Supplementary study guide for COMP130

John MacCormick, Dickinson College

This study guide contains brief explanations of content that is not fully explained in the assigned reading from the textbook. Anything that is explained in the assigned textbook reading will not be repeated here. When studying, please use the textbook as the primary reference. Use this study guide only for supplementary information that is not in the textbook.

Contents

[1 input() 1](#_Toc208054448)

[2 random.randint() 1](#_Toc208054449)

[2.1 turtle module vs jupyturtle module 1](#_Toc208054450)

[3 for loops 1](#_Toc208054451)

[4 Nested for loops 1](#_Toc208054452)

[5 More flexible versions of range() 1](#_Toc208054453)

[6 Constructors 1](#_Toc208054454)

[7 The graphics module and more constructors 1](#_Toc208054455)

[8 Methods and dot notation 1](#_Toc208054456)

[9 Speeding up the turtle and keeping the window open 1](#_Toc208054457)

[10 Alternative import styles: from and as 1](#_Toc208054458)

[11 Coordinates in graphics.py and the turtle module 1](#_Toc208054459)

[12 Boolean variables 1](#_Toc208054460)

[13 Standard layout and top-level code 1](#_Toc208054461)

[14 Using the debugger in IDLE 1](#_Toc208054462)

[15 Printing on the same line 1](#_Toc208054463)

[16 Factoring out repeated code 1](#_Toc208054464)

[17 Enforcing preconditions via guardians and assert 1](#_Toc208054465)

[18 Extending long lines 1](#_Toc208054466)

[19 Testing with assert 1](#_Toc208054467)

[20 The in operator for strings 1](#_Toc208054468)

[21 String methods, whitespace and special characters 1](#_Toc208054469)

[22 Text files and text editors 1](#_Toc208054470)

[23 CSV files 1](#_Toc208054471)

[24 Map, filter and reduce 1](#_Toc208054472)

# input()

The built-in input() function is covered fully in section 5.11 of the textbook, but we won’t cover that until week seven of the semester, so this supplementary study guide gives a brief explanation here.

The input() function is used to receive input from the user of the computer program while it is running, as in the following example

answer = input('What is your favorite day of the week? ')

The part between the parentheses is called the *prompt*. It’s a string, so it should be in either single or double quotation marks. In the above example, the prompt is 'What is your favorite day of the week? '.

The prompt will be printed. Then, the program waits to receive input from the user. The user is then expected to type their response using the keyboard, terminating the response using the Enter key. The string typed by the user is returned to the program. In the above example, the variable answer will store whatever text was typed by the user.

Here is a complete program demonstrating the use of input():

name = input('Please enter your name. ')

color = input('What is your favorite color? ')

print('Very interesting,', name, '...')

print('I wonder why', color, 'is your favorite color.')

# random.randint()

Facilities for using random numbers in Python are made available by importing the random module:

import random

In the first part of the semester, the only function we need is random.randint(a, b). This function returns a random integer between a and b inclusive. For example, the following program simulates rolling two 6-sided dice.

import random

roll1 = random.randint(1, 6)

roll2 = random.randint(1, 6)

total = roll1 + roll2

print('You rolled a', roll1, 'and a', roll2)

print('That gives a total of', total)

## turtle module vs jupyturtle module

For doing turtle graphics in this course, we will use a turtle module that is **different to the textbook’s turtle graphics module**. The textbook uses jupyturtle (because it is written to work with interactive Jupyter notebooks), but we use the module turtle. The specific changes needed are:

* use import turtle (not import jupyturtle);
* replace other occurrences of jupyturtle with turtle;
* delete the line jupyturtle.make\_turtle(), because a default turtle is automatically created in our turtle module;
* delete other occurrences of make\_turtle().

# for loops

The textbook reading provides a very brief introduction to for loops. Here we provide a more detail. The variable immediately after the for keyword is called the *loop counter*. By default, the loop counter starts counting from zero and increases by 1 each time. For example, the loop counter my\_num counts from 0 up to 5 in the following code fragment:

for my\_num in range(6):

 print('This is the start of the loop body.')

 print('The loop counter is currently', my\_num)

 print('This is the end of the loop body.')

Because the loop counter starts at zero, it counts up to but not including the range parameter. In the above example, the range parameter is 6. The loop is executed six times. The value of the loop counter ranges from 0 to 5 inclusive, which does include exactly six numbers: 0, 1, 2, 3, 4, 5.

