different kinds of errors and what to do about them.

When you realize you have a bug in your program, you're going to have to figure out where it is. When you are narrowing the source of a bug, the `print` statement can be your best friend.

Usually, you'll first notice either that a variable doesn't contain the value you think it should or that the flow of control isn't the way you think it should be because the wrong part of an `if` is executed.

You need to work backwards from the symptom of the bug to its cause. For example, suppose you had an `if` statement like this:

```
if length*width < possible_area:
```

If the condition doesn't seem to be working properly, you need to figure out why. You can add in some `print` statements to help you figure out what's really going on. For example,

```
print("l*w:", length*width)
print("possible:", possible_area)
if length*width < possible_area:
    print("I'm here")
```

When you check this way, be sure to copy and paste the exact expressions you're testing. If you accidentally mistype them here, it could take a *long* time to figure out what has happened.

You'll probably find that at least one of the `print` statements isn't doing what it should. In the example, suppose the value of `length*width` wasn't what we expected. Then, we could look at both variables separately:

```
print("l, w:", length, width)
```

If `length` was wrong, you would have to backtrack further and look at whatever code sets `length`. Remove these `print` statements and add in some more around the `length=`... statement.

Topic 3.8 Coding Style

Writing code that is *correct* and solves the problem isn't always enough. It's also important to write code that someone can actually read and understand.

It is often necessary for you or others to return to some code and add features or fix problems. In fact, in commercial software, most of the expense is in maintenance, not in the initial writing of the code.

It can be quite difficult to look at someone else's code (or even your own code after a few months) and figure out what's going on. In order to fix bugs or add features, you need to understand the logic and details of the code, otherwise you'll probably break more than you fix.

To help others (and yourself) understand the code you've written, it's important to try to make everything as clear as possible. There are no absolute rules in this, but there are some guidelines you can follow.

Comments

Comments can be used in your code to describe what's happening. In Python, the number sign (`#`, also called the hash or pound sign) is used to start a comment. Everything on a line after the `#` is ignored: it is there for the programmer only, and does not affect the way the program runs.

In your programs, you should use comments to explain difficult parts of the code. The comment should explain what is happening and/or why it needs to be done. This can include a description of the algorithm and purpose, if it's not immediately clear.

Often, when beginning programmers are told "comments are good," the results are something like this:

```
# add one to x
x = x + 1
```

Don't do that. Anyone reading your code should understand Python, and doesn't need the language explained to them. Comments that actually explain *how* or *why* are much more useful:

```
# the last entry was garbage data, so ignore it
count = count - 1
```

That comment will help somebody reading your code understand why it was necessary to decrease `count`.

It is often useful to put a comment at the start of each control structure, explaining what it does, or what the condition checks for. Here are some examples:

```
# if the user entered good data, add it in
if value >= 0:
    total = total + value
    count = count + 1

# search for a value that divides num
while num%factor != 0:
    factor = factor + 1
```

You can also put a comment at the top of a "section" of code. Look for chunks of code that do a specific task, and put a comment at the top that describes what that task is, and how it's done. For example,

```
# Get user input
# Ask for integers, adding them up, until the user
# enters "0".
```

The Code Itself

The way the code itself is written can make a huge difference in readability and maintainability.

Probably the easiest thing to do is to use good variable names. Variable names should describe what the variable holds and what it's for. Consider the first example above with poorly chosen variable names:

```
if x >= 0:
    y = y + x
    z = z + 1
```

It would take a lot of careful reading to figure out what this code actually does. With descriptive variable names, it's much easier, even without the comment:

```
if value >= 0:
    total = total + value
    count = count + 1
```

```
a = int(input("Enter an integer: "))
b = 0
for i in range(a+1):
    if i>0:
        if a % i==0:
            b = b + 1
print("There are " + str(b) + " factors.")
```

Figure 3.18: A program with poor style

There is a bit of a trade-off between the length of the variable name and how readable it is. On one hand, short variables names like `a`, `n1`, and `x` aren't very descriptive. But, variable names that are too long can be difficult to type and clutter code. The name `total_number_of_values_entered_by_user` may be very descriptive, but code using it would be unreadable. Perhaps `values_entered` would be better.

The spacing in your code is also important.

In Python, you are required to indent the blocks of code inside a control statement. In many other programming languages, this is optional, but still considered good practice. In Python, you should be consistent in your spacing: the standard style is to indent each block by four spaces.

Spacing *within* a statement can help readability as well. Consider these two (functionally identical) statements:

```
y = 100 / x+1
y = 100/x + 1
```

The spacing in the first one suggests that the `x+1` calculation is done first (as in `100/(x+1)`), but this is not the case. The order of operations in Python dictate that the expression is equivalent to `(100/x)+1`. So, the second spacing gives a more accurate first impression.

You can use space within lines, and blank lines in the code, to separate sections logically. This will make it easier to scan the code later and pick out the units.

Summary

Again, there are no absolute rules for coding style. It is overall a matter of opinion, but the guidelines above can be followed to point you in the right

```
# get user input and initialize
num = int(input("Enter an integer: "))
count = 0

# check every possible factor (1...num)
for factor in range(1, num+1):
    # if factor divides num, it really is a factor
    if num%factor == 0:
        count = count + 1

# output results
print("There are " + str(count) + " factors.")
```

Figure 3.19: Figure 3.18 with the style improved

direction.

When you are writing code, you should always keep in mind how easy it is to read and follow the code. Add comments and restructure code where necessary.

Consider the program in Figure 3.18. What does it do? How is it being done? Does the text displayed to the user in the last line help?

Now look at Figure 3.19. This program does the exact same thing, but has better style. Better variable names and a few comments make a big difference in how easy it is to understand. Even if you don't know what the `%` operator does in the `if` condition, you can probably figure out what the program does.

`a%b` computes the remainder of `a` divided by `b`. When it evaluates to zero, there is no remainder: `a` is evenly divisible by `b`.

There is another change in Figure 3.19 that is worth mentioning. The logic of the program was changed slightly to simplify it. The `if` in Figure 3.18 is only necessary to eliminate the case where the loop variable is zero: checking this case causes a division by zero error. We can change the `range` so the loop avoids this case in the first place.

This illustrates a final important aspect of coding style: the actual logic of the program. The first way you think of to accomplish something might not be the simplest. Always be on the lookout for more straightforward ways to do what needs to be done. You can often eliminate some logic in your program in favour of something that does the same thing in an easier way.

This, along with other aspects of coding style, takes experience. You will learn new methods and tricks to get things done in a program as you write more code, read more code, and take more courses. Be patient and keep your eyes open for new techniques.

▶ Go back to the code you wrote for the first assignment in this course. Can you easily understand how it works? Keep in mind that it's only been a few weeks: imagine coming back to it next year.

▶ Now, try to swap code with someone else in the course. Can you understand each other's code?

Topic 3.9 More About Algorithms

Now that you know about variables, conditionals, and loops, you have all of the building blocks you need to start implementing algorithms. (You'll still need to know some more about working with data structures before you can implement *any* algorithm. We will talk more about data structures in Unit 5.)

We have also said that coming up with an algorithm is generally much harder than implementing the algorithm with a programming language. Creating algorithms is something you'll practice over the next few years if you continue in computing science.

Binary Conversion

To get you started thinking about creating algorithms, we'll do one example here. In Topic 2.6, we talked about how to convert a binary value to decimal: each bit is multiplied by the next power of two and the results added.

But, how can we do the opposite conversion? If I give you the number 13, how can you determine its (unsigned) binary representation? (For the record, it's 1101.)

First, let's assume we're limited to 8-bit binary numbers. The 8-bit representation of the number 13 is shown in Figure 3.20, with the numeric value

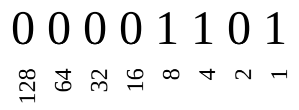


Figure 3.20: The number 13 represented with 8 bits

of each bit. We can check to see that this is the correct representation: $8 + 4 + 1 = 13$.

Now, back to the question of how we could come up with this. Let's try to get the eight-bit representation for the number 25.

First an easy question: do we need a 0 or 1 in the first position (the 128's position)? Obviously not, 128 is much bigger than 25, so adding in a 128 will make the whole thing too big. In fact, we can fill in the highest three bits this way. They are all larger than 25, so we'll definitely want 0's there:

000?????

Well, at least we're getting somewhere: we have the first few bits taken care of. Now, do we want a 0 or 1 in the next position (16)? According to the above reasoning, we *could* put a 1, but is that necessarily right? Suppose we don't: put a 0 in the 16's position. Then all of the bits we have left to fill are 8, 4, 2, and 1. These add up to 15, so there's *no way* we could get 25 out of that. We have to put a 1 in the 16's position:

0001????

For the rest of the positions, we can continue in a similar way. We have taken care of 16 of the number 25 we're trying to represent, so we have 9 left. Keep going down the line: for the 8's position, 8 is less than 9, so put a 1 in that position:

00011???

We now have 1 left to represent. We can't take a 4 or a 2, but will set the last bit to 1:

00011001

This has described a fairly simple algorithm to determine the binary representation of a number: if the bit we're looking at will fit in the number we have left, put a 1; if not, put a 0. Pseudocode for this algorithm is in Figure 3.21.

```
read $num$
for positions $p$ from 7, 6, ... 0, do this:
    if $num < 2^p$, then
        set $binary$ to $binary$ + "0"
    otherwise,
        set $binary$ to $binary$ + "1"
        set $num$ to $num - 2^p$
write $binary$
```

Figure 3.21: Pseudocode to determine the 8-bit binary representation

This algorithm only works for numbers up to 255. If you give the algorithm a number any bigger than that, it will produce all 1's. It's possible to fix the algorithm by starting with bit $\lfloor \log_2 n \rfloor$, instead of bit 7.

So?

You should probably know how to convert a number to its binary representation, but that's not the point of this topic.

As you go on into computing science, development of algorithms is important. The point here is to give you an idea of how you can go about coming up with an algorithm. Start by trying to work out the problem by hand and try to recognize the common steps and decisions needed. Try to work this into pseudocode expressing a general method and test it on different values.

Once you have the pseudocode, you can then start working on a program that implements the algorithm you've developed.

Summary

At this point, we have seen all of the major building blocks of computer programs. Once you have loops and conditionals, you can combine them to tackle just about any problem.

Again with this material, you have to practice these ideas by writing programs before you'll really understand them.

Key Terms

- `if` statement
- condition
- boolean expression
- `for` loop
- `while` loop
- running time
- debugging

Unit 4

Functions and Decomposition

Learning Outcomes

- Design and implement functions to carry out a particular task.
- Begin to evaluate when it is necessary to split some work into functions.
- Locate the parts of a program where particular variables are available.
- Import Python modules and use their contents.
- Read the Python module reference for information on a module's contents.
- Use objects provided by modules in your programs.
- Catch errors in programs and handle them gracefully.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Sections 3.6–3.12, 4.8, 5.1–5.4, 5.6 in *How to Think Like a Computer Scientist.*

Topic 4.1 Defining Functions

We have already seen how several *functions* work in Python. In particular, we have used `input`, `range`, `int`, and `str`. Each of these is built into Python and can be used in any Python program

```
def read_integer(prompt):
    """Read an integer from the user and return it."""
    entered = input(prompt)
    return int(entered)

num = read_integer("Type a number: ")
print("One more is", num+1)

num = read_integer("Type another: ")
print("One less is", num-1)
```

Figure 4.1: A program with a function defined

A function must be given *arguments*. These are the values in parentheses that come after the name of the function. For example, in `int("321")`, the string `"321"` is the argument. Functions can have no arguments, or they can take several.

Functions that *return* values can be used as part of an expression. We saw how the `int` function works, which returns an integer. It can be used in an expression like this:

```
x = 3*int("10") + 2
```

After this statement, the variable `x` will contain the number 32. In this expression the `int` function returns the integer 10, which is then used in the calculation.

Python functions can return any type of value including strings and floating point values.

Defining your own functions

You can define your own functions as well. They are defined with a `def` block, as shown in Figure 4.1. The code inside the `def` isn't executed right away. The function is defined and then run whenever it is *called*.

In Figure 4.1, the function is named "`read_integer`" and takes one argument that we'll call `prompt`. Inside the function definition, `prompt` works like a variable. Its value is filled in with whatever argument is given when the function is called.

The next line is a triple-quoted string that describes the function. This is called a *documentation string* or *docstring*. The docstring is a special form of a comment in Python: it has no effect on the behaviour of the function. It works like a comment and will help somebody reading your code figure out what it does.

Every function you write in this course must have a meaningful docstring. It will help us understand your code more easily when we mark it. It is also a good habit to get into. When you have to come back to some of your own code after a few weeks, you'll be glad you included it.

The statements in the body of the function are what will be executed when the function is called. The `return` statement indicates the value that the function returns.

The main part of the program in Figure 4.1 makes two calls to the `read_integer` function. Here's what the program looks like when it's run:

```
Type a number: 15
One more is 16
Type another: 192
One less is 191
```

You should define functions to do tasks that you'll have to do several times. That way you'll only have to type and debug the code once and be able to use it many times. As a general rule, you should never copy-and-paste code. If you need to reuse code, put it in a function and call it as many times as necessary.

Defining functions is also useful when you are creating larger program. Even if you're only going to call a function once, it helps you break your program into smaller pieces. Writing and debugging many smaller pieces of code is much easier than working on one large one.

Calling Functions

Consider the example function in Figure 4.2. This does a calculation that is common in many algorithms. The docstring should be enough for you to figure out what it does.

Suppose we then run this statement:

```
half_mid = middle_value(4,2,6) / 2
```

```
def middle_value(a, b, c):
    """
    Return the median of the three arguments.  That is,
    return the value that would be second if they were
    sorted.
    """
    if a <= b <= c or a >= b >= c:
        return b
    elif b <= a <= c or b >= a >= c:
        return a
    else:
        return c
```

Figure 4.2: A sample function

What exactly happens when the computer "runs" this code?

