CS 105 – Introduction to Computer Science

The purpose of this handout is to guide you through the first unit of the course, which deals with computer problem solving. This is your reading material to prepare for class. As with any course you take, don’t fall into the trap of doing nothing during the early weeks of the course. The material will accumulate, and you need to apply your best study habits right now from the beginning. Don’t wait. A month from now … you will be glad (or you will have wished) you started today.

Preliminary Remarks

In essence, computer science (CS) is the study of how to solve problems. Also, we are interested in how to represent various forms of information. Computer scientists study various ways to solve problems and the nature of problems themselves. We classify problems, and seek ways to solve them systematically. As you might expect, the types of problems addressed in CS are usually quantitative in nature. However, the information that our problems deal with are not always numbers. Information may include text, images, sounds and video.

You might have noticed that my definition of CS above did not involve the word “computer.” The computer is merely a general-purpose machine that assists us in solving problems much more rapidly and reliably than just working manually. So, this course is about how to think, and how to solve problems, especially using a computer.

What does it mean to “solve a problem” in this course?

For example, here’s a problem: “Calculate a grade point average.” The solution to this problem is not just a number like 2.4. This value may or may not be my GPA or somebody else’s. Instead, the way to solve a problem is to provide the necessary steps to get to a solution. This way, you can obtain the grade point average for anyone, not just for one person.

Throughout this course, it may help to think about cooking as an analogy. After all, every day we personally have the problem of “I am hungry.” A cookbook is not just a bunch of pretty pictures of food. It also gives you the instructions you need to follow to create dishes.

This is an important point, so it bears repeating: To solve a problem, we need to write down the steps necessary to get us to an answer. When we are done, the solution to a problem is called an “algorithm.”

How do we know if we are right?

If we solved a problem correctly, this means that we will always get the right answer when we follow the steps. In other words, when we work through the steps, either by ourselves or on the computer, and our final result is wrong, then this means that there is a mistake in our solution. (Of course, it’s possible that we followed the steps incorrectly, but the computer won’t make that kind of mistake!)
Since the solution to a problem is not simply a single word or number but rather a list of instructions, we should expect the problem-solving process to take a while. So, we need to be patient with ourselves, and have faith in our abilities.

I’d like to take this opportunity to reiterate some guiding principles for learning.

1. We are here to learn and explore.
   a. Seek discussions with the instructor and classmates about the material to reinforce your understanding and practice communicating ideas.
   b. Have fun. Live in the moment (i.e. don’t dwell too much on the difficulties of yesterday or tomorrow). Enjoy the journey and intellectual feast. Be enthusiastic about what you are doing.

2. You can be successful in this class. Every day is an opportunity for an epiphany. Don’t let mistakes or setbacks hold you back. After some effort, things can suddenly click in your mind.

3. Learn by doing!
   Not just passively reading, listening or watching.
   Each study period needs to have a clear goal.
   Pay attention to the big picture and the facts that you are collecting.

4. Be organized: Take notes on what you read. Review earlier material as needed. Create a cumulative study outline, and update it each week. Maintain a portfolio of your work.

5. Be patient when solving a homework or lab problem.
   a. There is no need to rush.
      Don’t worry if your first attempt at a solution is wrong.
      Read all instructions and be methodical.
      Take time to gather your thoughts.
      Deliberately write out your thought process and plan of attack.
   b. A computer program or other homework assignment may take up to several hours to complete.
      In programs you need to comment your code as you go, because you will quickly forget what looks obvious right now! Realize that you don’t need to finish everything in one sitting.
   c. Break up large problems into small, more manageable pieces.
   d. Don’t get bogged down with too many mechanical details. Computing is all about removing tedium from routine tasks.

6. Be curious, and always ask questions.
   a. Find a topic or application that you are enthusiastic about.
   b. Consider alternative solutions to a problem.
c. When finishing a problem, ask yourself if this problem or its solution lends itself to other problems.

7. Computer science is about logic, structured thinking, information, communication and problem solving. Thus, it has connections to many other fields in the sciences, humanities and social sciences. You will find the analytical techniques useful in your career.

In every course you take, including this one, it is sometimes necessary to answer a “what is” question as you study. Depending on the application (e.g. cheetah, sautéing, freedom, Canada, airplane, computer science), here are some ways you can approach such a question.

- What it consists of, or its properties
- What it resembles, is analogous to
- How it behaves
- How life would be different without it
- What it is NOT
- Its purpose

In this course we will study two things:

- How to solve problems in general using the computer
- How to apply these principles to a particular problem domain

Discussion:

1. If you had to describe the computer with one word, what would it be?

2. What is the role of the computer in computer science?

3. Find a cash register receipt from a recent trip to the store. How would you describe or characterize the information contained in it? What information on it is unimportant to you? Give an example of a labeled value on the receipt. What is the label, and what is the value?

4. Since you woke up this morning, what sort of “problems” did you need to work out in your mind? Did you follow a sequence of steps that you figured out some time ago?

5. As suggested above, we can practice a “what is” type of question. For a given noun (e.g. cheetah), decide which strategies for answering a “what is” question are most appropriate.
BINARY NUMBERS

Let’s begin with some basics: how information is represented inside the computer.

Information includes such things as: numbers, text, sounds, images and video. All information must be represented as 0s and 1s when stored inside the computer. The word binary refers to a number system using only 0s and 1s. Binary is the computer’s native language.

A bit is the smallest unit of information. It is a single 0 or a single 1. The word “bit” is short for “binary digit.”

The basic building block of computer hardware, such as the Central Processing Unit (CPU), is the logic gate. The purpose of logic gates is to manipulate individual bit values. They can do so very quickly, in less than a billionth of a second. Various logic gates are arranged to allow the hardware to perform mathematical operations.