It is often useful to store and update information in variables inside the loop body. For example, the following code calculates the value of $0^{2}+1^{2}+2^{2}+…+9^{2}$:

sum\_of\_squares = 0

for n in range(10):

 n\_squared = n\*n

 sum\_of\_squares = sum\_of\_squares + n\_squared

print('Sum is', sum\_of\_squares)

# Nested for loops

It is possible to include a for loop inside another for loop. We call this a *nested* for loop. Within a nested loop, the first for loop is called the *outer* loop and the second for loop is called the *inner* loop. The outer and inner loops should use different loop counters, as in the following example using the outer loop counter i and the inner loop counter j:

for i in range(3):

 print('Outer loop counter i is', i)

 for j in range(2):

 print('Inner loop counter j is', j)

 print('i + j is', i+j)

 print('Outer loop iteration is finishing, and i is still', i)

Note the extra indentation of the inner loop body. Also note that the inner loop body in the above example is executed a total of six times. There are two iterations in the inner loop, but these two iterations are themselves executed three times by the outer loop, for a total of $3×2=6$.

One common application of nested for loops is drawing two-dimensional grids in graphics applications. For example, the following code will draw circles of radius 4 in a rectangular grid spacing of 20 pixels, assuming we have already created a Turtle object t:

for i in range(3):

 x = 20 \* i

 for j in range(5):

 y = 20 \* j

 t.penup()

 t.goto(x, y)

 t.pendown()

 t.circle(4)

# More flexible versions of range()

We already know that an expression like range(5) produces a range of values up to but not including 5, in this case: 0, 1, 2, 3, 4. If we want to start at a value other than 0, we can use an expression like range(3, 7), which produces a range of values beginning at 3, up to but not including 7, in this case: 3, 4, 5, 6. We can also produce a range of values with the step size other than 1. For example, range(12, 29, 5) produces a range of values beginning at 12, up to but not including 29, stepping by 5, in this case: 12, 17, 22, 27.

# Constructors

We already know about datatypes. In computer programming, the word *class* can mean the same thing as *datatype*. Although there are certain technical distinctions between a class and a datatype, we will treat them as the same thing in this course. In computer programming, an *object* is an *instance* of a class. We create an object using a special kind of function called a *constructor*. The name of the constructor is always the same as the name of the class. For example, the following line of code calls the Turtle constructor inside the turtle module, creating an instance of the Turtle class:

my\_turtle = turtle.Turtle()

There are several important things to notice about this line of code:

* Python identifiers are case-sensitive, so turtle and Turtle are completely separate, different entities. In fact, lowercase turtle is the name of a module that can be imported. Uppercase Turtle is the name of a class (or datatype) that is defined in the turtle module.
* When we add parentheses after the word Turtle, it becomes the constructor Turtle() -- a special kind of function that is used to create a new object (or instance) in the Turtle class. Constructors can have parameters, but this one has a zero parameters.
* The variable my\_turtle refers to the new object that was created by the constructor. The datatype of this variable is Turtle.

# The graphics module and more constructors

Other examples of constructors are provided in the graphics module created by John M. Zelle. This module is not built in to Python. You must download the graphics.py file and save it in your current working folder before you can import this module using import graphics. A link is provided on the course webpages.

The following code creates a Point object and a Circle object.

p = graphics.Point(50, 100)

c = graphics.Circle(p, 25)

The variable p refers to an object which is an instance of the Point class, located at $x=50, y=100$. The variable c refers to an object which is an instance of the Circle class. The center of this circle is located at p, and it has radius 25 units. The online documentation of the graphics module explains these constructor parameters in more detail. It also explains the constructors of several other classes such as Line, Rectangle, and Text.

# Methods and dot notation

A *method* is a special kind of function which performs an action on an object. For example, given the my\_turtle object defined earlier, we can use the forward method of the Turtle class to move this object forward 100 pixels:

my\_turtle.forward(100)

Methods are always invoked using *dot notation*, which has the format **object.method(parameters)**.

In Python, dot notation is ambiguous. It can also be used to invoke a *function* inside a module, using the format **module.function(parameters)**. For example random.randint() invokes the randint function from the random module. Another example would be turtle.clear(), which invokes the clear function from the turtle module. It does not perform an action on any specific Turtle object, but clears the entire turtle module graphics system. Contrast this with my\_turtle.forward(100), which invokes the forward method on the specific object my\_turtle.