1. The expression on the right of the variable assignment must be evaluated before the variable can be assigned, so that is done first. It evaluates the expression `middle_value(4,2,6) / 2`.
2. The sub-expressions on either sode of the division operator must be evaluated. The first is a call to our function. It then evaluates the expression `middle_value(4,2,6)`.
3. This requires calling the function in Figure 4.2. Now, this statement is put on hold while the function does its thing.
4. The function `middle_value` is called.

   (a) The arguments that are given in the calling code (`4,2,6`) are assigned to the local variables given in the argument list (`a,b,c`). So, the behaviour is as if we had code in the function that made these assignments: `a=4`, `b=2`, `c=6`.

   (b) The code in the function body then starts to execute. This code executes until it gets to the end of the function or a `return` statement. It this case, the `if` condition is false, so the `return b` statement is skipped. The condition of the `elif` is true (since `b <= a <= c`), so the `return a` statement executes.

   (c) The function returns the integer `4` and exits. Any code after the `return` doesn't execute, since the function has already stopped.

5. The calling code gets the return value, `4`. This is used in place of the function call. The expressions is now `4/2`.
6. The division is done. The original expression has evaluated to the integer `2`.
7. The integer `2` is assigned to the variable `half_mid`.

This outline of a function call is reasonably representative of any function call. When the call occurs, that code pauses until the function is finished and returns a value.

Functions that don't return values are similar. The only difference is that they are not part of a larger expression. They just execute and the calling code continues when they are done.

Why Use Functions?

Functions can be used to break your program into logical sections. You can take a specific task or calculation, and define a function that accomplishes that task or calculation. Breaking the logic of a program up into sections can make it much easier to build. You can create functions to handle parts of your algorithm, and assemble them in a much simpler main program.

Using functions well can make your program much easier to read. Functions should have descriptive names, like variables. The function should be named after what it does or what it returns. For example, `read_data_file`, `initial_guess`, or `run_menu`.

The function definitions themselves can be relatively small (and understandable) stretches of code. Someone trying to read the program can figure out one function at a time (aided by a good function name and the docstring). Then, they can move on to the main program that assembles these parts. This is generally much easier than reading (and writing and debugging) one long section of code in the main program.

Functions are also quite useful to prevent duplication of similar code. If you need to do similar tasks in different parts of the program, you could copy-and-paste code, and many beginning programmers do. But, what happens when you want to change or update that task? You have to hunt for that code everywhere it occurs and fix it in every location. This is tedious and error-prone.

If the repeated task is separated into a function, then maintaining it is much easier. It only occurs in one place, so it's easy to fix. You should *never* copy-and-paste code within a program—it creates a maintainance nightmare. Functions are one tool that can be used to unify tasks.

Check-Up Questions

- ▶ Write a function square that takes one floating point value as its argument. It should return the square of its argument.
- ▶ Have a look at programs you've written for this course. Are there places where some work has been duplicated and could be put into a function?

Topic 4.2 Variable Scope

In Figure 4.1, the argument `prompt` is only available in the `read_integer` function. If we tried to use `prompt` outside of the function, Python would give the error

```
NameError: name 'prompt' is not defined
```

It does so because `prompt` is a *local variable* in the `read_integer` function. You could also say that the variable's scope with the `read_integer` function. Any variables that are created within a function are local to that function. That means that they can't be used outside of the function.

This is actually a very good thing. It means that when you write a function, you can use a variable like `num` without worrying that some other part of the program is already using it. The function gets an entirely separate thing named `num`, and anything named `num` in the rest of the program is undisturbed.

Have a look at the program in Figure 4.3. When it's run, it produces output like this:

```
How many lines should I print? 4
*
**
***
****
```

```
def stars(num):
    """
    Return a string containing num stars.
    This could also be done with "*" * num, but that
    doesn't demonstrate local variables.

    >>> print(stars(5))
    *****
    >>> print(stars(15))
    ***************
    """
    starline = ""
    for i in range(num):
        starline = starline + "*"
    return starline

num = int(input("How many lines should I print? "))
for i in range(num):
    print(stars(i+1))
```

Figure 4.3: A program that takes advantage of local variables

There is no confusion between the variable `num` in the function and the one in the main program. When the function uses the variable `num`, it is totally unrelated to the one in the main program. The same thing happens with `i`. It is used as the loop variable for both `for` loops. Since the function has its own version of `i`, there's no conflict and both loops do what they look like they should

So, to use the function `stars`, you don't have to worry about how it was implemented—what variables names were used and for what. All you have to know it what it does.

If a programming language doesn't have this property that variables are usually local to a particular function or other part of the program, it becomes *very* hard to write large programs. Imagine trying to write some code and having to check 20 different functions every time you introduce a new variable to make sure you're not using the same name over again.

Also notice that the docstring in Figure 4.3 is much longer. It includes two examples of what the function should do. Giving examples is a good idea because it gives you something to check when you test the function. The actual behaviour should match what you've advertised in the docstring.

◈ There is actually a Python module called `doctest` that looks through your docstrings for things that look like examples of the function's use. It then checks them to make sure the examples match what actually happens.

Why Use Functions?

Local variables introduce another reason to use functions. Since variables in functions are separate from the variables elsewhere in the program, the code has very limited interaction with the rest of the program. This makes it much easier to debug programs that are separated with functions.

Functions take in values as arguments, and can return a value when they are done. They have no other way to interact with variables in other functions.

That means that they can be debugged separately: if a function does its job given the correct arguments, then it works. If there is a problem in other parts fo the program, we don't have to worry about other functions changing variable values because they can't. Each function can be checked for correctness on its own.

```
import time
print("Today is " + time.strftime("%B %d, %Y") + ".")
```

Figure 4.4: Program that prints today's date.

Since it's much easier to work with many small chunks of code than one large one, the whole writing and debugging process becomes much easier. As a programmer, you have to create and test individual functions. Once you're reasonably sure the function is corret, you can forget about it and move on to other tasks.

Topic 4.3 Python Modules

In most programming languages, you aren't expected to do everything from scratch. Some prepackaged functions come with the language, and you can use them whenever you need to. These are generally called *libraries*. In Python, each part of the whole built-in library is called a *module*.

There are a lot of modules that come with Python—it's one of the things that experienced programmers tend to like about Python.

For example, the module `time` provides functions that help you work with times and dates. The full documentation for the `time` module can be found online. It's important that a programmer can find and interpret documentation like this. It might seem daunting at first—the documentation is written for people who know how to program—it should get easier with practice.

In the documentation, you'll find that the `time` module has a function `strftime` that can be used to output the current date and time in a particular format. The program in Figure 4.4 uses the `time` module to output the current date.

When the program runs, it produces output like this:

```
Today is December 25, 2024.
```

The first line in Figure 4.4 *imports* the `time` module. Modules in Python must be imported before they can be used. There are so many modules that if they were all imported automatically, programs would take a *long* time to start up. This way, you can just import the modules you need at the start of the program.

In the next line, the function `strftime` is referred to as `time.strftime`. When modules are imported like this, you get to their contents by calling them ***`modulename.function`***. This is done in case several modules have functions or variables with the same names.

You can also import modules so that you don't have to do this: You could just call the function as `strftime`. To do that, the module's contents are imported like this:

```
from time import strftime
```

This is handy if you want to use the contents of a particular module *a lot.*

How did we know that `"%B %d, %Y"` would make it output the date in this format? We read the documentation online. The `%B` gets replaced with the full name of the current month, `%d` with the day of the month, and `%Y` with the four-digit year.

There are Python modules to do all kinds of things, far too many to mention here. There is a reference to the Python libraries linked from the course web site.

We will mention a few more modules as we cover other topics in the course. You can always go to the reference and get a full list and description of their contents.

Check-Up Questions

- ▶ Have a look at the module reference for `time` and see what else is there.
- ▶ Look at the other modules available in Python. You probably won't understand what many of them do, but have a look anyway for stuff that you do recognize.

Topic 4.4 Objects

As you start writing programs, you will often have to represent data that is more complicated that a "number" or "string". There are some other types that are built into Python. There are also more complicated types that can hold collections of other information. These are called *objects.*

Most modern programming languages have the concept of objects. You can think of an "object" in a programming language like a real-world object like a DVD player.

A DVD player has some buttons you can press that will make it do various things (play *this* DVD, go to menu, display information on-screen) and it displays various information for you (playing, stopped, current time). Each of the buttons correspond to various actions the player can take. Each item that is displayed reflects some information about the current state of the player.

Objects in a programming language are similar. Objects are collections of *properties* and *methods*.

A property works like a variable. It holds some information about the object. In the DVD player example, the current position in the movie might be a property. Part way through the movie, the position might be 1:10:41. You could use the remote to change this property to 1:00:00 if you want to re-watch the last ten minutes. In Python, you can set the value of a property directly, just like a variable.

A method works like a function. It performs some operation on the object. For the DVD player, a method might be something like "play this DVD". A method might change some of the method's properties (like set the counter to 0:00:00) and perform some other actions (start the disc spinning, put the video on the screen).

A particular kind of object is called a *class*. So in the example, there is a class called "DVD Player". When you create an object in the class, it's called an *instance*. So, your DVD player is an instance of the class "DVD Player".

An instance behaves a lot like any other variable, except it contains methods and properties. So, objects are really variables that contain variables and functions of their own.

Objects in Python

Classes in Python can be created by the programmer or can come from modules. We won't be creating our own classes in this course, just using classes provided by modules.

To instantiate an object, its *constructor* is used. This is a function that builds the object and returns it. For example, in Python, the module `datetime` provides a different set of date and time manipulation functions than the `time` module we saw in Topic 4.3. The `datetime` module provides everything in classes which contain all of the functions that can work on particular kinds of date information.

```
import datetime

newyr = datetime.date(2025, 1, 1)
print(newyr.year)                    # the year property
print(newyr.strftime("%B %d, %Y"))  # the strftime method
print(newyr)
```

Figure 4.5: Date manipulation with the `datetime` module's objects

The `datetime` module provides the class called "`date`" which can hold information about a day. Figure 4.5 shows an example of its use.

After the `datetime` module is imported, a `date` object is created. The constructor for a date object is `datetime.date()`—this function from the `datetime` module returns a `date` object. This object is stored in the `newyr` variable.

Now that we have an object to work with, we can start poking around at its properties (variables inside the object) and methods (functions inside the object).

In the first `print` statement, you can see that a `date` object has a property called `year`. You get to a property in an object the same way you get to a function inside a module: use the name of the object, a dot, and the name of the property. The `year` behaves like a variable, except it's living *inside* the `date` object named `newyr`.

The second `print` statement shows the use of a method. Date objects contain a method called `strftime` that works a lot like the function from the `time` module. The `strftime` method takes whatever date is stored in its `date` object and formats that the way you ask it to.

Finally, we see that a `date` object knows how to convert itself to a string if we just ask that it be printed. By default, it just uses a year-month-day format.

So, the program in Figure 4.5 produces this output:

```
2025
January 01, 2025
2005-01-01
```

The ways you can use an object depend on how the class has been defined. For example, some classes know how they can be "added" together with the `+` sign, but `date` doesn't:

```
>>> import datetime
>>> first = datetime.date(1989, 12, 17)
>>> print(first)
1989-12-17
>>> print(first+7)
TypeError: unsupported operand type(s) for +:
'datetime.date' and 'int'
```

So, Python doesn't know how to add the integer 7 to a date. But, it does know how to subtract dates:

```
>>> import datetime
>>> first = datetime.date(1989, 12, 17)
>>> second = datetime.date(1990, 1, 14)
>>> print(second-first)
28 days, 0:00:00
>>> type(second-first)
<type 'datetime.timedelta'>
```

So, something in the definition of the `date` class says that if you subtract two dates, you get a `timedelta` object. The `timedelta` class is also defined by the `datetime` module and its job is to hold on to lengths of time (the time between event A and event B).

This is where the power of objects begins to show itself: a programmer can create objects that represent any kind of information and "know" how to do many useful operations for that type of information. Particularly when writing larger programs, classes and objects become very useful when it comes to organizing the information your program needs to work with.

◈ Object oriented programming is important in modern programming. It will be introduced in more detail in CMPT 125 and 225.

Check-Up Question

▶ Have a look at the module reference for `datetime`. What can you do with a `timedelta` object? What other classes are provided?

```
m_str = input("Enter your height (in metres): ")

try:
    metres = float(m_str)
    feet = 39.37 * metres / 12
    print("You are " + str(feet) + " feet tall.")
except ValueError:
    print("That wasn’t a number.")
```

Figure 4.6: Catching an exception

Topic 4.5 Handling Errors

So far, whenever we did something like ask for user input, we have assumed that it will work correctly. Consider the program in Figure 2.7, where we got the user to type their height and converted it to feet. If the user enters something that can’t be converted to a float, the results are not very pretty:

```
Enter your height (in metres): tall
Traceback (most recent call last):
  File "inches0.py", line 1, in ?
    metres = float(input( \
ValueError: invalid literal for float(): tall
```

This isn’t very helpful for the user. It would be much better if we could give them another chance to answer or at least a useful error message.

Python lets you catch any kind of error, as Figure 4.6 shows. Here are two sample runs of that program:

```
Enter your height (in metres): tall
That wasn’t a number.
```

```
Enter your height (in metres): 1.8
You are 5.9055 feet tall.
```

Errors that happen while the program is running are called *exceptions*. The `try`/`except` block lets the program handle exceptions when they happen. If any exceptions happen while the `try` part is running, the `except` code is executed. It is ignored otherwise.

```
got_height = False

while not got_height:
    m_str = input("Enter your height (in metres): ")
    try:
        metres = float(m_str)
        got_height = True # if we’re here, it was converted.
    except ValueError:
        print("Please enter a number.")

feet = 39.37 * metres / 12
print("You are " + str(feet) + " feet tall.")
```

Figure 4.7: Asking until we get correct input

Figure 4.7 shows another example. In this program, the `while` loop will continue until there is no exception. The variable `got_height` is used to keep track of whether or not we have the input we need.