Since single bits are not very interesting, we tend to group them so that they can store a useful amount of information. The most common way to group bits is to arrange them in units of eight. A byte equals 8 bits of information. The byte is the fundamental unit of measuring amounts of memory or information. One byte is usually enough space to hold a single letter of text or a small number.

There are two essential skills that you need to learn:

- Interpreting binary: converting a binary number into decimal
- Encoding into binary: converting a decimal number into binary.

Just to clarify, the word “decimal” here means “base 10”. The numbers that we will use in this course will be non-negative integers. In future courses, you will see how negative numbers and real numbers are represented in binary.

Converting binary to decimal

How do binary numbers work?

The first thing to understand about binary numbers is that this system is base 2, rather than the base 10 (decimal) that we humans use. We write out a binary number using the same place-value system as in decimal. Let’s review what the place-value system means with this example:

We understand that when we see the number 278 that this means (2 * 100) + (7 * 10) + (8 * 1). Each digit of the number 278 has to be multiplied by the appropriate power of 10. Binary works exactly the same way except for two details:

- Each digit is either a 0 or a 1; no other symbols can be used.
- Digits are multiplied by powers of 2, not powers of 10.
For example, let’s look at the binary numbers 001110 and 100011:

<table>
<thead>
<tr>
<th>32 times</th>
<th>16 times</th>
<th>8 times</th>
<th>4 times</th>
<th>2 times</th>
<th>1 times</th>
<th>Number value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>= 14</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>= 35</td>
</tr>
</tbody>
</table>

Notice that each digit of a binary number is multiplied by a power of 2. The rightmost digit is multiplied by 1, the digit to its left is multiplied by 2, etc. The total value of a binary number is the sum of the values we get from each digit.

At this point, it would be helpful to take a look at some powers of 2.

\[
2^0 = 1 \\
2^1 = 2 \\
2^2 = 4 \\
2^3 = 8 \\
2^4 = 16
\]

You should memorize these powers of 2, because they occur very frequently. Actually, there is very little here that needs to be memorized. At a bare minimum, you should know that \(2^0 = 1\). Any of the other powers of 2 can be obtained by repeatedly doubling until you reached the desired power.

In particular, you don’t need to memorize \(2^5\) and \(2^6\). You can figure them out just by looking at \(2^4\) and doubling as needed. What are the results?

Sometimes it is very handy to work with much larger powers of 2. But rather than worrying about their exact values, it is often helpful to estimate them. You should memorize these approximations:

\[
2^{10} \text{ is about 1 thousand} \\
2^{20} \text{ is about 1 million} \\
2^{30} \text{ is about 1 billion}
\]

If you combine the small exact powers of two with these approximations, you can quickly come up with approximate values of many powers of 2. But you will need to use a mathematical fact that you learned in basic algebra: \(a^{bc} = (a^b)(a^c)\). For example, suppose we wanted to approximate \(2^{25}\). The number 25 may be written as 25 = 5 + 20. Therefore, \(2^{25} = (2^5)(2^{20})\). From our earlier work, we know that \(2^5\) is 32 and \(2^{20}\) is about 1 million. Thus, \(2^{25}\) should be approximately 32 million.
Discussion:

1. Estimate the values of $2^{16}$ and $2^{34}$.

2. You can also go the other way: which powers of two are near the following values: 4 billion, 128 thousand, 2 million?

3. Let’s say we have 4 bits. What is the lowest number we can have? What is the highest number we can have?

4. Repeat the previous question assuming we have 5 bits.

5. Is there a pattern going on here? You should be able to write a formula for the largest possible that can be written with $n$ bits.

Converting decimal to binary

It has been said that, “There are 10 kinds of people in the world. Those who understand binary, and those who don’t!”

One thing to note is that binary numbers are longer than decimal. It takes more binary digits to express a number than base 10 can. For example, a 5-digit decimal number may turn out to need 15 bits! In general, binary numbers are about 3 times the length of decimal numbers.

Converting decimal into binary is slightly more complicated than going the other way. My technique for converting decimal numbers into binary is to imagine shopping at a very special store that I call the “Binary Store.” All merchandise in this store is priced at a power-of-2 number of dollars: $1, $2, $4, $8, $16, $32, etc.

You enter the store with a certain number of dollars, say $45. As you shop, your goal is to always purchase the most expensive gift possible. And you must spend all of your money. In the case of $45, the first item we can buy is $32. We subtract in order to figure out how much money we have left: $45 – $32 = 13. Next, we find an $8 item, so that $13 – $8 = 5. With $5 remaining we see that we can buy a $4 item and a $1 item. When we check out of the store and look at our receipt, we see that we bought items costing $32, $8, $4 and $1. In other words, mathematically we have $45 = 32 + 8 + 4 + 1$. Note that we did not purchase the $16 item or the $2 item. Some items got skipped based on what money we had to start with.

To write down our binary answer, we just fill in this table:

<table>
<thead>
<tr>
<th>32 *</th>
<th>16 *</th>
<th>8 *</th>
<th>4 *</th>
<th>2 *</th>
<th>1 *</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Once we know that $45 = 32 + 8 + 4 + 1$, we write down all of the powers of 2 from 32 down to 1. Underneath each power of 2, we write a 0 or a 1. Write down “1” if we bought that item, and write “0” if we did not. In this case, we bought the items costing 32, 8, 4 and 1 dollars, so those powers of 2 get 1s. But we did not buy the items costing 16 or 2 dollars, so those powers of 2 get 0s. Our final answer is to say that 45 in decimal is equivalent to 101101 in binary.