# Speeding up the turtle and keeping the window open

For experiments with the turtle module, the default animation speed can be too slow. This can be adjusted using:

* turtle.tracer(), a function in the turtle module; and/or
* Turtle.speed(), a method in the Turtle class

You can find out more details of the above function and method in the online documentation of the turtle module. However, for this course, the following information is all you need to know:

* Place the command turtle.tracer(100) near the start of your top-level code
* Place the commands

|  |
| --- |
| turtle.update()turtle.mainloop() |

at the end of your top-level code.

Explanation:

* The command turtle.tracer(100) will make all turtle animations extremely fast, because it will update the screen 100 times less frequently than the default. There is no need to adjust the speed of individual Turtle objects via Turtle.speed().
* We need turtle.update() at the end because some of the screen updates at the end of the program may have been skipped.
* On some Python systems, we need turtle.mainloop() to keep the turtle window open.

# Alternative import styles: from and as

To import a module such as graphics.py, we have been using the command import graphics. An alternative way to import a module is using the from … import \*, as in the following example:

from graphics import \*

The from style of import allows you to use items from the imported module without dot notation. For example, after using from graphics import \*, we can write

window = GraphWin("My graphics window", 300, 300)

instead of

window = graphics.GraphWin("My graphics window", 300, 300)

We will mostly avoid the from style, because it can lead to many names being defined at the top level in your Python program. This is sometimes referred to as *polluting the global namespace*. We want to avoid polluting the global namespace, because a large program can become confusing and unmanageable when many names are defined at the top level. In this course, however, we are mostly writing small programs, and it is acceptable to use the from style if desired.

To save typing, and to enhance the readability of your program, a good alternative is to import a module using an *abbreviation*. We can do this using the import … as … form. For example, we can use

import graphics as gr

Now we can refer to content from the graphics module using the abbreviation gr, as in the following example:

window = gr.GraphWin("My graphics window", 300, 300)

# Coordinates in graphics.py and the turtle module

***For graphics.py:*** In many computer graphics applications, the origin $(0,0)$ is at the **top** left of the screen. In this situation, the $x$ axis points to the right as usual, but the $y$ axis points downwards, so the **positive** $y$**direction is down**. This is the default coordinate system for the graphics module. But it is confusing, so we prefer to avoid it.

To use a more familiar coordinate system in the graphics module, in which the origin is at the **bottom** left, the positive $x$ direction is to the right, and the **positive** $y$**direction is up**, we can use the window.setCoords() function. For example, the following code creates a graphics window of size 300 pixels by 300 pixels and sets up a standard mathematical coordinate system with the origin at the bottom left.

import graphics as gr

window\_size = 300

window = gr.GraphWin("graphics window with origin at bottom left", window\_size, window\_size)

window.setCoords(0, 0, window\_size, window\_size)

***For the turtle module:*** When using the turtle module, the origin is in the **center** of the window. But the directions of the axes are the usual ones, with the positive *y* direction pointing up.

# Boolean variables

From the textbook we are familiar with the data type bool, which can be either True or False. We can use and manipulate variables with this data type too. We call them Boolean variables. The code below demonstrates this, using the two Boolean variables happy and sad.

happy = True

sad = not happy

if happy:

 print("I'm happy")

else:

 print("I'm not happy")

if sad:

 print("I'm sad")

else:

 print("I'm not sad")

print('The statement "I am happy" is', happy)

print('The statement "I am not happy" is', not happy)

print('The statement "I am sad" is', sad)

print('The statement "I am not sad" is', not sad)

We can use Boolean variables to build up more complex Boolean expressions, as in the if statement below.

excited = False

if happy and excited:

 print('What a fantastic day!')

elif happy:

 print("This is a good day. I'm happy but not excited.")

We can also use the bool data type in parameters:

def umbrella\_advice(is\_raining):

 if is\_raining:

 print('Take your umbrella')

 else:

 print('No need for an umbrella right now.')

umbrella\_advice(True)

# Standard layout and top-level code

From this point onwards in the course, all Python files will employ a *standard layout*. All files submitted for labs and lab exams should employ the standard layout. The standard layout of a Python file consists of four sections:

1. Imports
2. Top-level setup code
3. Function definitions
4. Top-level main code

In Python, *top-level code* is code that is outside of function definitions. We think of it as being at the “top level” because it will be executed without having to call any functions. In the standard layout of a Python file, sections 1, 2, and 4 consist of top-level code; whereas section 3 consists of function definitions (which are not top-level code). Here is an example:

# Section 1: Imports

import math

import turtle

# Section 2: Top-level setup code

t = turtle.Turtle()

t.speed(0)

# Section 3: Function definitions

def draw\_face(x, y, radius, color):

    ...

def draw\_body(x, y, height):

    ...