Check-Up Question

▶ Take a program you have written previously in this course that takes numeric input and modify it so it gives a nice error message.

Summary

This unit covers a lot of bits and pieces that don’t necessarily let your programs *do* any more, but help you write programs that are better organized and are thus easier to maintain.

The modules in Python are very useful. In this course, we will try to point out relevant modules when you need them; we don’t expect you to comb through the entire list of built-in modules every time you need something.

Key Terms

- functions
- arguments

- return value
- docstring
- variable scope
- module
- import
- object
- class
- property
- method
- exception

Part II

Problem Solving

Unit 5

Data Structures

Learning Outcomes

- Manipulate string data.
- Use lists for storing and manipulating data.
- Describe the difference between mutable and immutable data structures.
- Identify mutable and immutable data structures, and describe the difference between them.
- Describe the use of references in assignment and argument passing.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapters 7 and 8 in *How to Think Like a Computer Scientist.*

Topic 5.1 Lists

So far, all of the variables that we have used have held a single item: one integer, floating point value, or string. These types are good at storing the information they are designed for, but you will often find that you want to store a collection of values in your programs.

For example, you may want to store a list of values that have been entered by the user or a collection of values that are needed to draw a graph.

In Python, *lists* can be used to store a collection of other values. Lists in Python can hold values of any type; they are written as a comma-separated list enclosed in square brackets:

```
numlist = [23, 10, -100, 2]
words = ['zero', 'one', 'two']
junk = [0, 1, 'two', [1,1,1], 4.0]
```

Here, `numlist` is a list holding four integers; `words` holds three strings; and `junk` has five values of different types (one of them is itself a list).

Lists are like strings

To get a particular value out of a list, it can be *subscripted.* This is done just like subscripting a string to extract a single character:

```
>>> testlist = [0, 10, 20, 30, 40, 50]
>>> print(testlist[2])
20
>>> print(testlist[0])
0
>>> print(testlist[10])
IndexError: list index out of range
```

Like strings, the first element in a list is element 0.

You can determine the length of a list with the `len` function:

```
>>> print(len(testlist))
6
```

So, we can walk through each element of a list the same way we iterated over the characters in a string:

```
>>> for i in range(len(testlist)):
...     print(testlist[i], end=' ')
...
0 10 20 30 40 50
```

Lists can be joined (*concatenated*) with the `+` operator:

```
>>> testlist + [60, 70, 80]
[0, 10, 20, 30, 40, 50, 60, 70, 80]
>>> ['one', 'two', 'three'] + [1, 2, 3]
['one', 'two', 'three', 1, 2, 3]
```

Lists can also be returned by functions:

```
>>> s = 'abc-def-ghi'
>>> s.split('-')
['abc', 'def', 'ghi']
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In all of the above examples, lists are similar to strings. As far as we've seen so far, you could just think of strings as just lists of characters. Lists (so far) work just like strings, except you can put anything in each element, not just a character.

If you ever program in C, you'll find that there really isn't any "string" type in C. Strings are just implemented as lists (called arrays in C) of characters. In Python, strings get a separate data type.

Lists are different from strings

Of course, the biggest difference between lists and strings is what they can hold. A string holds only characters, but a list can hold any type of Python data.

More than that, there are many operations that you can do on lists that aren't possible on string. It's not possible to change individual parts of a string without totally rebuilding it. When you want to change a string, you need to write an expression that created a new string, and over-wrote the old variable.

With a list, you can assign a new value to a *part* of the list, without having to rebuild the entire list. This is called *element assignment*.

```
>>> colours = ['red', 'yellow', 'blue']
>>> print(colours)
['red', 'yellow', 'blue']
>>> colours[1] = 'green'
>>> print(colours)
['red', 'green', 'blue']
```

The third statement here changes a single element of the lists, without having the change the entire list (by doing a `colours=...` assignment).

It is also possible to *delete* an element from a list, using the `del` statement.

```
>>> colours = ['red', 'yellow', 'blue']
>>> del colours[1]
>>> print(colours)
['red', 'blue']
>>> del colours[1]
>>> print(colours)
['red']
```

You can also add a new element to the end of a list with the `append` method.

```
>>> colours = ['red', 'yellow', 'blue']
>>> colours.append('orange')
>>> colours.append('green')
>>> print(colours)
['red', 'yellow', 'blue', 'orange', 'green']
```

In order to do something similar with a string, a new string must be built with the `+` operator:

```
>>> letters = 'abc'
>>> letters = letters + 'd'
>>> letters = letters + 'e'
>>> print(letter)
abcde
```

You can do the same thing with lists (rebuild them with `+` or other operators), but it's slower. As another example, see Figure 5.1

All of these operations can change *part* of a list. Changing one element (or a few elements) is more efficient than creating an entirely new list. These operations can make working with lists quite efficient.

If you try change part of a string, you will get an error. We will discuss this difference further in Topic 5.5.

There are many other list operations as well: lists are a very flexible data structure. See the online Python reference for more details.

```
print("Enter some numbers, 0 to stop:")

numbers = []
x=1
while x!=0:
    x = int(input())
    if x!=0:
        numbers.append(x)

print("The numbers you entered are:")
print(numbers)
```

Figure 5.1: Building a list with the `append` method

Topic 5.2 Lists and `for` loops

In an earlier example, you might have noticed how the `range` function was used to create a list:

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

So `range` doesn't return a list, but it gives us something that can be *converted* to a list. The `range` object returned by the `range` function represents the values in the range: if we convert that to a list, we get a list of those values.

All along, we have been using the `range` function with the `for` loop:

```
for i in range(10):
    # do something with i
    ...
```

So, what's the relationship?

It turns out that the `for` loop in Python can iterate over *any* collection of values, not just those produced by the `range` function. The `range` function is a convenient way to produce a collection of integers, but not the only way. We have seen other ways to build a list, and those can be used with the `for` loop as well.

```
words = ["up", "down", "green", "cabbage"]
for word in words:
    print("Here’s a word: " + word)
```

Figure 5.2: Iterating over a list

Have a look at the code in Figure 5.2. There, the `for` loop iterates over each element in the list `words`. The `for` loop does the same thing it does with a `range`: it runs the loop body once for each element. The output of Figure 5.2 is:

```
Here’s a word: up
Here’s a word: down
Here’s a word: green
Here’s a word: cabbage
```

Iterating through the elements of a list can be quite convenient. It's common to have to do the same operation on each element of a list, and this gives an easy way to do it.

It also makes code very readable. You can interpret the loop in Figure 5.2 as "for every word in the list `words`, do this. . . ." So, the meaning of the code is very close to the way you'd read it. This is always a benefit when trying to read and maintain code.

Topic 5.3 Slicing and Dicing

Hopefully you are comfortable with indexing by now. You can access a single element from a string or list with indexing:

```
>>> colours = [’red’, ’yellow’, ’blue’]
>>> colours[1] = ’green’ # set an element with indexing
>>> print(colours[2])    # index to retrieve an element
’blue’
```

It's also possible to access several elements of a list by *slicing*. Slicing looks like indexing, but you can specify an entire range of elements:

```
>>> colours = [’red’, ’yellow’, ’green’, ’blue’]
>>> print(colours[1:3])
[’yellow’, ’green’]
```

As you can see, the slice `[1:3]` refers to elements 1–2 from the list. In general, the slice `[a:b]` extracts elements `a` to `b-1`.

You might be thinking that the slice operator should extract all of the elements from `a` to `b` (including `b`), but it stops one before that. Maybe it's not intuitive, but it does match the behaviour of the `range` function: `range(a,b)` gives you all of the integers from `a` to `b-1`, and the slice `[a:b]` gives you the elements from `a` to `b-1`.

Special Slice Positions

In addition to selecting "elements `a` to `b-1`," there are special values that can be used in a slice.

Negative values count from the *end* of a list. So, `-1` refers to the *last* item in the list, `-2` to the second-last, and so on. You can use this to (for example) extract everything *except the last element*:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> print(colours[0:-1])
['red', 'yellow', 'green']
```

If you leave out one of the values in the slice, it will default to the start or end of the list. For example, the slice `[:num]` refers to elements `0` to `num-1`. The slice `[2:]` gives elements from `2` to the end of the list. Here are some more examples:

```
>>> colours = ['red', 'yellow', 'green', 'blue', 'orange']
>>> print(colours[2:])
['green', 'blue', 'orange']
>>> print(colours[:3])
['red', 'yellow', 'green']
>>> print(colours[:-1])
['red', 'yellow', 'green', 'blue']
```

The slice `[:-1]` will always give you everything except the last element; `[1:]` will give everything but the first element. These cases in particular come up fairly often when programming. It's common to use the first element of a list (work with `thelist[0]`) and then continue with the tail (`thelist[1:]`). Similarly, you can work with the last element((`thelist[-1]`), and then use the head of the list (`thelist[:-1]`).

Manipulating Slices

You can actually do almost anything with list slices that you can do with simple indexing. For example, you can assign to a slice:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> colours[1:3] = ['yellowish', 'greenish']
>>> print(colours)
['red', 'yellowish', 'greenish', 'blue']
>>> colours[1:3] = ['pink', 'purple', 'ecru']
>>> print(colours)
['red', 'pink', 'purple', 'ecru', 'blue']
```

Notice that in the second assignment above, we assigned a list of three elements to a slice of length two. The list expands to make room or the new elements: the slice `colours[1:3]` (`['yellowish', 'greenish']`) is replaced with the list `['pink', 'purple', 'ecru']`. If the list assigned had been shorter than the slice, the list would have shrunk.

You can also remove any slice from a list:

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> del colours[1:3]
>>> print(colours)
['red', 'blue']
```

Slices give you another way to manipulate lists. With both slices and the operators and methods mentioned in Topic 5.1, you can do many things with lists.

Topic 5.4 Strings

Much of what we have seen in the previous sections about lists also applies to strings. Both are considered *sequence types* in Python. Strings are a sequence of characters; lists are a sequence of any combination of types.

In Topic 5.2, we saw that the `for` loop can iterate through any list. Lists aren't the only type that can be used in the `for` loop. Any type that represents a collection of values can be used as the "list" in a `for` loop.

Since a string represents a sequence of characters, it can be used. For example, this program iterates through the characters in a string:

```
for char in "abc":
    print("A character:", char)
```

When this is executed, it produces this output:

```
A character: a
A character: b
A character: c
```

List looping over a list, this can make your code very readable, and is often a very useful way to process a string.

Slicing Strings

In Topic 5.3, we used slices to manipulate parts of lists. You can slice strings using the same syntax as lists:

```
>>> sentence = "Look, I'm a string!"
>>> print(sentence[:5])
Look
>>> print(sentence[6:11])
I'm a
>>> print(sentence[-7:])
string!
```

But, you can't modify a string slice, just like you can't assign to a single character of a string.

```
>>> sentence = "Look, I'm a string!"
>>> sentence[:5] = "Wow"
TypeError: object doesn't support slice assignment
>>> del sentence[6:10]
TypeError: object doesn't support slice assignment
```

Just like lists, you can use the slice `[:-1]` to indicate "everything but the last character" and `[1:]` for "everything but the first character".

Topic 5.5 Mutability

You may be wondering why assigning to a slice (or single element) works for a list, but not a string. For example:

```
dots = dots + "."      # statement #1
values = values + [n]  # statement #2
values.append(n)       # statement #3
```

Figure 5.3: Manipulating strings and lists

```
>>> colours = ['red', 'yellow', 'green', 'blue']
>>> colours[1:3] = ['yellowish', 'greenish']
>>> print(colours)
['red', 'yellowish', 'greenish', 'blue']
>>> sentence = "Look, I'm a string!"
>>> sentence[:5] = "Wow"
TypeError: 'str' object does not support item assignment
```

Why is it possible to do more with lists than strings?

In fact, lists are the only data structure we have seen that can be changed *in-place*. There are several ways to modify an existing list without totally replacing it: assigning to a slice, using `del` to remove an element or slice, extending with the `append` method, and so on. Notice that none of these require creating a new list.

On the other hand, any string manipulation requires you to build a new string which can then be stored in a variable. Consider the statements in Figure 5.3. Statement #1 first builds a new string object (by evaluating the right side of the assignment, `dots + "."`), and then stores that new string in `dots`. The old value in `dots` is discarded because it's no longer in use.

In statement #2, the same thing happens with a list. A new list is built (by evaluating `values + [n]`), and the variable `values` is set to refer to it instead of its old value. The old value is discarded since it's no longer used. Each of these statements requires a lot of work if the initial string/list is large: it must be copied, along with the new item, and the old data is dropped from memory.

Statement #3 has the same effect as #2, but it happens in a very different way. In this case, the `append` method uses the *existing* list, and just adds another element to the end. This requires much less work, since the list doesn't have to be copied as part of evaluating an expression.

At the end of statement #2 or #3, the `values` variable holds the same list. In the case of #3, the list has been modified but not entirely rebuilt. This should be clear since it is not an assignment statement (i.e. a `var=`

statement). Any statement that assigns to a variable must be building a new value in the expression on the right of the `=`. If there is no assignment, the variable is still holding the same object, but the object may have changed.

Data structures that can be changed in-place like this are called *mutable*. Lists are the only mutable data structure we have seen in detail. The strings and numbers are not mutable: they are *immutable*.

Objects in Python are mutable if they contain methods that can change them without a new assignment. The `date` objects that were used in Topic 4.4 are immutable since there are no methods that can modify an existing `date` object. There are other object types in Python modules that have methods that modify them in-place: these are mutable objects.

There are two types in Python that hold "sets": `set` and `frozenset`. These hold values like lists, but they aren't in any order. They are just a collection of values and a particular value can be either in the set or not.
The only difference between a `set` and `frozenset` is that sets are mutable (contain methods like `add` to insert a new element), and frozen sets are immutable (those methods aren't included). There are instances where either is useful, so they are both available.