Let’s try another example. Convert 61 into binary.

When we go to the binary store, we see that we can buy items costing 32, 16, 8, 4 and 1 dollars. In other words, $61 = 32 + 16 + 8 + 4 + 1$. Once again, we need to write out the powers of 2, this time from 1 on the right end to 32 on the left end. Write a “1” underneath each power of 2 that we actually bought.

<table>
<thead>
<tr>
<th>Numbered position</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power of 2</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
<tr>
<td>Value of power of 2</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Did we buy it?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Our binary answer is 111101. Incidentally, we could have written our powers of 2 using exponents, like this: $61 = 2^5 + 2^4 + 2^3 + 2^2 + 2^0$. The exponents on 2 tell us where to write the 1s in our binary answer. A more thorough table explaining everything that is going on as we convert 61 into binary would look like this:

As with any skill, you should pause and try more examples yourself. Think of a random decimal number, and convert it to binary. You can use the earlier technique of interpreting binary numbers to check your answers. I recommend that you try both odd numbers and even numbers. Along the way, you should note the following rule of thumb:

- If a number is even, the rightmost bit will be 0
- If a number is odd, the rightmost bit will be 1

This rule can give you a quick check of your answer. For example, if you tried to write the binary representation of 38, but you see that the rightmost bit is 1, you know that something is wrong.

Octal shorthand

Writing down a binary number can be tedious. So, it is convenient to write a shorthand notation. Once such notation is called octal, which means base 8. Here, each digit represents a power of 8, rather than
a power of 2. There are 8 possible symbols in octal: 0, 1, 2, 3, 4, 5, 6, 7. The digits “8” and “9” are not possible. Because \(8 = 2^3\), each octal digit will correspond to 3 bits in our number.

Converting between binary and octal is pretty simple. The key is to group the binary into clumps of 3 bits at a time. For example:

Octal 671 = 110 111 001 in binary: because 6 = 110, 7 = 111 and 1 = 001

For your convenience, here is a table showing the correspondence between octal digits and their corresponding binary codes. Do not memorize this table. You should be able to perform the conversions yourself.

<table>
<thead>
<tr>
<th>Octal digit</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

Try one yourself: Convert octal 7325 to binary. You should be able to work this out without using the table above as a crutch. First, how many total bits will you need?

Let’s go the other way: binary to octal. Group the bits in threes. If there do not seem to be “enough” bits, then pad with extra zeros on the left.

For example, consider the binary 101011100. There are 9 bits here. 9/3 = 3, so we should expect 3 octal digits in our answer. Space out each clump of 3 bits, and convert each clump to its own octal digit.

101011100 binary =

101  011  100 binary =

5  3  4  octal.

Our answer is 534.

Unlike converting between binary and decimal, converting between binary and octal can be done pretty fast no matter how big the number is.
Hexadecimal shorthand

It turns out that there is a more commonly used shorthand notation called hexadecimal, or just “hex” for short. The word hexadecimal literally means “base 16”. It is based on the idea that $16 = 2^4$. Hexadecimal works just like octal, except that each hex digit represents 4 bits rather than 3.

Hexadecimal also means that we have 16 different symbols for each digit. We may use the conventional digit values 0-9, but we need six more. They are:

a = 10,  b = 11,  c = 12,  d = 13,  e = 14,  f = 15

It may look odd at first, but the letters a-f are in fact used to represent digit values in hexadecimal. It does not matter if you use capital or lowercase letters as long as you are consistent. In practice, lowercase letters are common.

Again, for your convenience, here is a table listing all the hexadecimal digits along with their corresponding binary codes:

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>a</td>
<td>1010</td>
</tr>
<tr>
<td>b</td>
<td>1011</td>
</tr>
<tr>
<td>c</td>
<td>1100</td>
</tr>
<tr>
<td>d</td>
<td>1101</td>
</tr>
<tr>
<td>e</td>
<td>1110</td>
</tr>
<tr>
<td>f</td>
<td>1111</td>
</tr>
</tbody>
</table>

Let’s practice converting between binary and hexadecimal.

964 in hex = 1001 0110 0100 in binary, because 9 = 1001, 6 = 0110, and 4 = 0100
d123 in hex = 1101 0001 0010 0011 in binary, because d (13) = 1101, 1 = 0001, 2 = 0010, and 3 = 0011.

Practice yourself:

1. Convert the binary number 111000 to hexadecimal.

2. Convert the binary number 10011111 to hexadecimal.
3. The above hexadecimal table shows the hex and binary representations of the numbers 0-15. What are the hex and binary representations of the numbers 16, 17, and 18?

A note about notations: When we write down a number, and we want to avoid confusion as to what base it is in, it is customary to write the base as a subscript after a number. Therefore, $100_8$ means the octal number 100, rather than “one hundred” in decimal or the binary equivalent of 4.

However, when typing, entering a subscript can be awkward. In computer science, it is customary to write a leading zero at the beginning of a number to signify that it is octal. For example, rather than typing “octal 671” or “6718”, it is often easier to type 0671. And for hexadecimal, the usual convention is to precede the number with 0x. For example, the notation 0xd123 means the hexadecimal number d123.

Conversions between decimal and octal/hex

Often, the best way to come up with octal or hexadecimal representations is to go through binary first. It becomes a two-step process: convert from decimal to binary, and then from binary to hex/octal.

For example, how is the decimal number 71 written in octal?

- The Binary Store gives us: $71 = 64 + 4 + 2 + 1$.
- Therefore, our binary number is 1000111.
- Grouped as octal, we have 001 000 111 = 1078.