# Section 4: Top-level main code

draw\_body(-50, 20, 100)

draw\_body(150, -10, 80)

Section 2, the top-level setup code, should be used only to define objects and settings that are needed throughout the file. It will often be empty. Section 4, the top-level main code, is the place where your program will begin executing the code that does useful work, by calling functions that were defined earlier. In this sense, it is the “main” part of the program.

Sections 2 and 4 -- the top-level setup code and top-level main code -- should be kept as brief as possible. This makes it easier to maintain and reuse your code. (The detailed reasons for this will become clearer once you have additional software engineering experience.) Remember this as the following slogan: **avoid too much top-level code**. If you have more than a few lines of top-level code, define a new function and move most of your top-level code into the new function.

# Using the debugger in IDLE

There are many ways to use the debugger in IDLE. The following instructions describe one possible method.

1. In the IDLE Shell window, go to Debug menu and choose Debugger.
2. Check the Source box in the Debug Control window.
3. Go to the window containing your Python source code. Turn on line numbers. Right-click on the line where you would like the program to freeze, and choose Set Breakpoint.



1. Run the program as usual, via Run Module in the Run menu. The program will freeze at the first line of code. In the Debug Control window, click Go.



1. The program will run until it hits the breakpoint that you set previously. Note the local variables listed in the lower pane. Variables in the other stack frames can be viewed by double clicking on different parts of the stack, in the upper pane.



1. Experiment on your own with Step, Over, and Out. The basic definitions are:
* **Step** executes one line of code (and steps inside any function on the current line).
* **Over** executes one line of code (but steps over any function on the current line).
* **Out** steps out of the current function by finishing the remaining lines in the current function and returning to the calling function, one level up in the call stack.

# Printing on the same line

By default, the print function adds a special character, called a *newline*, to its output. The output of the next print function will then appear on a new line. If you wish the output to appear on the same line, you can set the end parameter of print to the empty string:

print('hi ', end='')

print('how ', end='')

print('are ', end='')

print('you')

The above code prints 'hi how are you', all on the same line.

# Factoring out repeated code

The textbook section 4.5 explains the general concept of *refactoring*. Here we explain a special type of refactoring, known as *factoring out repeated code*. We do this when some code is repeated in two or more locations, as in the following example.

def roll\_sixsided\_dice(num\_dice):

 total = 0

 for i in range(num\_dice):

 roll = random.randint(1, 6)

 total = total + roll

 print('Dice total is', total)

def roll\_eightsided\_dice(num\_dice):

 total = 0

 for i in range(num\_dice):

 roll = random.randint(1, 8)

 total = total + roll

 print('Dice total is', total)

The repeated code could be identical, or it may contain slight differences. In the above example, the repeated code is highlighted in gray, and some slight differences are highlighted in yellow.

We factor out the repeated code by copying it into a new function. Sometimes, we need to add parameters to the new function. In the following example, we use the variable num\_dice as one parameter and we generalize by incorporating the new parameter num\_sides. The new parameter is needed to allow for the slight differences in the repeated code highlighted above.

def roll\_dice(num\_dice, num\_sides):

 total = 0

 for i in range(num\_dice):

 roll = random.randint(1, num\_sides)

 total = total + roll

 print('Dice total is', total)

Finally, we eliminate the original repeated code, replacing it with invocations of the new function:

def roll\_sixsided\_dice(num\_dice):

 roll\_dice(num\_dice, 6)

def roll\_eightsided\_dice(num\_dice):

 roll\_dice(num\_dice, 8)

There are two reasons that we should factor out repeated code. First, the code is more *maintainable*: if we need to make changes, it is quicker and less error-prone to change the single function that was factored out, compared to changing both versions of the repeated code. Second, the code is usually easier to read and understand after repeated code has been factored out.

When refactoring code: the functionality of the code must remain the same (so it produces exactly the same outputs); and the interface of the code must remain the same (so the headers of existing functions cannot be altered in any way). Recall that the first line of a function definition is called the *header*; this is also known as the *signature* of the function. Therefore, when refactoring code, we cannot change the signatures of existing functions.