Topic 5.6 References

There are several cases where the contents of one variable are copied to another. In particular, here are two operations that require duplicating the variable `x`:

```
# x copied to a parameter variable in some_function:
print(some_function(x))
# x copied into y:
y = x
```

You probably don't think of copying the contents of a variable as a difficult operation, but consider the case where `x` is a list with thousands of elements. Then, making an full copy of `x` would be a lot of work for the computer, and probably unnecessary since all of its contents are already in memory.

In fact, Python avoids making copies where possible. To understand how this happens, it's important to understand *references*.

Statements:

```
my_string = "one" + "two" + "three"
my_list = [0, 10, 20]
```

Result:

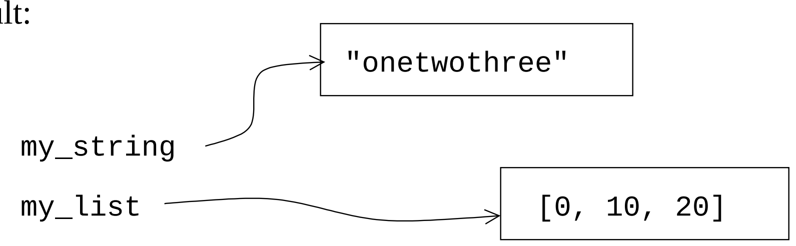


Figure 5.4: Variables referencing their contents

Every variable in Python is actually a reference to the place in memory where its contents are stored. Conceptually, you should think of a variable referencing its contents like an arrow pointing to the contents in memory. In Figure 5.4, you can see a representation of two variables referencing their contents.

When you use a variable in an expression, Python follows the reference to find its contents. When you assign to a variable, you are changing it so the variable now references different contents. (The old contents are thrown away since they are no longer being referenced.)

Usually, the expression on the right side of an assignment creates a new object in memory. The calculation is performed, and the result is stored in memory. The variable is then set to refer to this result. For example, `total = a+b` calculates `a+b`, stores this in memory, and sets `total` to reference that value.

The exception to this is when the right side of an assignment is simply a variable reference (like `total=a`). In this case, the result is already in memory and the variable can just reference the existing contents. For example, in Figure 5.5, `my_list` is created and refers to a list. When `my_list` is assigned to `list_copy`, the reference is copied, so the list is only stored once. This saves memory, and is faster since the contents don't have to be copied to another location in memory.

Statements:

```
my_list = [0, 10, 20]
list_copy = my_list
```

Result:

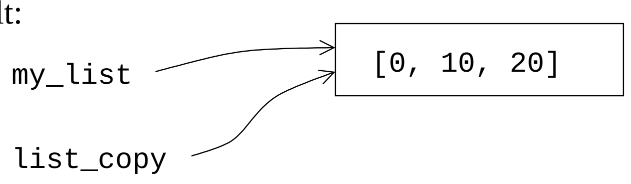


Figure 5.5: Reference copied during assignment: aliasing

Statements:

```
my_list = [0, 10, 20]
list_copy = my_list
list_copy.append(30)
my_list.append(40)
```

Result:

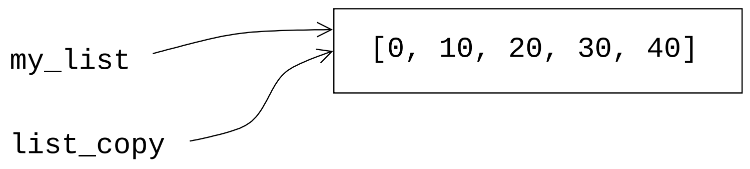


Figure 5.6: Changing either alias changes both

Aliases

When two variables refer to the same contents, they are *aliases* of each other. This has always happened when we assigned one variable to another, and it's generally good since it doesn't require copying the contents to another location in memory.

But, now that we have mutable data structures (lists and some objects), aliases complicate things. Since mutable data structures can be changed without totally rebuilding them, we can change the contents without moving the reference to a new object in memory.

That means that it's possible to change a variable, and the changes will

Statements:

```
my_string = "one" + "two" + "three"
string_copy = my_string
string_copy = string_copy + "four"
```

Result:

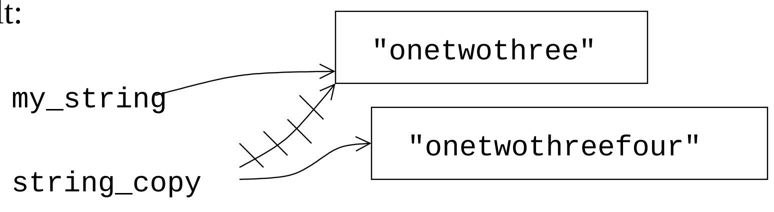


Figure 5.7: An expression creates a new reference and breaks the alias

affect any other variables *that reference the same contents*.

For example, in Figure 5.6, `my_list` and `list_copy` are aliases of the same contents. When either one is changed, both are affected. If you were to print out `my_list` and `list_copy` after these statement, you would find that they are both `[0, 10, 20, 30, 40]`.

The same things would happen if we used any methods that change the list (or any object that has such methods). So for any mutable data structure, aliasing is an issue.

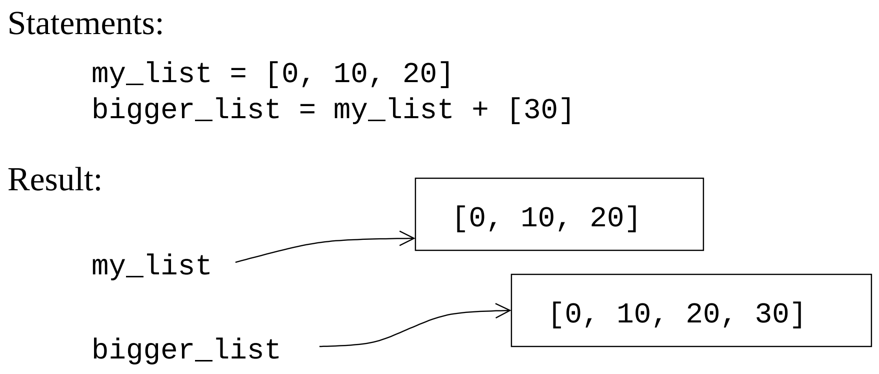
Immutable data structures can also be aliased, but since they can't be changed, it never causes problems. For example, in Figure 5.7, a string is aliased when it is copied. But the only way to change it is to construct a new string during an assignment and the alias is removed.

Remember that any expression (that's more complicated than a variable reference) will result in a new reference being created. If this is assigned to a variable, then there is no aliasing. This is what happened in Figure 5.7. It also occurs with lists, as you can see in Figure 5.8.

Really Copying

If you want to make a copy of a variable that isn't a reference, it's necessary to force Python to actually copy its contents to a new place in memory. This is called *cloning*.

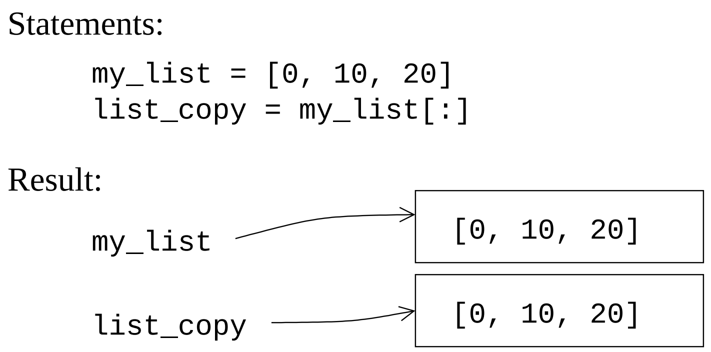
Cloning is more expensive than aliasing, but it's necessary when you do want to make a copy that can be separately modified.


Statements:

```
my_list = [0, 10, 20]
bigger_list = my_list + [30]
```

Result:

Figure 5.8: A calculation creates a new instance containing the results


Statements:

```
my_list = [0, 10, 20]
list_copy = my_list[:]
```

Result:

Figure 5.9: Slicing a list forces copying of its elements

For lists, the slice operator can be used to create a clone. Since cloning requires creating a new list, Python will copy whatever contents are needed by the slice. In Topic 5.3, we saw that in a slice like `[a:b]`, leaving out the `a` starts the slice at the start of the original list, and leaving out the `b` goes to the end of the list. If we combine these, we have a slice that refers to the entire list: `my_list[:]`.

For example, in Figure 5.9, you can see the slice operator being used to copy the contents of a list. This creates a new list and reference. Now, the lists can be changed independently.

You could also make a copy of a list with the `list` function that creates a new list (out of the old one). So, `list(my_list)` would give the same result as `my_list[:]`.

For other data types, the `copy` module contains a `copy` function. This function will take any Python object and clone its contents. If `obj` is a Python object, this code will produce a clone in `new_obj`:

```
import copy
new_obj = copy.copy(obj)
```

This should work with any mutable Python object, including lists. It will also clone immutable objects, but it's not clear why you would want to do that.

Summary

In this unit, you have learned the basics of lists in Python. You should also have picked up more tools that can be used to manipulate strings.

The concept of references might seem odd at first, but it's fundamental to many programming languages. It's one of those ideas that comes up often, and you'll have to be able to deal with it when it does.

Key Terms

- list
- append
- slice
- sequence
- mutable
- reference
- alias
- cloning

Unit 6
Algorithms

Learning Outcomes

- Use and compare two algorithms for searching in lists.
- Use sorting to solve problems.
- Design and implement functions that use recursion.
- Understand and implement a simple sorting algorithm.
- Describe some problems that aren't computable.
- Use recursion to solve problems and implement algorithms.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Sections 4.9–4.11 in *How to Think Like a Computer Scientist*.

Topic 6.1 Searching

Searching is an important program in computing. "Searching" is the problem of looking up a particular value in a list or other data structure. You generally want to find the value (if it's there) and determine its position.

We will only worry about searching in lists here. There are many other data structures that can be used to store data; each one has its own searching algorithms that can be applied.

```
def search(lst, val):
    """
    Find the first occurrence of val in lst.  Return its
    index or -1 if not there.

    >>> search([0, 10, 20, 30, 40], 30)
    3
    >>> search([0, 10, 20, 30, 40], 25)
    -1
    """
    for i in range(len(lst)):
        if lst[i]==val:
            # we only care about the first match,
            # so if we've found one, return it.
            return i

    # if we get this far, there is no val in lst.
    return -1
```

Figure 6.1: Python implementation of linear search

Linear Search

For lists in general, you have to look through the whole list to determine if the value is present or not. The algorithm for this is simple: just search through the list from element 0 to the end: if you find the value you're looking for, stop; if you never do, return -1. (We will always use the "position" -1 to indicate "not found".)

This search algorithm is called *linear search* and a Python implementation can be found in Figure 6.1.

What's the running time of a linear search for a list with n items? At worst, it will have to scan through each of the n elements, checking each one. So, the running time is n.

This isn't too bad, but if you have to do a lot of lookups in a list, it will take a lot of time. We can do better if the list is arranged properly.

◈ The list method `lst.index(val)` does a linear search to find the position of `val` in the list `lst`. If the value isn't there, it causes a `ValueError` instead of returning −1 like Figure 6.1. The "`in`" operator (`if val in lst:`) also uses linear search.

Binary Search

Think about how you look up numbers in a phone book.

If you are trying to find out who has a particular *number*, you have to look through the whole book like the linear search does. Starting with the first person in the phone book and scan all of the phone numbers until you find the phone number you're looking for, or get to the end of the book. So, it's possible to use the phone book to translate a phone number to the person's name that owns it, but it's very impractical.

On the other hand, what you usually do with a phone book is translate a persons *name* to a phone number. This is a lot easier because phone books are sorted by name, so you don't have to scan every entry; you can quickly find the person you're looking for.

So, if we have a Python list that's in order, we should be able to take advantage of this to search faster. We can write an algorithm that formalizes what you do with a phone book: use the fact that it's sorted to find the right general part of the book, then the right page, the right column, and the right name.

The algorithm that is used to search in sorted lists has a lot in common with the guessing game that was designed in Figure 1.4 and implemented in Figure 3.11. You can think of this game as searching for the value the user was thinking of in the list `[1, 2, 3, ..., 100]`.

The search algorithm will use the same strategy: keep track of the first and last possible position that the value you're looking for can be. This will start at the first and last items in the list, but can be narrowed quickly.

Then, look at the list item that's halfway between the first and last possible values. If it's too large, then you know that none of the values after it in the list are possibilities: they are *all* too large. Now you only have to look at the first half of the list, so the problem has immediately been chopped in half.

Similarly, if the value we check in the middle is too small, then we only have to look at values after it in the list. In the guessing game, we did the exact same thing, except we had to ask the user to enter “less”, “more”, or “equal”. Here, you can just check the value in the list.

This algorithm is called *binary search*. A Python implementation of binary search can be found in Figure 6.2.

Just like the original guessing game, this algorithm cuts the list we’re searching in half with each iteration. So, like that algorithm, it has running time $\log n$.

Have a look back at Figure 3.17 for a comparison of n (linear search) and $\log n$ (binary search). As you can see, the binary search will be *much* faster for large lists.

But, to use binary search, you have to keep the list in sorted order. That means that you can’t just use `list.append()` to insert something into the list anymore. New values have to be inserted into their proper location, so inserting takes a lot more work

Inserting into a sorted list takes up to n steps because Python has to shuffle the existing items in the list down to make room. You don’t see this since you can just use `list.insert()`, but it does happen behind-the-scenes.

This means that keeping a sorted list and doing binary search is only worthwhile if you need to search a lot more than you insert. If you have some data that doesn’t change very often, but need to find values regularly, it’s more efficient to keep the list sorted because it makes searches so much faster.