Here is an exercise you can try yourself:

1. How are the decimal numbers 8, 9, 10 and 16 represented in octal?

2. How are the decimal numbers 16, 17 and 32 represented in hexadecimal?

Here is a little joke: Why couldn’t the computer scientist tell the difference between Halloween (Oct 31) and Christmas (Dec 25)?
A FIRST LOOK AT COMPUTER PROGRAMS

Objectives:

- Define a computer program, and describe what it looks like
- Identify certain things that we expect to find in computer programs, such as comments, statements that perform input, calculations, and output
- How to run a program

Why are we interested in creating our own software? The main reason is that we want to harness the computational power of the computer, to have direct control of the machine. Sometimes, existing software that is installed on the computer is not sufficient for us. It doesn’t do exactly what we want. Writing our own programs can be extremely useful and even fun for people to use, such as a game, or converting data to an image.

In order for us to create software, we need to learn a computer programming language. There are thousands of programming languages, but only a few have become especially famous in computer science, such as Java, PHP, Python and C++. These languages are machine independent. This means that if you create a program in such a language, it should be able to run on any type of machine. Programming languages have been around for about 60 years, and the modern ones used today have many built-in features that simplify coding. Many common tasks and calculations are pre-defined as part of the language, such as sorting data, opening files, surfing the Web, creating a graphical user interface, etc.

A computer program is a list of instructions written in a computer language that solves some problem. In this course, we will use the Python language to write our programs. Python is a computer programming language. The text that makes up a computer program is called source code, or simply “code” for short. Note that this use of the word code is a mass noun, similar to the word money. In other words, when talking about source code, we always say code in the singular.

When we get into lab, a certain routine is going to emerge. Here are some of the steps.

1. A program is submitted to the computer.
2. We run the program.
3. As the program is running, it may ask for input from you.
4. When the program is finished, we look at its output.

To gain confidence at writing short computer programs, we need to get into the habit of reading some computer code. Study carefully the code examples in this short book and ones we do in class. Eventually, you should be able to read several lines of code, and have a precise idea in your mind of exactly what it is doing.

There are a few reasons why learning a programming language like Python will be much easier than learning a natural language like French:
• There are very few words to learn.
• Those words are already in English.
• The grammatical rules are very simple.

So, the challenge of computer problem solving is not learning the language. It is that writing a computer program forces us to think in a way that is logical, organized, and precise – more so than in everyday English.

What does a computer program look like? In essence it works like a recipe: it must list all of the necessary ingredients (data) and instructions (steps) that the computer needs to obey. Cooking may be a good analogy, because it solves an important problem: “I’m hungry!” 😊 What do we generally see in recipes? Well, here is one:

• Brown the beef for 15 minutes. Drain the grease.
• Dice carrot, celery, and onion (otherwise known as mirepoix)
• Cut up and boil 6 potatoes until soft.
• Mash the potatoes.
• Add flour, spices, sauce, and mirepoix to the beef.
• Put the meat mixture into a casserole. Top with potatoes.
• Bake in the oven at 400 degrees for 30 minutes.

Discussion:

1. What dish is this recipe preparing? (In other words, what is the output?) How can you tell?
2. List all the ingredients (input) of the recipe. Why is it important to know all of the ingredients before you start the recipe?
3. Which steps in the recipe imply that we need to continue or wait for something, or do some process repeatedly?
4. At what point(s) in the recipe would it be necessary for a cook to check the progress of a step or to make a decision? What are the decisions or judgments that need to be made?

A simple program will have these elements.

• Comment(s)
• Input statement(s)
• Calculation statement(s)
• Output statement(s)

Here is an example program. Programs are usually contained in files, and this one may be called hello.py. By convention, the name of a file containing Python source code will end in “.py”.
# hello.py
# This program will ask the user to enter his/her name,
# and then say hello to that person.
# This program is very simple. There are no calculations.

# Input
name = input("Please tell me your name: ")

# Output
print("Hello, ", name, ", nice to meet you!")

Here is a second example program called animals.py.

# animals.py
# Let's illustrate simple I/O and calculations.
# Add up how many animals on a farm.

horses = int(input("How many horses? 
))
pigs = int(input("How many pigs? 
))
chickens = int(input("How many chickens? 
))

total = horses + pigs + chickens

print("You have a total of ", total, " animals.")

Computer programs share one thing with poetry: they are arranged in lines. Lines that begin with a # symbol are called comments. Comments are included in a computer program for the benefit of the human reader. The computer actually ignores comments. It is extremely important for us to put comments in our programs to help us remember the purpose of a program. Also, notice that some of the lines are blank. Python allows us to format our code to make the program more readable to us. The remaining lines are the actual Python statements. In the first example above, there are just two Python statements. One performs input, and the other does output.

Let's look at a more interesting program, one that computes the area of a circle. It can be stored in a file called circle.py:
This program will find the area of the circle whose radius is supplied by the user. Later on, we could modify the program by having it also compute the circumference.

```
import math

# Input
radius = float(input("What is the radius of the circle? "))

# Calculations
area = math.pi * radius * radius

# Output
print("The area of the circle is ", area, ")."
```

This program contains four statements.

1. The first statement “import math” tells Python that we intend to make use of some special mathematical constants or functions. In this case, we want to use Python’s built-in constant value for pi rather than typing it in ourselves.
2. An input statement for the circle’s radius. We use Python’s built-in input() function to get what the user types. We then ask Python to convert this input into a real number by using the float() function.
3. A calculation statement where we specify the formula for the area of a circle.
4. The output statement.

Many of the programs we will write in this course will have this simple structure: input statements come first, then calculations, and then finally output. The Python system will execute the statements in the order that they are specified.