# Enforcing preconditions via guardians and assert

Section 4.9 of the textbook mentions *preconditions*. A precondition is a requirement that should be true before a function can execute. There are many ways of enforcing preconditions. In this course, we use a simple approach that employs Python’s assert statement. The assert statement tests if a given condition is true, then immediately halts the program if the condition is false. When we use the assert statement to enforce a precondition, we refer to the statement as a *guardian*. This is because it guards the function from violations of the preconditions. Consider the print\_course\_code function below. The subject parameter is intended to be a string such as 'comp' or 'math' or 'data', corresponding to one of the subjects in Dickinson’s course catalog. The number parameter should be an integer course number such as 130 or 232. The three guardian statements ensure that the parameters are reasonable before the rest of the function executes.

def print\_course\_code(subject, number):

 assert isinstance(subject, str)

 assert isinstance(number, int)

 assert number>=100 and number<600

 print('Course code is', subject.upper() + str(number))

The builtin Python function isinstance is especially useful in guardians. We use it to check that the datatype of a parameter is correct.

We can optionally add a meaningful error message to a guardian statement, as shown in the following example.

def print\_course\_code(subject, number):

 assert isinstance(subject, str), 'subject must be a string'

 assert isinstance(number, int), 'number must be an integer'

 assert number>=100 and number<600, 'course number outside expected range'

 print('Course code is', subject.upper() + str(number))

# Extending long lines

When a single line of Python code is too long to fit on a screen, you can break it into multiple lines using a backslash character, as in the following example.

if (apple > 6 or banana < 3) and (banana == 10 or cherry < 7) and \

 not (donut <= 3 or eggplant == 7 or falafel == 9) \

 and (apple < 7 or banana < 5):

 print('hi')

# Testing with assert

In computer programming, it is essential to test all code for correctness. There are many ways to write tests. In this course, we test a Python function f by writing a *test function* test\_f. The test function uses assertions to check that the values returned by f are correct, then prints a message that the test was successful. We should check a variety of possible values to be confident that f contains no errors. At a minimum, the test function should achieve *statement coverage* for f, which means that the test function causes every line in f to be executed at least once. As an example, consider the following function.

def icecream\_cost(num\_scoops):

 """Return the cost of an icecream in cents, based on the number of scoops. One scoop

 costs $4.50. Two scoops cost $6. Any other number of scoops is invalid and the

 return value is -1."""

 if num\_scoops == 1:

 return 450 # $4.50 in cents

 elif num\_scoops == 2:

 return 600 # $6.00 in cents

 else: # invalid number of scoops

 return -1

Here is a suitable test function for the function above.

def test\_icecream\_cost():

 assert icecream\_cost(1) == 450

 assert icecream\_cost(2) == 600

 assert icecream\_cost(3) == -1

 assert icecream\_cost(-7) == -1

 print('test\_icecream\_cost succeeded')

To achieve statement coverage, the above test needed to use a minimum of three different assertions to cover the three different paths through the conditional statement. A fourth assertion is used to test for negative inputs. There are no rigid rules about how many assertions to use, but it is a good idea to include different kinds of inputs (such as positive and negative numbers) and to test for *edge cases* (that is, values on the boundary of a particular behavior). The value 3 is an example of an edge case above, because this is the smallest number of scoops that is invalid.

As we know from our earlier use of assert, it is also possible to add an explanatory message to each assertion:

def test\_icecream\_cost2():

 assert icecream\_cost(1) == 450, 'failed with single scoop'

 assert icecream\_cost(2) == 600, 'failed with two scoops'

 assert icecream\_cost(3) == -1, 'failed with three scoops, which should be invalid'

 assert icecream\_cost(-7) == -1, 'failed with negative number of scoops'

 print('test\_icecream\_cost2 succeeded')

Because there can be a tiny amount of round-off error when using the float datatype, we need to allow for small differences when testing the value of floating point numbers. In the test below, we define a small value eps as 0.000001 (one millionth, or $10^{-6}$), which can also be written as 1e-6 in Python. The name eps is short for “epsilon”, the name of the Greek letter $ε$, which is often used to represent very small quantities. We can test whether the expected answer is within $10^{-6}$ of the calculated answer using the built-in absolute value function, abs.

def add\_1000\_times(increment):

 """Add the given increment 1000 times and return the result."""

 total = 0.0

 for i in range(1000):

 total = total + increment

 return total

def test\_add\_1000\_times():

 eps = 1e-6

 assert abs(add\_1000\_times(0.1) - 100.0) < eps

 assert abs(add\_1000\_times(4.567) - 4567) < eps

 print('test\_add\_1000\_times succeeded')

Note that the calculated values do have some round-off error here: add\_1000\_times(0.1) returns 99.9999999999986 and add\_1000\_times(4.567) returns 4567.000000000006.