Searching is covered in more detail in CMPT 225 and 307. These courses discuss other data structures and how they can be used to hold sets of data so searching, inserting, and deleting are all efficient. There are data structures that can do insert, search, and delete all with running time $\log n$.

Topic 6.2 Sorting

Sorting is another important problem in computing science. It’s something that comes up very often, in a variety of situations. There are many problems that can be solved quite quickly by first sorting the values you need to work with: once the values are in order, many problems become a lot easier.

```
def binary_search(lst, val):
    """
    Find val in lst.  Return its index or -1 if not there.
    The list MUST be sorted for this to work.

    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 100)
    5
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 10)
    -1
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 2000)
    -1
    """

    # keep track of the first and last possible positions.
    first = 0
    last = len(lst)-1

    while first <= last:
        mid = (first+last)//2
        if lst[mid] == val:
            # found it
            return mid
        elif lst[mid] < val:
            # too small, only look at the right half
            first = mid+1
        else: # lst[mid] > val
            # too large, only look at the left half
            last = mid-1

    # if we get this far, there is no val in lst.
    return -1
```

Figure 6.2: Python implementation of binary search

```
word = input("Enter the word: ")
counter = 0

letters = list(word) # converts to a list of characters.
                     # 'gene' becomes ['g','e','n','e']
letters.sort()       # now identical letters are adjacent
                     # above becomes ['e','e','g','n']

for i in range(len(word)-1):
    if letters[i]==letters[i+1]:
        counter = counter + 1

if counter>0:
    print("There are repeated letters")
else:
    print("There are no repeated letters")
```

Figure 6.3: Checking for repeated letters with sorting

In the previous topic, you saw that searching is much faster if the list is sorted first. Sorting takes longer than even a linear search, but if there are going to be many searches, it is worth the work.

A list in Python can be put in order by calling its `sort` method:

```
>>> mylist = [100, -23, 12, 8, 0]
>>> mylist.sort()
>>> print mylist
[-23, 0, 8, 12, 100]
```

Example: Repeated letters with sorting

As an example of a problem where sorting can greatly speed up a solution, recall the problem of finding repeated letters in a string. Figure 3.13 gives an algorithm with running time n^2 and Figure 3.14 is a Python implementation.

Now consider the program in Figure 6.3. It does the same thing as the program in Figure 3.14, but it will be significantly faster than the previous solution for long strings.

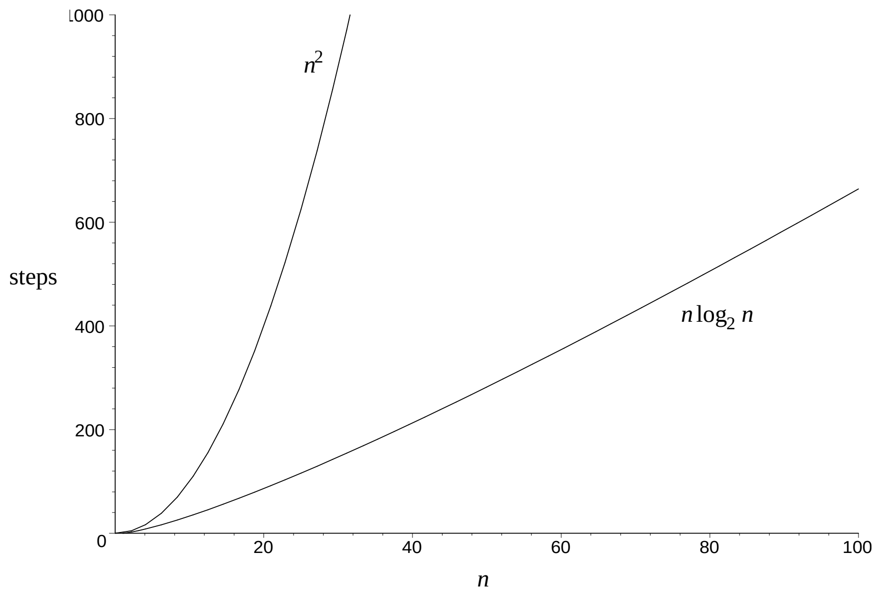


Figure 6.4: Graph of the functions n^2 and $n \log_2 n$

The idea is that the program first sorts the letters of the string. Then, any identical letters will be beside each other. So, to check for repeated letters, you only have to skim through the characters once, looking at characters `i` and `i+1` to see if they are the same. If none are, then there are no repeated letters.

The `sort` method has running time $n \log n$ on a list with n elements. The rest of the program just scans once through the list, so it takes n steps. The total running time for Figure 6.3 will be $n \log n + n$. Removing the lower-order terms, we get $n \log n$.

See Figure 6.4 for a comparison of n^2 and $n \log n$. As you can see, the new program that takes advantage of a fast sorting algorithm will be much faster as n grows.

How to sort

Usually, you can use the `sort` method that comes with a list in Python when you need to get items in order. But, sorting is important enough that you should have some idea of how it's done.

As noted above, the `sort` method of a list has running time $n \log n$. In general, it's not possible to sort n items in less than $n \log n$ running time. There are also some special cases when it's possible to sort with running time n. We won't cover these algorithms here, though.

Algorithms that sort in $n \log n$ steps are fairly complicated and will have to wait for another course. So will a proof of why that's the best you can do. If you're really interested in $n \log n$ sorting, look for *mergesort* or *quicksort* algorithms.

You probably have some idea of how to sort. Suppose you're given a pile of a dozen exam papers and are asked to put them in order by the students' names. Many people would do something like this:

1. Flip through the pile and find the paper that should go first.
2. Put that paper face-down on the table.
3. Repeat from step 1 with the remaining pile, until there are no papers left.

This method is roughly equivalent to the *selection sort* algorithm that can be used on a list. The idea behind selection sort is to scan through a list for the smallest element and swap it with the first item.

So, if we look at the list 6, 2, 8, 4, 5, 3, a selection sort will do the following operations to get it in order. At each step, the parts of the list that we *know* are sorted are bold.

Iteration	**Initial List**	**Operation**
1.	6, 2, 8, 4, 5, 3	swap 2 and 6
2.	**2**, 4, 8, 6, 5, 3	swap 3 and 4
3.	**2**, **3**, 8, 6, 5, 4	swap 4 and 8
4.	**2**, **3**, **4**, 6, 5, 8	swap 6 and 5
5.	**2**, **3**, **4**, **5**, 6, 8	swap 6 and 6 (do nothing)
6.	**2**, **3**, **4**, **5**, **6**, 8	swap 8 and 8 (do nothing)
	2, **3**, **4**, **5**, **6**, **8**	

Pseudocode of the selection sort algorithm can be found in Figure 6.5. A Python implementation can be found in Figure 6.6. This algorithm has running time n^2.

If you count the number of times the inner loop runs for each element, you find that it takes n steps for the first element, then $n-1$ for the second, then $n-2$, . . . , 2, 1. So, the total number of steps is

$$\sum_{i=1}^{n} i = \frac{n(n-1)}{2} = n^2/2 - n/2\,.$$

```
for every element $e$ from the list,
    for every element $f$ from $e$ to the end of the list,
        if $f < smallest$,
            set $smallest$ to $f$
    swap $smallest$ and $e$
```

Figure 6.5: Algorithm for selection sort

Removing lower-order terms and constants, you can see that the running time is n^2.

Selection sort will be quite slow for large lists; one of the $n \log n$ algorithms should be used instead.

Sorting is covered in more detail in CMPT 125, 225, and 307. As you progress, the focus is less on sorting itself and more on how it is used to solve other problems.

Topic 6.3 Recursion

As we have seen many times, it's possible for a function to call another function. For example, in Figure 6.1, the `search` function uses both `range` and `len` to create the appropriate loop.

But, it's also possible for a function to call *itself*. This is useful when a problem can be solved by breaking it into parts and solving the parts. This technique is called *recursion*, and is very important since many algorithms are most easily described recursively.

For example, consider calculating the *factorial* of a number. The factorial of n is usually written $n!$ and is the product of the numbers from 1 to n: $1 \times 2 \times 3 \times \cdots \times (n-1) \times n$. The factorial function is often defined in terms of itself:

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n \times (n-1)! & \text{for } n > 0 \end{cases}$$

We can use this same definition to create a Python function that calculates the factorial of a (positive) integer.

The code in Figure 6.7 defines a function that correctly calculates $n!$. It does this by calling *itself* to calculate the factorial of $n - 1$.

```
def selection_sort(lst):
    """
    Sort lst in-place using selection sort
    """
    for pos in range(len(lst)):
        # get the next smallest in lst[pos]

        # find the next smallest
        small = lst[pos] # smallest value seen so far
        smallpos = pos   # position of small in lst
        for i in range(pos+1, len(lst)):
            # check each value, searching for one
            # that's smaller than the current smallest.
            if lst[i] < small:
                small = lst[i]
                smallpos = i

        # swap it into lst[pos]
        lst[pos], lst[smallpos] = lst[smallpos], lst[pos]
```

Figure 6.6: Python implementation of selection sort

```
def factorial(n):
    """
    Calculate n! recursively.

    >>> factorial(10)
    3628800
    >>> factorial(0)
    1
    """
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```

Figure 6.7: Recursively calculate factorials

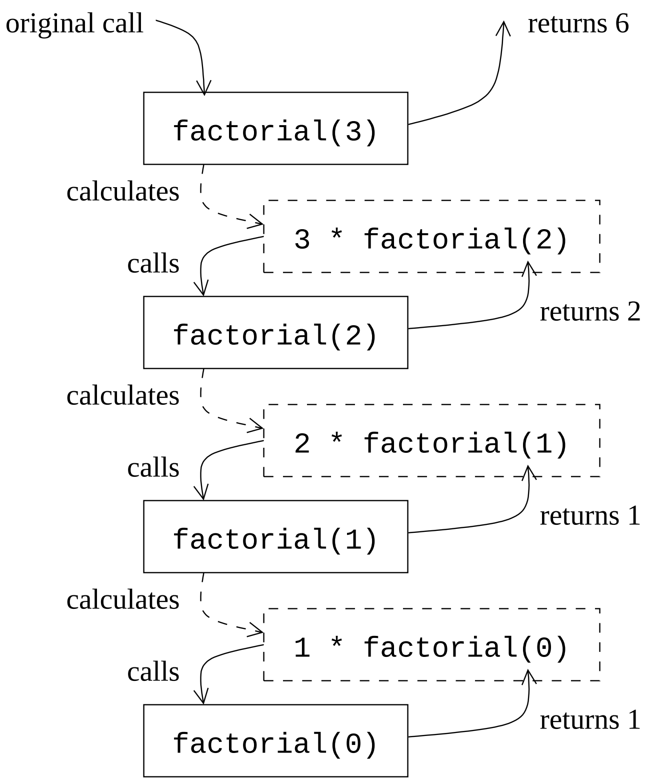


Figure 6.8: Functions calls made to calculate `factorial(3)`

How It Works

Whenever a function calls itself, you should think of a new copy of the function being made. For example, if we call `factorial(3)`, while running, it will call `factorial(2)`. This will be a separate function call, with separate arguments and local variables. It will run totally independently of the other instance of the function.

Figure 6.8 contains a diagram representing all of the calls to the `factorial` function required to calculate `factorial(3)`. As you can see, `factorial(3)` calls `factorial(2)`, which itself calls `factorial(1)`, which calls `factorial(0)`.

Because of the way the `factorial` function is written, `factorial(0)` returns one. Then, `factorial(1)` can complete its calculations and return; then `factorial(2)` can finish. Finally, `factorial(3)` completes and returns the result originally requested.

Once everything is put together, the function actually computes the correct value. This is made possible by the design of the function: as long as the recursive calls return the right value, it will calculate the correct result for `n`.

The idea that the function calls itself, or that many copies of the function are running at one time will probably seem strange at first. What you really need to remember is simple: it works. Python can keep track of several instances of the same function that are all operating simultaneously, and what each one is doing.

Understanding Recursion

When looking at a recursive algorithm, many people find it too complicated to think of every function call, like in Figure 6.7. Keeping track of every step of the recursion isn't really necessary to believe that the function works, or even to design one yourself.

When reading a recursive function, you should just assume that the recursive calls return the correct value. In the example, if you take on faith that `factorial(n-1)` returns $(n-1)!$ correctly, then it's clear that the function does the right thing in each case and actually calculates $n!$.

You can let the programming language take care of the details needed to run it.

If you look at Figure 6.7 and assume that the recursive call works, then it's clear that the whole function does as well. The key to the logic here is that the recursive calls are to *smaller* instances of the same problem. As long as the recursive calls keep getting smaller, they will eventually hit the $n = 0$ case and the recursion will stop.

When writing recursive functions, you have to make sure that the function and its recursive calls are structured so that analogous logic holds. Your function must make recursive calls that head towards the *base case* that ends the recursion.

We will discuss creating recursive functions further in the next topic.

◈ Students who have taken MACM 101 may recognize this as being very similar to proofs by induction. The ideas are very similar: we need somewhere to start (base case), and some way to take a step to a smaller case (recursive/inductive case). Once we have those, everything works out.

Topic 6.4 Designing with Recursion

As an example of a recursive function, let's look at the problem of reversing a string. We will construct a recursive function that does this, so calling `reverse("looter")` should return `"retool"`.

In order to write a recursive function to do this, we need to view the problem in the right way and create a recursive function that implements our recursive algorithm.

Step 1: Find a Smaller Subproblem

The whole point of making a recursive call is to solve a similar, but smaller problem. If we have decomposed the problem properly, we can use the recursive solution to build a solution to the problem we are trying to solve.

In the factorial example, this relies on noticing that $n! = n \times (n-1)!$ (in almost all cases). Then, if we can somehow calculate $(n-1)!$ it's easy to use that to calculate $n!$.

To reverse the string, we have to ask: if we reverse part of the string, can we use that to finish reversing the whole string? If we reverse the tail of the string (`string[1:]`, everything except the first character), we can use it.

For example, if we are trying to reverse the string `"looter"`, the tail (`string[1:]`) is or `"ooter"`. If we make a recursive call to reverse this (`reverse(string[1:])`), it should return `"retoo"`. This is a big part of the final solution, so the recursive call will have done useful work. We can later use this to build the reverse of the whole string.

Step 2: Use the Solution to the Subproblem

Once you have taken the problem you're given and found a subproblem that you can work with, you can get the result with a recursive call to the function.

In the factorial example, we know that once we have calculated $(n-1)!$, we can simply multiply by n to get $n!$. In the Python implementation, this becomes `n * factorial(n-1)`. If we put our faith in the correctness of the calculation `factorial(n-1)`, then this is definitely the correct value to return.

When reversing a string, if we reverse all but the first character of the string (`reverse(string[1:])`), we are very close to the final solution. All that remains is to put the first character (`string[0]`) on the end.

Again using `"looter"` as an example, `reverse(string[1:])` returns `"retoo"` and `string[0]` is `"l"`. The whole string reversed is:

```
reverse(string[1:]) + string[0]
```

This evaluates to `"retool"`, the correct answer. In general, this expression gives the correct reversal of `string`.

Again, in both examples the method is the same: make a recursive call on the subproblem, and use its result to construct the solution you have been asked to calculate.

Step 3: Find a Base Case

There will be a few cases where the above method won't work. Typically these will be the smallest cases where it's not possible to subdivide the problem further.

These cases can simply be handled with an `if` statement. If the arguments point to a base case, return the appropriate result. Otherwise, proceed with the recursive solution as designed above.

In the factorial example, the identity $n! = n \times (n-1)!$ isn't true for $n = 0$. In this case, we know the correct answer from the definition of factorial: $0! = 1$. In the implementation, we check for this case with an `if` statement and return the correct result. In all other cases, the recursive solution is correct, so we use it.

For reversing a string, there is also one case where the method outlined above for the recursive case can't be followed. Again, we look at the smallest cases of the problem, which cannot be subdivided further.

What should be the result when we call `reverse("")`? Presumably, the reverse of the empty string is `""`, but the above method won't give this result. In fact, if we try to extract element 0 from this string, it will cause an `IndexError` since it doesn't have a zero-th character. Since this case doesn't match the recursive algorithm, it will be our base case and handled separately.

We should also check other small cases: what is the reverse of a single-character string? The function call `reverse("X")` *should* return `"X"`. We can check our recursive method for this case, when `string` is `"X"`:

```
reverse(string[1:]) + string[0] == reverse("") + "X"
```

Since we just decided that `reverse("")` will return `""`, this becomes `"" + "X"`, which is `"X"`. This is the correct result, so we don't have to worry about single-character strings as a base case: the recursive case already handles them correctly.

It's very important every recursive call will *eventually* get to a base case. The recursive call that is made in the function must be at least one step closer to the base case: the part we decide to solve recursively is *smaller* than the original problem.

Remember that the base case is where the recursion will end. Once it does, the base case will return the correct result (since we made sure it did). From there, the recursive calls will begin to "unwind", with each one returning the correct result for the arguments it was given.

If this isn't the case, the function will keep making more and more recursive calls without ever stopping. This is called *infinite recursion.* Look back at Figure 6.8 and imagine what it would look like if we didn't have the special case for `num==0`. It would keep making more and more calls until the program was halted. This is analogous to the infinite loops you can create with `while`.

Python will stop when the recursion passes a certain "depth". It will give the error "maximum recursion depth exceeded". If you see this, you have probably created infinite recursion.

Step 4: Combine the Base and Recursive Cases

Once we have identified the base case(s) and what recursive calculation to do in other cases, we can write the recursive function.

Assembling the parts is relatively easy. In the function, first check to see if the argument(s) point to a base case. If so, just return the solution for this case. These should be very easy since the base cases are the smallest possible instances of the problem.

If we don't have a base case, then the recursive case applies. We find our subproblem and call the same function recursively to solve it. Once that's done, it should be possible to transform that into the solution we need. Once we know it, it will be returned.

Pseudocode for a recursive function can be found in Figure 6.9. Compare this with the implementation of `factorial` in Figure 6.7. Figure 6.7 doesn't

```
if we have a base case,
    return the base case solution
otherwise,
    set rec_result to the result of a recursive call on the subproblem
    return the solution, built from rec_result
```

Figure 6.9: Pseudocode for a recursive algorithm

put the recursive result in a variable because the calculation is so short, but it's otherwise the same.

The example of reversing a string has been implemented in Figure 6.10. It uses the parts constructed above and the outline in Figure 6.9. It does correctly return the reverse of any string.

Debugging Recursion

Debugging a recursive function can be trickier than non-recursive code. In particular, when designing the recursive function, we made an assumption: that the recursive call to the same function returned the correct result. Since the function is relying on itself working, tracing problems can be difficult.

The key to finding errors in recursive code is to first side-step this problem. There are cases in the recursive function that don't make recursive calls: the base cases. This gives us somewhere to start testing and debugging.

The first thing that should be done when testing or trying to find errors is to examine the base case(s). For example, in the above examples, the first things to test would be:

```
>>> factorial(0)
1
>>> reverse("")
''
```

If the base cases work correctly, then at least we have that to work with. If not, we know what code to fix.

Once we know the base cases are working, we can then easily test the *cases that call the base case.* That is, we now want to look at the arguments to the function that are one step away from the base case. For example,

```
def reverse(string):
    """
    Return the reverse of a string

    >>> reverse("bad gib")
    'big dab'
    >>> reverse("")
    ''
    """
    if len(string)==0:
        # base case
        return ""
    else:
        # recursive case
        rev_tail = reverse(string[1:])
        return rev_tail + string[0]
```

Figure 6.10: Recursively reversing a string

```
>>> factorial(1)
1
>>> reverse("X")
'X'
>>> reverse("(")
'('
```

In each case here, the recursive code runs once, and it calls the base case. We can manually check the calculations in these cases and confirm that our expectations match the results.

If these aren't correct, then there is something wrong with the way the recursive code uses the base case (which we have already tested) to compute the final solution. There are a couple of possibilities here: the wrong recursive call might be made, or the calculation done with the recursive result could be incorrect. Both of these should be checked and corrected if necessary.

If these cases work, but the overall function still isn't correct, you can proceed to cases that are another step removed (two steps from the base case). For example, `factorial(2)` and `reverse("Ab")`. You should quickly find some case that is incorrect and be able to diagnose the problem.

Another Example

As a final example of recursion, we will create a function that inserts spaces *between* characters in a string and returns the result. The function should produce results like this:

```
>>> spacedout("Hello!")
'H e l l o !'
>>> spacedout("g0ODbyE")
'g 0 O D b y E'
```

We can work through the steps outlined above to create a recursive solution.

1. [Find a Smaller Subproblem.] Again, we look for cases that are one "step" smaller and can contribute to an overall solution. Here, we will take all but the *last* character of the string, so for `"marsh"`, we will take the substring `"mars"`.

   The recursive call will be `spacedout(string[:-1])`.

2. [Use the Solution to the Subproblem.] If the recursive is working correctly, we should get a spaced-out version of the substring. In the example, this will be `"m a r s"`. We can use this to finish by adding a space and the last character to the end.

   So, the final result will be `rec_result + " " + string[-1]`.

3. [Find a Base Case.] The above method clearly won't work for the empty string since there is no last character to remove. In this case, we can just return the empty string.

   But, the above also won't work for strings with a single character. The recursive method applied to `"X"` will return `" X"`, with an extra space at the beginning. We will have to handle these as a base case as well: the correct result is to return the same string unchanged.

   In fact, these two cases can easily be combined. If the string has 0 or 1 characters, it can be returned unchanged.

4. [Combine the Base and Recursive Cases.] The above work has been combined in the implementation in Figure 6.11.

```
def spacedout(string):
    """
    Return a copy of the string with a space between
    each character.

    >>> spacedout("ab cd")
    'a b   c d'
    >>> spacedout("")
    ''
    >>> spacedout("Q")
    'Q'
    """
    if len(string) <= 1:
        return string
    else:
        head_space = spacedout(string[:-1])
        return head_space + " " + string[-1]
```

Figure 6.11: Recursively putting spaces between characters in a string

Check-Up Questions

- ▶ Repeat the spaced-out string example using everything but the *first* character as the subproblem. That is, for the string `"hearth"`, the subproblem should be `"earth"`, instead of `"heart"`. You should be able to get a different recursive function that produces the same results.
- ▶ Write a recursive function `list_sum(lst)` that adds up the numbers in the list given as its argument.
- ▶ Write a recursive function `power(x,y)` that calculates x^y, where y is a positive integer. To create a subproblem, you will have to decrease *one of* `x` or `y`. Which one gets you a useful result?

Topic 6.5 What isn't computable?

Throughout this course, we have solved many problems by creating algorithms and implementing the algorithms in Python. But, there is a fundamental question here: are there problems that you can't write a computer program to solve?

The answer is yes. It's possible to prove that, for some problems, it's impossible to write a program to solve the problem. That means that the lack of a solution isn't a failing of the programming language you're using, your programming abilities, or the amount of time or money spent on a solution. There is no solution to these problems because it's fundamentally impossible to create one with the tools we have available to do computations.

The Halting Problem

For example consider the following problem:

> Consider a particular program. Given the input given to the program, determine whether or not the program will ever finish.

This problem is called the *halting problem*. The basic idea is to decide whether or not a program "halts" on particular input. Does it eventually finish or does it go into an infinite loop?

This is something that would be very useful to be able to compute. The Python environment could have this built in and warn you if your program

```
prog = raw_input("Program file name: ")

if halts(prog, prog):
    while True:
        print "looping"
else:
    print "done"
```

Figure 6.12: Fooling a function that claims to solve the halting problem.

was never going to finish. It would be very helpful when it comes to debugging.

Unfortunately, it's *impossible* to write a program that looks at any program to determine whether or not it halts.

Suppose someone comes along who claims to have created a function `halts(prog, input)` that solves the halting problem: it returns `True` if the program in the file `prog` halts on the given input and `False` if not. You write the program in Figure 6.12 and ask if you can test their function.

This program is based on asking the `halts` function what happens when *a program is given itself as input.* So, we imagine what would happen if you ran a program and then typed its filename and pressed enter. Maybe that isn't sensible input for the program but that doesn't matter: we only care if the program halts or not, not if it does something useful.

Does the program in Figure 6.12 halt? Sometimes. If its input is a program that halts when given itself as input, Figure 6.12 enters an infinite loop. If the input program doesn't halt, then the program in Figure 6.12 stops immediately.

Suppose we run the program in Figure 6.12 and give it itself as input. So, we enter the file name of the program in Figure 6.12 at the prompt. What answer does the program give?

Without knowing anything about the `halts` function, we know that it will always give the wrong answer for these inputs. These are the two possibilities:

1. `halts("fig6.12.py", "fig6.12.py")` returns `True`. Then the program in Figure 6.12 would have entered an infinite loop: it should have returned `False`.

2. `halts("fig6.12.py", "fig6.12.py")` returns `False`. Then the program in Figure 6.12 would have printed one line and stopped: it should

have returned `True`.

So, no matter what claims are made, a program that claims to compute the halting problem will *always* make some mistakes.

The idea of feeding a program itself as input might seem strange at first, but it's a perfectly legitimate thing to do. It should be enough to convince you that there are *some* inputs where a halting function fails. There will be many more for any particular attempt.

There's nothing that says you can't write a program that answers the halting problem correctly *sometimes*. The point is that the problem in general is impossible to solve with any computer.

Virus Checking

One problem that you may have had to deal with in the real-world is keeping viruses away from your computer.

A computer *virus* is a program that is designed to spread itself from one location to another—between different programs and computers. There are many programs that scan your computer for viruses and report any problems.

All of these programs require an up to date list of virus definitions. Every week or so, they download a new list of viruses over the Internet. These "definitions" contain information like "files that contain data like this... are infected with virus X."

You may have wondered why this is necessary: why can't virus checkers just look for programs that behave like viruses. By definition, a virus has to be a program that reproduces; just look for programs that put a copy of themselves into another program.

But again, we run into a problem of computability. Writing a program to check for programs that reproduce themselves isn't possible. So, it's impossible to write a perfect anti-virus program. The most effective solution seems to be the creation of lists of viruses and how to detect them. A fairly simple program can scan a hard drive looking for *known* virus signatures.

The downside is that the program will only detect known viruses. When new viruses are created, they must be added to the list. As a result, it's necessary to update virus definition files regularly.

Again, remember that these problems (halting and virus checking) aren't impossible to compute because of some limitation in a programming language or the computer you're working with. They are impossible with *every* computational device anybody has ever thought of.

Summary

This unit covers several important aspects of algorithms and computability. Sorting, searching, and recursion are important techniques for designing efficient algorithms; you will see much more of them if you go on in Computing Science.

The uncomputability topic is important to the way problems are solved with computer. The same problems can be solved with any computer and there are some problems that can't be solved with any computer.

Key Terms

- searching
- linear search
- binary search
- sorting
- selection sort
- recursion
- base case
- uncomputable
- halting problem
- virus checking

Unit 7

Working with Files

Learning Outcomes

- Create a program that outputs information to a text file.
- Create a program that reads a text file and does some simple processing of its contents.
- Describe the role on the operating system.
- Explain in general terms how a disk stores information.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapter 11 in *How to Think Like a Computer Scientist.*

Topic 7.1 File Output

Up to this point, we have had many ways to manipulate information in the computer's memory, but haven't had any way to write information to a file on a disk. Doing so will allow us to save information permanently, and create files that can be loaded into other programs.

We will only discuss reading and writing *text files* in this course. These are files that consist only of ASCII characters. These files can be opened and edited with a *text editor* like the IDLE editor, Notepad, or Simpletext.