One important thing to note is that the computer has no common sense. It may be an impressive machine, but we do have to tell the computer how to calculate the area of a circle. In a computer program, we must spell out details like this, because the computer is very good at following our literal instructions. If we enter the area formula incorrectly, the computer will never warn us we made that kind of mistake.

Well, now you have seen some computer programs. Let’s examine them closely. What do they contain?

- Statements – A statement performs one step of the program. Each “line of code” may have one statement. There are several types of statements in Python that we will study. Sometimes, we will use the words instruction or command interchangeably with the word statement. However, in later courses, you will see that there is a slight difference in meaning between these words.
• Variables – A variable is a place in memory to hold information, such as a person’s name or the area of a circle. Every variable has a name, and the name of a variable is called an identifier. Variables have an implicit type, such as integer, float (i.e. real number), and string (i.e. text).
• Constants
• Expressions – often the stuff you see in parentheses or on the right side of an = sign.

The way that we type out programs should make them easy to read.

• Whitespace – This includes spaces, tabs and newlines. They are generally ignored by the Python system. But they are necessary for human readability. For example, please put a space on either side of an operator like + or =.
• Indentation – Sometimes, statements need to be grouped together, and we convey this by indenting. We won’t need to do this yet.
• Continuation – Lines of code should not exceed 80 characters. If for some reason you do need a long statement, put a backslash (\) at the end of the line, and continue typing the statement on the next line. I think we will rarely need to have such long lines in this course.

At this point, you may notice that some words we use are also found in ordinary mathematics. For example, in algebra, one talks about constants, variables, and expressions. One major difference between computing and algebra terminology has to do with the = sign. In algebra, the = sign signifies an equation. An equation is a fact stating that two numbers are the same. But in computer programming, we don’t use the word equation, and we don’t work out equations as you would in algebra. A computer program is a list of instructions for the computer to perform, rather than simply a list of facts about numbers.

Discussion:

1. What is a computer program?

2. What does a computer program look like?

3. Does a program resemble anything you have seen before?

4. What little things do we see in a computer program?

5. How can we tell if it works correctly or not?

6. If incorrect, what could have gone wrong?

7. What different kinds of statements have we seen so far?
Variables

The purpose of a variable is to keep track of a value that we need in our program. Every variable needs a name. The technical term for a variable name is an identifier. Python has specific rules on identifiers.

- It must begin with a letter or underscore.
- After the first character, the identifier may include letters, digits, or underscores.
- Identifiers are case sensitive.
- It may not be the name of an existing function you are using in the program (e.g. input, int, float, str, sum).
- It may not be a keyword (reserved word that already has special meaning in Python). The following table shows a list of words that you may not use for your variables. You do not need to memorize this list!

| and, as, assert, break, class, continue, def, del, elif, else, except, exec, False, finally, for, from, global, if, import, in, is, lambda, None, not, or, pass, print, raise, return, True, try, while, with, yield |

Besides these basic rules, in the computer science community, we have a general rule of style that also advises us that our variable names should be meaningful. If you want a variable to count something, then call it something like `count`. Don’t give it a generic name like `x`. Also, we generally avoid using capital letters and underscores in our variable names, unless we feel that it’s necessary for a variable name to consist of more than one word.

Occasionally it is necessary for a variable name to contain more than one word. For example, suppose your program needs to keep track of the radius and mass of both a planet and its moon. In this case, a variable called `moon` or `mass` would be ambiguous, and we have to resort to using 2-word identifiers. Good variable names in this case would be:

```python
planetMass
planetRadius
moonMass
moonRadius
```

Note the convention of uncapitalizing the first word and capitalizing the second word of the variable name. This is convention is called “camel case.” Alternatively, you could use lowercase letters throughout, and separate the words of an identifier with an underscore, like this: `planet_mass`.

The assignment statement

This is the most important statement type. It allows us to initialize or change the value of a variable. Assignment statements appear very frequently in computer programs. This type of statement uses the
= sign, which is formally called the assignment operator. In most cases, the format of an assignment statement is as follows:

\[
< \text{Variable} > = < \text{Expression} >
\]

To the left of the = sits the variable that we wish to define or update. To the right of the = is the value or the expression that we want to store in the variable.

An aside about my notation here: the symbols < and > when used around a word like < Variable > are called angle brackets. They are used to enclose the name of some category. For example, in a grammar book, you might see angle brackets used when defining the format of a sentence, like this:

\[
< \text{Subject} > < \text{Verb} > < \text{Direct Object} >
\]

Now, here are some examples of assignment statements:

```python
rate = 1.07
name = "Mickey Mouse"
numPeople = 24
root = -b + math.sqrt(b*b - 4*a*c) / (2 * a)
area = height * (top + bottom) / 2.0
```

When we use the assignment operator =, we are issuing a command to the Python system. For example, “rate = 1.07” says that we want the value 1.07 to be stored in a variable called rate. This kind of statement can be translated in English as “let rate be 1.07” or “assign rate the value 1.07”.

There are two possible effects of an assignment statement:

1. If the variable does not already exist, create it, and give it an initial value.
2. If the variable already exists, replace its current value with a new value.

For example, when we say “x = 5”, this has two possible effects. If the variable x has never yet been used in the program up to this point, then the Python system will create a new variable x, and put the value 5 in it. But if the variable x already existed, its old value is overwritten with the value 5. The old value is then discarded. Python will not warn you if a variable in your assignment statement already exists or not. The system will assume that you know what you are doing.