# The in operator for strings

The Python keyword in tests whether one string is a substring of another string. For example:

* 's' in 'eggs' is True,
* 'g' in 'eggs' is True,
* 'gg' in 'eggs' is True,
* 'egg' in 'eggs' is True,
* 'ggg' in 'eggs' is False.

Note that the keyword in has other uses. We have already seen it used with for loops, such as for i in range(5). We can also iterate over the characters in a string, using code such as for c in 'xyz'. The keyword in can also be used with more advanced datatypes that we will study later, such as lists. The following example demonstrates two different uses of in.

def count\_upper\_case(s):

 upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

 count = 0

 for c in s:

 if c in upper:

 count = count + 1

 return count

# String methods, whitespace and special characters

Here are the string methods that you should be familiar with by the end of topic 8:

count, endswith, find, isalpha, isdigit, lower, lstrip, replace, rfind, rstrip, startswith, strip, upper

Some of the string methods require an understanding of Python lists. We won’t study lists until topic 10. Once we have completed topic 10, you should also be familiar with the following string methods:

join, split, splitlines

When manipulating strings, it’s important to understand the notion of *whitespace*. Whitespace characters are characters that create space within a string but are not visible. These include the newline character, the tab character and the space character. String methods that use whitespace include strip and split. By default, strip removes whitespace from the start and end of the string. By default, split will split a string using whitespace as the separator.

Some special characters can be typed into a string literal using special notation. For example, \n represents a newline character, \t represents a tab character, \' represents a single quote, \" represents a double quote, and \\ represents a backslash character.

# Text files and text editors

Within a computer’s file system, a *text file* is a file that contains characters with no formatting. There is no concept of color, size, or font. The layout is determined only by whitespace characters. Python files are text files. Files that end in .txt or .csv are usually text files too, and there are many other examples.

A *text editor* is a program that you can use to edit text files. IDLE is actually a text editor, with some additional features that make it easy to edit and run Python programs. There are many other text editors, including Notepad (on Windows computers) and TextEdit (on Apple computers). To use IDLE as a text editor for files that are not Python files, look for a dropdown menu to enable other file types in addition to \*.py when using the Open or Save As menu options. Depending on your operating system, the drop-down menu may look like this:



# CSV files

A CSV file such as courses.csv is a text file that contains data in rows and columns. Each row is a single line of the file, and the columns are separated by commas. The acronym CSV stands for *comma-separated values*. For example, the courses.csv file might contain the following text:

course,instructor,enrollment

COMP130,John MacCormick,23

COMP232,Farhan Siddiqui,18

COMP356,Matt Ferland,21

COMP190,William Goble,15

Once we have studied lists, we will be able to read and write CSV files easily using Python programs. Using our current knowledge, however, we can already write certain CSV files. For example, the following program creates a CSV file that stores the sums of some random numbers.

fout = open('random\_sums.csv', 'w')

fout.write('x,y,x+y\n')

for i in range(10):

 x = random.randint(1,100)

 y = random.randint(1,100)

 sum = x + y

 fout.write(str(x)+','+str(y)+','+str(sum)+'\n')

fout.close()

# Map, filter and reduce

*[The text in this section is copied verbatim from section 10.7 of the second edition of* Think Python*, by Allen B. Downey. We are using the third edition for this course, but the third edition does not cover this required material. This text was originally published under an Attribution-NonCommercial-ShareAlike 4.0 International license, and the verbatim copy here is made available under the same license.]*

To add up all the numbers in a list, you can use a loop like this:

def add\_all(t):

 total = 0

 for x in t:

 total += x

 return total

total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable. This augmented assignment statement,

 total += x

is equivalent to

 total = total + x

As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an *accumulator*.

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

>>> t = [1, 2, 3]

>>> sum(t)

6

An operation like this that combines a sequence of elements into a single value is sometimes called *reduce*.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

def capitalize\_all(t):

 res = []

 for s in t:

 res.append(s.capitalize())

 return res

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.

An operation like capitalize\_all is sometimes called a *map* because it “maps” a function (in this case the method capitalize) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

def only\_upper(t):

 res = []

 for s in t:

 if s.isupper():

 res.append(s)

 return res

isupper is a string method that returns True if the string contains only uppercase letters.

An operation like only\_upper is called a *filter* because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of map, filter and reduce.