```
dataout = open("sample.txt", "w")

dataout.write("Some text\n")
dataout.write("...that is written to a file\n")

dataout.close()
```

Figure 7.1: Writing text to a file

```
Some text
...that is written to a file
```

Figure 7.2: File created by Figure 7.1

Writing data to a text file is quite easy in Python. Writing a text file is a lot like printing text to the screen with `print`.

Before we can send information to a file, it must be *opened*. The `file` function opens a file on the disk for our use, and returns a *file object* that represents the file. For example,

```
dataout = open("sample.txt", "w")
```

This opens file file sample.txt in the current directory for write (the `"w"` indicates that we will write to the file). A file object is returned and stored in `dataout`. Note that when a file is opened for write, any existing contents are discarded.

When you're done with a file object, is should be closed with the `close` method. This writes all of the data to the file and frees it up so other programs can use it. For example,

```
dataout.close()
```

When we have a file object opened for write, we can send text to it. This text will be stored in the file on disk. The standard way to write text to a file object is to use its `write` method. The `write` method takes a string as its argument, and the characters in the string are written to the file.

See Figure 7.1 for a complete program that creates a text file. This program produces a file sample.txt; its contents can be seen in Figure 7.2.

In the two calls to the `write` method, you can see that a *newline character* is produced to indicate a line break should be written to the file. In a Python string, `\n` is used to indicate a newline character.

```
import math
csvfile = open("sample.csv", "w")

for count in range(10):
    csvfile.write( str(count) + "," )
    csvfile.write( str(count*count) + "," )
    csvfile.write( str(math.sqrt(count)) + "\n" )

csvfile.close()
```

Figure 7.3: Writing text to a file

When using `print`, a newline character is automatically generated after each statement. When using the `write` method, this must be done manually. A line break is produced for every `\n` in the argument. For example, there would be a blank line between the two lines of text if this statement had been used:

```
dataout.write("Some text\n\n")
```

There is actually a form of the `print` statement that can be used on files. The statement `print("Hello world", file=dataout)` will "print" to the file object `dataout`.

Text files in specific formats can be used by many programs. As long as you know how information must be arranged in the file, it's possible to write a Python program to create a file than can then be imported/loaded in another program. This can be very useful in many applications.

Spreadsheet programs can import *comma-separated value* (or *CSV*) files. The format for these files is simple. Each line in the file represents a row in the spreadsheet; the cells on each line are separated by commas.

It is quite easy to produce this format in a program. For example, the program in Figure 7.3 produces a CSV file with three columns: the first column is a number, the second is the square of the number, and the third is the square root. A row is produced for every number in `range(10)`.

The output of this program can be seen in Figure 7.4. This file can be loaded into any spreadsheet program, or any other program that can import CSV files. This way, a data produced in a Python program can be passed to a spreadsheet or other program for further manipulation or analysis.

```
0,0,0.0
1,1,1.0
2,4,1.4142135623730951
3,9,1.7320508075688772
4,16,2.0
5,25,2.23606797749979
6,36,2.449489742783178
7,49,2.6457513110645907
8,64,2.8284271247461903
9,81,3.0
```

Figure 7.4: File created by Figure 7.3

The `csv` module contains objects that allow even more convenient reading and writing of CSV files. It correctly handles cells that contain commas and other special characters, which is tricky to do by-hand.

Check-Up Questions

- ▶ Run the program in Figure 7.3; it will produce a file sample.csv. Load this into a spreadsheet program.

Topic 7.2 File Input

Reading data from a file is somewhat like getting input from the user with `input`. The same problems arise: bad input and extracting the information we want from the string of characters. But, unlike user input, we can't just ask the question again if we get bad input. We either have to process the data in the file or fail outright.

A file can be opened for reading the same way as for writing, except we use `"r"` to indicate that we want to read the file.

```
datain = open("sample.txt", "r")
```

Again, a file object is returned and we store it in the variable `datain`. File input objects should also be closed when you're done using them.

```
filename = input("Enter the file name: ")
datain = open(filename, "r")

for line in datain:
    print(len(line))

datain.close()
```

Figure 7.5: Reading a text file and counting line lengths

The usual way to read a text file in Python is to process it one line at a time. Files can be very large, so we probably don't want to read the whole file into memory at once. Handling one line at a time doesn't use too much memory, and is often the most useful way to look at the file anyway.

In Topics 5.2 and 5.4, we saw that the `for` loop in Python can be used to iterate over every item in any list or string. It can also be used to iterate through file objects. The body of the `for` is executed once for every line in the file.

For example, the code in Figure 7.5 reads a text file and prints the number of characters on each line. If we give it the file from Figure 7.2, the output would be:

```
Enter the file name: sample.txt
10
29
```

Again, the `for` loop reads just like what it actually does: "`for` (every) `line in` (the) `datain` (file object)...".

Processing File Input

Have a more careful look at the text in Figure 7.2, and the sample output of Figure 7.5 above. The first line in Figure 7.2 is "`Some text`" (9 characters), but the program reports the length of the line as 10. Where did that extra character come from?

Look back at the code in Figure 7.1 that produced the file. The first call to `write` actually produces 10 characters: 9 "visible" characters and a newline. It we modified Figure 7.5 to just output the contents of `line` instead of the length, we would see an extra line break caused by the newline character in the string.

There are a couple of ways to deal with the newline character if you don't want it in the string as you process it. The easiest is to just discard the last character of each line:

```
for line in datain:
    line = line[:-1]
    ...
```

Assuming there's a newline on the *last* line of the file, this will work. If you want to get rid of the newline and *any other* spaces or tabs at the end of each line, the `rstrip` string method can do that:

```
for line in datain:
    line = line.rstrip()
    ...
```

Remember that this removes *any* trailing whitespace, which may not be appropriate in all situations.

Once the newline character has been removed (if necessary), you can process the line as you would with any other string. Of course, there are many different ways you might need to handle the lines of a file.

As an example, we will read a file that contains a time on each line: each time will be in `hh:mm:ss` format. We will calculate and report the total number of seconds represented by each time. For example, `1:00:00` is $60 \times 60 = 3600$ seconds.

In order to do this, we will read each line of the file (as in Figure 7.5). Any trailing whitespace will be removed from the line with `rstrip`. Finally, the line can be split into hour, minute, second sections by the `split` string method.

The `split` method is used to divide a string into "words" that are separated by a given *delimiter*. In this case, we want to divide the string around any colon characters, so the method call will be `string.split(":")`. This will return substrings for the hour, minute, and second, in order.

Once we have strings for each of the hour, minute, and second, these can be converted to integers and the number of seconds easily calculated. This has been done in Figure 7.6. Figure 7.7 contains a sample input file, and Figure 7.8 contains the output when the program is run.

```
timefile = open("times.txt", "r")
total_secs = 0

for line in timefile:
    # break line up into hours, minutes, seconds
    h,m,s = line.rstrip().split(":")

    # calculate total seconds on this line
    secs = 3600*int(h) + 60*int(m) + int(s)
    total_secs += secs
    print(secs)

timefile.close()
print("Total:", total_secs, "seconds")
```

Figure 7.6: Program to read a series of times in a file

```
2:34:27
0:58:10
01:09:56
0:23:01
10:12:00
```

Figure 7.7: Input file (times.txt) for Figure 7.6

```
9267
3490
4196
1381
36720
Total: 55054 seconds
```

Figure 7.8: Output of Figure 7.6

Topic 7.3 The Operating System

In Topic 7.2, you didn't have to know what part of the disk contained the file, you only had to know its file name to get at its contents.

Similarly, in the Topic 7.1, when you wanted to write data to a file, you didn't have to actually know how information was arranged on the disk or what part of the disk your file was being written to. When your data is written to the disk, something has to make sure you're given a part of the disk that isn't being used by another file; if you're using an existing file name, the old version has to be overwritten; and so on.

All of this is taken care of by the *operating system*. The operating system is a piece of software that takes care of all communication with the computer's hardware. The operating system handles all communications with the hard disk, printer, monitor, and other pieces of hardware. Thus, it can make sure that no two application programs are using the same resource at the same time.

When we write files, we are relying on the operating system to give us parts of the hard disk, and put our data there so we can retrieve it later. Since the OS takes care of all of this, we don't have to worry about it when we're programming. The operating system is also responsible for allocating parts of the computer's memory to particular programs; this is necessary when we use variables in a program.

Modern operating systems come bundled with many applications: file managers, Wordpad, Media Player, iPhoto, and so on. Microsoft has even claimed that some of these (notably, Internet Explorer) are inseparable from the operating system. Still, they are application programs, not really part of the operating system, according to its definition. Whether or not they can be separated from the OS is another problem that has more to do with marketing than computing science.

So, what really separates the operating system from other software is that the OS does all of the communication with hardware. It also mediates conflicts (if two applications want to access the hard disk at the same time) and allocates resources (giving out memory and processor time as needed). Figure 7.9 summarizes the communication between the various pieces of the system.

In Figure 7.9, there is one user (Langdon) who has four applications open.

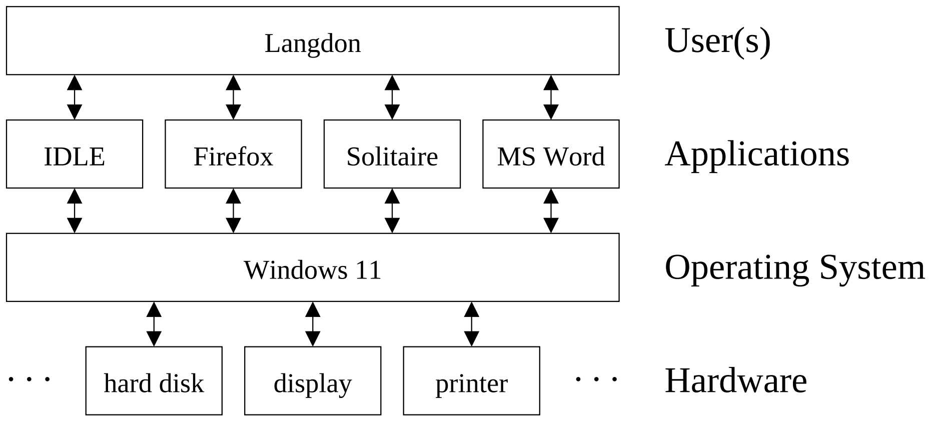


Figure 7.9: Communication between the user, applications, OS, and hardware

Whenever any of these applications needs to interact with the computer's hardware (open a file, draw something on the screen, get keyboard input, and so on), it makes a request through the operating system. The operating system knows how to talk to the hardware (through *device drivers*) and fills these requests.

Having the OS in the middle means that the applications don't have to worry about the details of the hardware. If an application wants to print, it just asks the OS to do the dirty work. If not for the OS, every application would have to have its own printer drivers and it would be very hard to avoid problems if several tried to print at once.

◈ The role of the operating system and how it does its job are explored further in CMPT 300 (Operating Systems). The interaction between hardware and the OS is discussed briefly in CMPT 250 (Computer Architecture).

Topic 7.4 Disks and Files

When we read and wrote files in Topics 7.1 and 7.2, we took for granted that the operating system could put the information on the disk and get it back later. This is no small job for the OS to do—we should look a little more at what happens.

Note that whenever we're talking about storing information, a *disk* can refer to any device you can use in a computer to store information. The storage must keep the information when the computer is turned off, so the computer's memory doesn't count. These are referred to as *nonvolatile storage.* They include:

- *hard drive*: fast high capacity storage that is used to store the operating system, applications, and most of your data on a computer.
- *floppy disk*: slow low capacity disks that can be easily transported.
- *flash media cards*: small storage devices with no moving parts. These are often used with digital cameras, MP3 players, and other portable devices
- *USB "disks"*: small keychain-sized devices that can be connected directly to a USB port on a computer and then easily transported to another.
- *MP3 player* or *digital camera*: These can often be connected directly to your computer and transfer information to and from built-in storage (or any inserted media cards).
- *compact disc*: used to distribute programs and other information since they are high capacity and cheap to produce. You can't write to compact discs (at least, not quite the same way you can to the other devices listed here).

All of these are treated as "disks" by the operating system. As far as the user and programmer are concerned, they all work the same way. The operating system makes sure they all behave the same way as far as the user or programmer is concerned, regardless of how they work.

Once the operating system knows how to physically store information on all of these "disks", it still has to arrange the information so that it can find it later. When your program asks for a particular file, the computer has to know what information on the disk corresponds to that file; if you change the file, it has to change the information on the disk, adding or removing the space reserved for the file as necessary.

The operating system arranges information on the disk into a *file system.* A file system is a specification of how information is stored on a disk, how to store directories or folders, information about the files (last modified date and access permissions, for example), and any other information that the computer has to store to keep everything working.

The space on the disk is divided up into *disk blocks*. The block size can vary, but is most commonly 4 kB. The blocks are then allocated to the various files that are stored on the disk.

If the filesystem uses 4 kB blocks, a 10 kB file would need three blocks. That means that the file is actually using 12 kB of disk space—the last 2 kB in the last block are wasted. This is refereed to as *internal fragmentation*. Internal fragmentation occurs whenever some of the disk block is left over after the file is written—every file on the disk (unless it *exactly* fills the block) will cause a little internal fragmentation.

When the computer tries to read this file, it will have to read the information from three widely separated blocks on the disk. This will be slow since you have to wait for the read head to move and disk to spin to the right position three times.

Topic 7.5 Example Problem Solving: File Statistics

As an example of using file input, we will create a program that opens up a text file specified by the user and outputs some statistics about it. We want the program to count the number of lines, words, and characters in the file.

The first step will be to get the file name and open the file. This has been done in Figure 7.10. This program also loops through the lines in the file, counting as it goes.

We can now test this program with a sample file, as seen in Figure 7.11. When we run the program in Figure 7.10 and give it this file, the output is:

```
Filename: wc-test.txt
Total lines: 6
```

There are six lines in the input file, so we're on the right track.

Next, we can try to count the number of characters. Since each line in the file is a string, we can just use `len` to get the number of characters in the string and add it to another counter. This has been done in Figure 7.12.

Notice that a `print` statement has been added to the loop to help with debugging. When this program is run on our sample data file, we get this output:

```
# get the filename and open it
filename = input("Filename: ")
file = open(filename, "r")

# initialize the counter
total_lines = 0

for line in file:
    # do the counting
    total_lines += 1

# summary output
print("Total lines:", total_lines)
```

Figure 7.10: Word count: open file and count lines

```
"One trick is to tell them stories that don't go anywhere, like
the time I caught the ferry over to Shelbyville. I needed a new
heel for my shoe, so I decided to go to Morganville, which is
what they called Shelbyville in those days. So I tied an onion
to my belt, which was the style at the time..."
- Abe
```

Figure 7.11: Word count test file

```
# get the filename and open it
filename = input("Filename: ")
file = open(filename, "r")

# initialize the counters
total_lines = 0
total_chars = 0

for line in file:
    # do the counting
    total_lines += 1
    total_chars += len(line)
    print(total_chars)

# summary output
print("Total lines:", total_lines)
print("Total characters:", total_chars)
```

Figure 7.12: Word count: counting characters 1

```
Filename: wc-test.txt
64
128
190
253
303
309
Total lines: 6
Total characters: 309
```

It's not easy to check if this is correct or not. The program probably should be tested on a smaller file, where we can actually count the number of characters by hand to verify. But, the debugging output has given us something to work with.

Look at the last line of Figure 7.11. It contains five characters: dash, space, A, b, e. Why did our program count $309 - 303 = 6$ characters on that line?

Like we saw in Topic 7.2, reading the lines from the file includes the newline characters. But, we don't want to count the "invisible" characters

```
# get the filename and open it
filename = input("Filename: ")
file = open(filename, "r")

# initialize the counters
total_lines = 0
total_chars = 0

for line in file:
    # clean any trailing whitespace off the string
    line = line.rstrip()

    # do the counting
    total_lines += 1
    total_chars += len(line)
    print(total_chars)

# summary output
print("Total lines:", total_lines)
print("Total characters:", total_chars)
```

Figure 7.13: Word count: counting characters 2

in the file. We can use `rstrip` to get rid of these, along with any trailing spaces, before we do the counting. The program in Figure 7.13 does this.

Now when we run the program on the sample file, we get this output:

```
Filename: wc-test.txt
63
126
187
249
296
301
Total lines: 6
Total characters: 301
```

Now we get the right number of characters on the last line. Testing with some other sample files confirms that we are now counting the characters on each line properly.

We can now turn to the problem of counting the number of words on each line. Your first thought might be to just count the number of spaces on the line, but that won't work. If there are several spaces together (the test file has two spaces after each period), then that will count as two "words".

In order to count the number of words in the line, we will to check for *the beginning of each word*. This takes some more careful examination of the string. Since it's more complicated than the other counting we've done, it has been split into a separate function, `words`.

The function `words` characterizes the "start of a word" as a non-space character after by either the start of the string or a space.

See Figure 7.14 for the implementation. Since it's in a function, we can test it separately:

```
>>> words("abc-def ghi")
2
>>> words("abcde f      ghijkl")
3
>>> words("... ,,,    :::")
3
>>> words("")
0
```

The final program is assembled in Figure 7.14. If we run it on our test program, we get this output:

```
Filename: wc-test.txt
Total lines: 6
Total words: 62
Total characters: 303
```

Check-Up Questions

- ▶ Try Figure 7.13 on a text file with a few shorter lines in it. Is it counting the number of characters correctly?
- ▶ Make a copy of Figure 7.14. Print out the number of words after each iteration of the `for line in file` loop. Does it match the number of words you count on each line?
- ▶ In Figure 7.14, the `if` condition in the `words` function checks `pos+1` against `len(line)-1`. What is the purpose of the plus one and minus one?

```
def words(line):
    """
    Count the number of words in the string line.
    """
    words = 0
    for pos in range(len(line)):
        # line[pos] is the start of a word if it is a non-space
        # and it is either the start of string or comes after
        # a space
        if line[pos]!=" " and (pos==0 or line[pos-1]==" "):
            words += 1

    return words

# get the filename and open it
filename = input("Filename: ")
file = open(filename, "r")

# initialize the counters
total_lines = 0
total_chars = 0
total_words = 0

for line in file:
    # clean any trailing whitespace off the string
    line = line.rstrip()

    # do the counting
    total_lines += 1
    total_chars += len(line)
    total_words += words(line)

# summary output
print("Total lines:", total_lines)
print("Total words:", total_words)
print("Total characters:", total_chars)
```

Figure 7.14: Word count: final program

Summary

Working with files gives you a way to “save” things in your program. Information you put in a file can be read back in the next time the program runs.

Hopefully you have an idea of how to read and write simple text files with a Python program. You should also have some idea of what actually happens when a program, one you write or any other, stores information on a disk.

Key Terms

- text file
- file object
- newline character
- operating system
- disk
- file system
- disk block
- internal fragmentation
- external fragmentation

Part III

Appendices

Appendix A

Technical Instructions

Learning Outcomes

This material will help you learn how to use the software you need to do your work in this course. You won't be tested on it.

Learning Activities

- Install the Python software, if you're working with your own computer.
- Follow along with the Python instructions yourself and make sure you can work with the tools.
- Explore the software more on your own.

Topic A.1 Installing Python

We are going to use Python to write and run Python programs in this course. The following tutorial will help you get familiar with some of the functionality of the Python software.

This installation tutorial assumes that you're using Windows. Python is available for the MacOS and for Linux as well. You can use any operating system for your work in this course. You can also use Python in a computer lab on-campus. If you do, Python is already installed and you can skip to the next topic.

You can download the most recent version of Python from the Python web site, http://www.python.org/download/. Click on the link that says: "Download Python 3.x.x" (where 3.x.x is the most recent release). Save this file on your desktop.

Once the file has downloaded, double-click the installation file. You can safely accept all of the defaults for the installation.

You can delete the installation file you downloaded once the installation is complete.

Topic A.2 Using Python

There are two different interfaces where you can write Python code: *IDLE* (Integrated DeveLopment Environment) or the Command Line. We will use IDLE in this course since it provides a graphical interface for you to work with. You can start IDLE by selecting "IDLE (Python GUI)" from the Start menu if you're using Windows.

Note that in this sections, the screen shots are from Windows, but the instructions apply to any operating system.

The usual first program that's written in every programming language is one that prints "Hello World" on the screen. Let's see how we can keep up the old tradition in Python. Open up the IDLE if you haven't already. Your IDLE window will have some text similar to this:

```
Python 3.12.2 (main, Feb 6 2024, 21:26:36)
Type "help", "copyright", "credits" or "license()"
for more information.
>>>
```

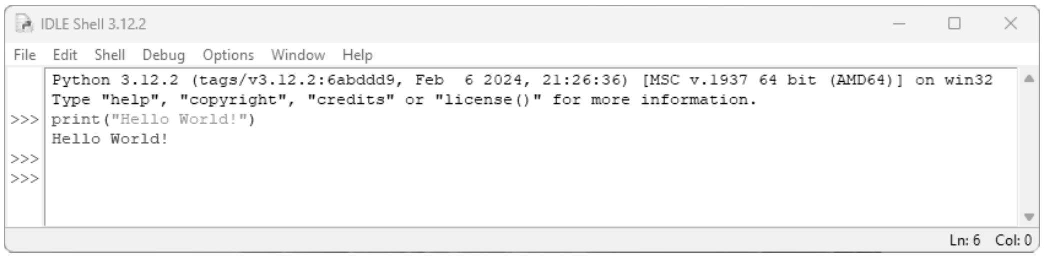
The `>>>` at the bottom is the prompt for writing the statements. When you open up IDLE, your cursor should by default be in front of this prompt. Type in the following statement in IDLE:

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

You have just executed your first Python statement and your IDLE window should look like Figure A.1.

Your first Python program

Typing statements in the IDLE environment isn't the only way to execute Python statements. You can make programs, store them and run them later.



```
    Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
    Hello World!
>>>
>>>
```

Figure A.1: IDLE, after running a "Hello World" statement

These programs are also called script files and are usually saved with a .py extension. Let's make a simple script and run it. Select "New Window" from the File menu in IDLE. An editor window will appear on your screen.

Type the same statement as you did in the IDLE earlier:

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

Select the "Save As..." option from the File menu and save the program as HelloWorld.py. Don't forget the .py extension.

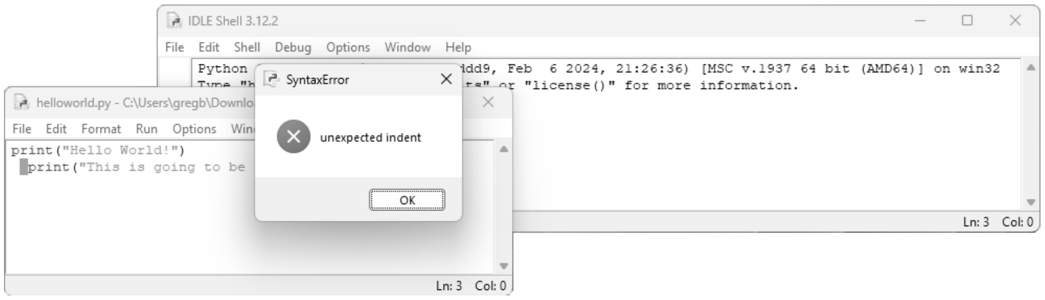
Now run this script file. The easiest way to run and debug your script file is to press F5 while your script file's editor window. The script file should run in the main IDLE window, displaying its output below the other output that previous commands have created.

If you change your script file and try to run it, you will be asked if you want to save your file—you must save the program before it can be run. Change the "Hello World" program that you just made: add the following print statement after the first one:

```
print("This is going to be fun!")
```

Press F5 now. The following window will appear asking you to save the source first. Click on OK and the IDLE will display the out put of your script file.

If there are any syntax errors in your script file, those are identified automatically before the file is run. For example, in Figure A.2 an error message popped up when F5 was pressed. The error here is the incorrect indenting of the second `print` statement—the cursor is moved the the interpreter's best guess at the location of the error. Indentation plays a vital role in Python programming. You will learn more about this as you proceed in the course.

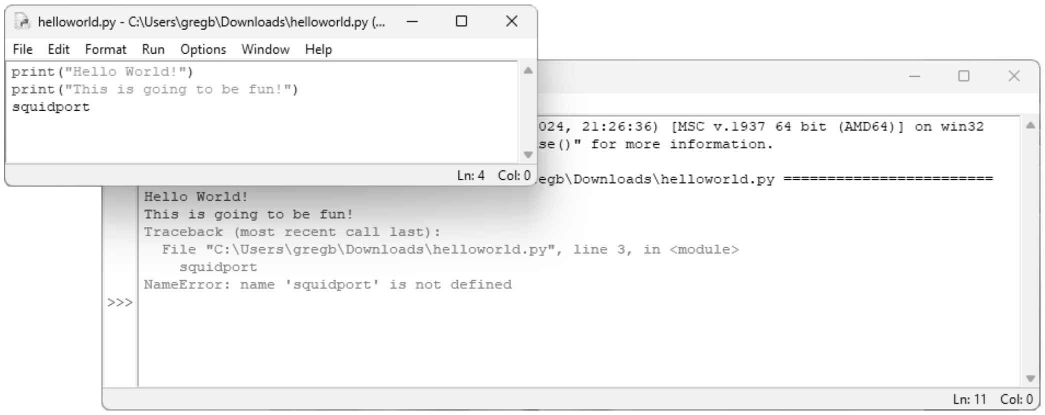

IDLE Shell 3.12.2
File Edit Shell Debug Options Window Help
helloworld.py - C:\Users\gregb\Downlo
File Edit Format Run Options Win
SyntaxError
OK
Ln: 3 Col: 0
Ln: 3 Col: 0

```
Python                    dd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "h                   ts" or "license()" for more information.
```

```
print("Hello World!")
 print("This is going to be
```

unexpected indent

Figure A.2: IDLE, displaying a syntax error


helloworld.py - C:\Users\gregb\Downloads\helloworld.py (...
File Edit Format Run Options Window Help
Ln: 4 Col: 0
Ln: 11 Col: 0

```
print("Hello World!")
print("This is going to be fun!")
squidport
```

```
024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
se()" for more information.

egb\Downloads\helloworld.py ========================
Hello World!
This is going to be fun!
Traceback (most recent call last):
  File "C:\Users\gregb\Downloads\helloworld.py", line 3, in <module>
    squidport
NameError: name 'squidport' is not defined
>>>
```

Figure A.3: IDLE, displaying a run-time error

Once any syntax errors are fixed, the script will run. There might be more errors that the interpreter can't catch until its running the program. For example, in Figure A.3, the interpreter has caught an error. When it got to the word `squidport`, it didn't know what to do with it. It can't catch this any earlier since you might have created a variable or function called `squidport`; the only way to tell for the computer to tell for sure was to run it and see.

The error message indicates that it noticed the error on line 3 of the current program ("`<module>`"). The IDLE editor tells you the line number that the cursor's on in the bottom left corner. In Figure A.3, "Ln: 4" indicates that the cursor is on line 4, just below the error. The "Ln: 11" in the interpreter window isn't what we're interested in: the error message always gives lines in the source file.

◈ Remember: if you want to save the program so you can run it later (or hand it in), it has to go in an editor window, not at the IDLE `>>>` prompt.

Topic A.3 Working in the Lab

All of the software you need in this course is installed in both the ACS labs and the CSIL lab. You can access both and can work in either (or on your own computer is you prefer).

See the course web site for more information about using the labs.

Summary

This material will help you learn how to use the software you need to do your work in this course. You won't be tested on it.

If there are any updates to this material, they will be posted on the course web site or sent by email.

Index