The computer never forgets. When we assign a value to a variable, as in x = 5 for example, this value is stored in the variable for the rest of the program, unless we overwrite this value later. In other words, the computer will not “forget” or lose the values that you assign, until the program is finished. Upon program termination, all of its variables disappear.
For example, suppose a computer program has these statements:

```plaintext
a = 7
b = a + 1
a = 4
```

And then the program does a thousand other things. Assuming that the values of a and b are not reassigned again, at the end of the program, the values of a and b will still be 8 and 4, respectively. Just because you haven’t used a variable for a long time in a program does not mean the information becomes stale or forgotten.

Discussion:

1. What values are contained in the variables a through d after the following code executes?
   ```python
   a = 3
   b = 4
   c = 5
   d = c
   c = a
   b = b + a
   a = d
   ```

2. How should we respond to the following comment made by a former student?
   “Computers make no sense. For instance, they allow you to say n = n + 2. Obviously, that is a false statement. How could n possibly be equal to 2 more than itself?”

Variable types

At this point, there are three variable types that you need to be aware of. They are called `int`, `float` and `str`. They are for integers, real numbers, and strings, respectively. A variable may be of any one of these types. Better yet, you may even change the type of a variable at any time. When you introduce a variable in Python, it is not necessary to give it a type. A variable can be given or reassigned a type as the result of an assignment statement. Recall that the general format of an assignment is as follows:

```python
< variable >   =   < expression >
```

When Python executes this statement, not only does the value of the expression get copied into the variable, but so does the type.

Therefore, Python allows you to change the type of a variable. However, for stylistic reasons, we do not recommend this. The name that you give to a variable should be a strong clue as to the type of value that you wish to store. Some quantities are naturally meant to be integer, such as a number of people.
Money should be considered a real number. And text is a string. Nevertheless, the following code is legal:

```python
x = "hello"
x = 2.5
x = 16
```

Here we see three consecutive assignment statements. In each case, the variable `x` is assigned a value, and this value has an implicit type.

Constants (e.g. “hello”, 2.5, and 16) automatically have a type. A string is easy to spot by its double quotation marks. The difference between a `float` and an `int` is that a `float` constant will always have a decimal point with digits on both sides. Thus, 4 is an `int`, while 4.0 is a `float`. Both these numbers are equal, but they are of different types. In fact, you will even sometimes see number inside a string: “4” is a string rather than an integer or real number.

Python has special built-in functions called `constructor` functions to allow us to change the type of an expression. The names of these constructor functions are simply the names of the types: `int()`, `float()`, and `str()`. For example:

- `int("4")` and `int(4.7)` both return 4
- `float(4)` and `float("4.0")` both return 4.0
- `str(92)` returns “92”

There are some simple rules concerning the type of an expression.

1. When combining expressions of the same type, the resulting type is unchanged. For example, adding integers will result in an integer.
2. When combining an integer and a float, the result will be a float. For example, 2.0 + 3 will result in 5.0, not 5.
3. The `/` operator performs exact division, and its result will always be a float. If you want an integer result instead (and truncate any remainder that might result), then you may use the alternative division operator `//`. So, remember: `/` means exact division, and `//` means truncated division. For example, 29 / 10 equals 2.9, while 29 // 10 equals 2.0.

In Python, mathematical expressions are evaluated just as in ordinary arithmetic. We follow the same order of operations:

1. Parentheses first
2. Then, exponentiation (the exponent operator is `**`)
3. Then, multiplication and division (operators `*` and `/ and `//`)
4. Finally, addition and subtraction (operators `+` and `–`)

Python has one more arithmetical operator that will be very useful for us. It is the `%` operator which is called modulo, or “mod” for short. The purpose of the `%` operator is to calculate the remainder of
integer division. For example, when we do long division on integers, we compute a quotient and remainder. When we divide 14 by 3, the quotient is 4 and the remainder is 2. In Python, we would say that 14 // 3 is 4, and 14 % 3 is 2. The % operator has the same level of precedence as the multiplication and divide operators.

Besides the precedence of operators, we need to know that the evaluation proceeds from left to right. For example, 8 – 3 + 2 is interpreted to mean (8 – 3) + 2. The only exception to this left-to-right evaluation is when we have consecutive uses of the ** operator. As in ordinary arithmetic, the ** is evaluated from right to left.

Discussion:

1. How are variables in Python different from the variables that you saw in high school algebra?

2. Show how the % operator can be used to work out the following problems:
   a. What day of the week is it 30 days from today?
   b. Is this year a leap year?
   c. What is the minimum number of pennies we need to pay 58 cents?

3. By hand, work out this expression: 30 – 3 ** 2 + 8 // 3 ** 2 * 10

4. Write a program that finds the area of a rectangle, using input given by the user.

Shortcut assignment operators

As we saw before, = is called the assignment operator. But in Python, it is occasionally helpful to use a few more assignment operators at our disposal. They are used to make an assignment more concise. There are five of them: +=, -=, *=, /= and %=, Notice that each one consists of a certain arithmetic operator followed by an equals sign. They all work analogously. I will show you how the += works.

+= means that we want to increment (raise) the value of a variable by some amount. For example, if I want to increase the current value of x by 6, I could say x = x + 6. But using the += operator, we can simply write this as x += 6.

You are not required to use shortcut assignment operators, but it could save you from making a mistake. Consider this code: I have two variables, and I want to increase the value of each one by 1.

```python
usa_grid_polygon_x = usa_grid_polygon_x + 1
usa_grid_polygon_y = usa_grid_polygon_x + 1
```

Do you see the mistake? I accidentally typed an “x” instead of a “y” near the end of the second assignment statement. Typos such as these are difficult to find. If I had used the shortcut assignment
operator +=, I would not have needed to type such a long variable name twice. The correct code would look like this:

```python
usa_grid_polygon_x += 1
usa_grid_polygon_y += 1
```

### Algorithm

Before we move on, I think it would be a good idea to explain one very important concept that we will need throughout the course. It is the notion of an algorithm.

An algorithm is a list of instructions written in English that explains how to solve a problem.

Does this sound familiar? How does this definition compare with the definition of a computer program that we saw earlier? They are both lists of instructions; they are both recipes for a solution. But a computer program must be written in a computer programming language, while an algorithm is written in our human language.

When we solve problems, we are basically in the business of writing algorithms. Besides being correct, an algorithm should also be:

- Unambiguous
- Detailed
- Deterministic: As we “control” the CPU, we should be clear where the next step is, and when we are finished.

Later on, we will practice creating algorithms. It’s a very important skill, even more important that the nuts and bolts of any programming language.

At this point, you already know how to write some simple computer programs. From now on, we will learn a little more about Python in order to add to our arsenal. For example, you will be enlarging your knowledge about the different kind of statements that can appear in a Python program, and the different types of variables that you can use.

### Discussion:

1. What data types are available?
2. What do these operators do? `*` `**` `/` `//` `%`
3. What is a shortcut assignment? What good is it?
4. What is an algorithm?
CONTROL STRUCTURES

Objectives:

- The `if` statement
- Adding more choices with `elif` and `else`
- While-loops and for-loops
- Error checking of input

As we have already seen, a computer program is a list of instructions. In what order are the instructions performed? By default, in the order that we specify. In other words, if a computer program has 3 statements, then the computer will perform the first statement, followed by the second statement, and finally the third statement.

To be able to solve more useful problems, we need to write computer programs that can do more than just sequential execution. In particular, two capabilities are needed:

- To make choices; to take a fork in the road. In other words, to follow one path instead of another. This includes the ability to skip some statements that we sometimes don’t need to perform.
- To repeat some steps. In other words, sometimes we want a “loop.”

In computer science, we usually use the term “control structures” when referring to these two capabilities. We will practice making choices and repetition quite a lot, because they are so important in problem solving. Along the way, we will learn some Python keywords to handle these situations. The words `if`, `elif` and `else` are used for making choices, and the words `while` and `for` are used for repetition.

Before we dive into our discussion of control structures, I want to review some basic facts about computer programs:

- A program needs to have output. (Otherwise, how would we know that it works correctly?)
- Output is usually the result of some calculations we need to do in the program.
- A program’s calculations are usually based on input.

This is why we tend to say that a computer program has 3 important parts: input, calculations and output. We can solve a greater variety of problems if we allow our programs also to make choices, to ask questions about either the input values or the results of calculations. A program can decide which path to follow based on whether the answer is yes or no.
The if-statement

Sometimes, we need our program to make a decision. In other words, we want our program to do something based on what values are in our variables. Here is the basic format of an if-statement:

```python
if <condition>:
    # steps to perform if the <condition> is True
```

Optionally, we can include an “else” portion, so that the entire if-statement looks like this:

```python
if <condition>:
    # steps to perform if the <condition> is True
else:
    # steps to perform if the <condition> is False
```

Every if-statement has a condition to test. An if-statement is essentially asking a yes/no question, usually about the value in a variable. Comparisons are done with a relational operator. There are 6 relational operators that we can use in Python:

<table>
<thead>
<tr>
<th>Relational operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to (i.e. is at most)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to (i.e. is at least)</td>
</tr>
<tr>
<td>==</td>
<td>is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
</tr>
</tbody>
</table>

Please note the difference between = and ==. The = operator is used in an assignment statement. The == is a relational operator. To illustrate, consider the following examples:

- x = 5 means that we command the computer to put the number 5 into the variable x.
- x == 5 is a question asking the computer if x is equal to 5 or not.

A comparison is an example of a boolean expression. Boolean means the value is either True or False. It turns out that boolean is actually a data type in Python, along with int, float, and str. So now you know four data types. With the boolean type, it is possible to assign a variable to be True or False. Later, we will see examples where this is especially useful.
Using the words “and” and “or”

When using relational operators to ask a question about a variable, it’s even possible to ask multiple questions, combined with the words and or or. For example, an if statement could perform two comparisons like this: if answer == 1 or answer == 3:

This is sometimes called a “compound condition” because it consists of two or more conditions separated by the words and or or. Here, this first condition is answer == 1, and the second condition is answer == 3. The purpose of the compound condition is to test whether the value of answer is either 1 or 3. Please note that when typing out a compound condition like this, all of the individual conditions that make up the compound condition need to contain a relational operator. For example, one mistake that beginners sometimes make is to write an if condition like this:

    if month == "September" or "April" or "June" or "November":

The intent here is to create a compound condition with four parts. However, the string “April” by itself is not a condition. To write this correctly, we have to spell out exactly what we want as 4 complete conditions, as follows.

    if month == "September" or month == "April" or month == "June" or month == "November":

We often need to use the word and if we want to ask if a variable is between two values. For example, we might want to know if x is an integer between 50 and 100. We could write the condition as follows:

    if x > 50 and x < 100:

As you know, an inequality expression can be turned around as long as we remember to reverse the inequality sign. So, x > 10 has the same meaning as 10 < x. However, when comparing a variable to a constant, it is customary to put the variable on the left and constant on the right. Nevertheless, it is legal to write the above example as:

    if 50 < x and x < 100:

And as soon as we write it this way, we notice that in ordinary mathematics, there is a more compact way of stating this condition. We can take out the word “and” and write the condition this way:

    if 50 < x < 100:

The good news is that Python also allows you to write conditional expressions like this, if you prefer. Note that if we wanted x to be between 50 and 100 inclusive, we would write

    if 50 <= x <= 100:
Here is an example of a program or program fragment that uses if-statements.

```python
# input
n = int(input("Enter an integer: "))

# Tell user if it's positive
if n > 0:
    print("Your number is positive.")
# --------------------------------
# Tell user if the number is odd or even
if n % 2 == 0:
    print("Your number is even.")
else:
    print("Your number is odd.")
```

It is important to take note of some syntax requirements with if-statements.

1. A colon is required at the end of the line introducing the if-statement. You also need a colon at the end of the line containing the word `else`.
2. You need to indent the body of the if-statement, and the body of the else clause. Indentation is required because the body may contain more than one statement. Indentation is Python’s way of recognizing what is part of the body, and what is not. As you type your program, your editor might indent automatically for you. If not, hit the space bar 4 times to achieve the proper level of indentation.

You are already familiar with the concept of indentation when grouping information in a hierarchical manner. Think of an outline, or the bullets of a presentation. For example, suppose you are studying the notes of a meteorology lesson. A simple outline might look like this:

1. Temperature
2. Precipitation
   a. Rain
   b. Snow
   c. Hail
3. Wind

Notice that section 2 has subsections a, b and c. When we finish subsection c and want to proceed to section 3, we unindent back to the level of the Arabic numerals. Python code is indented and unindented the same way, except that we do not label our statements with numbers or letters.
Let’s practice writing if conditions for the following situations. In the following exercises, assume that we have a positive number held in an integer variable called n. In each case, you should write a line of code that introduces an if-statement. It will have this format, and all you have to do is to supply what goes in the blank.

    if ___________________:

1. The number is between 5 and 25 inclusive.
2. The number is less than 12 or equal to 95.
3. The number is even and less than 100.
4. The number is between 10 and 20, or between 80 and 90.
5. The number ends in 7.
6. The number ends in 67.
7. The number is neither 13 nor 17.
8. Now, let’s suppose we have two numbers x and y. Write the condition that says that y is positive and x is strictly between 0 and y.

A very useful application of the if-statement is determining if a year is a leap year. The traditional Julian definition of leap years is pretty simple. We just need to verify that the year number is divisible by 4. Here is what the code would like:

    if year % 4 == 0:
        print("Leap year")
    else:
        print("Not a leap year")

Unfortunately, we no longer use the Julian definition of leap years. It turns out that to be more astronomically accurate, we need to use what is called the Gregorian definition of leap years. The rules to qualify as a leap year are as follows:

- If the year ends in 00, then the year must be divisible by 400.
- If the year doesn’t end in 00, then it must be divisible by 4.

Exercise:

1. Using the Gregorian definition, write Python code that will print “Leap year” or “Not a leap year”, as appropriate, based on the value of the year variable.

2. Give an example of a year that qualifies as a leap year under the Julian definition but not under the Gregorian definition.
3. Let’s calculate someone’s weekly wage. Ask the user for the hourly rate and number of hours worked. Be sure to pay overtime if the number of hours worked is over 40: for each additional hour the hourly rate needs to be 50% higher.

4. Ask the user to enter two numbers. Determine which number is larger, and by how much. For example, if the user enters 2 and 6, tell the user that the second number is larger, and that the difference is 4.

**If-statements using multiple choices**

Just as it’s possible for a program to need several variables, it’s also possible that we need more than 2 alternatives with our if-statements. How would we handle 3 or more? For example, given an input value, we may need to test if it is positive, negative or zero. Another example is converting a numerical grade to one of 5 different letter grades according to a grading scale.

You already know that we use the words *if* and *else* with if-statements. The way to allow for additional choices is by using the Python keyword *elif*, which is an abbreviation for “else if.” Here is the general format of an if-statement that uses elif:

```python
if <condition 1>:
    # steps to perform if condition 1 is true
elif <condition 2>:
    # steps to perform if condition 2 is true
# You can have as many elifs as needed
else:
    # steps to perform if all conditions were false
```

For example, here is how to test for positive / negative / zero:

```python
if value > 0:
    # do positive case
elif value < 0:
    # do negative case
else:
    # At this point, we know value has to be zero!
```

And here is how a grading scale can be used to assign letter grades. If the criteria are 90, 80, 70 and 60, the code would look like this:
if grade >= 90:
    letter = 'A'
elif grade >= 80:
    letter = 'B'
elif grade >= 70:
    letter = 'C'
elif grade >= 60:
    letter = 'D'
else:
    letter = 'F'

Please note that when you write an if-statement with several alternatives, only one of the bodies can execute. Suppose that grade equals 75. Even though 75 > 70 and 75 > 60, the value of letter is ‘C’ not ‘D’, because we execute the body corresponding to the first condition that we encounter that is true. Once we execute the body of the if or one of the elif conditions, we exit the entire if-statement.

Discussion: Modify the grading scale code above to account for plus and minus grades. For example, let’s assume that grades ending in 0 or 1 get a minus, and grades ending in 8 or 9 get a plus. What other assumptions should you make?

**Repetition using the while statement**

“Stir the soup until it boils.” The concept of doing something repeatedly is second nature to us. Now let’s teach the computer how to repeat an action.

In Python, we use a while statement to allow some statements to be performed several times. What is really nice is that the format of a while statement is very similar to if:

```python
while <condition>:
    # steps that will be repeated
```

This group of statements that starts with the while statement and includes the body is called a loop. In particular, we would call this a “while loop” since it began with the word while. There is another type of loop called a “for-loop” that we will see later. Doing the body of the loop each time is called one iteration of the loop.

Here is how a while loop works: Python will first test the condition. If it is already false, then the entire loop is skipped and we go on to the next piece of the program. But if the condition is true, then we enter the loop and perform the statements in the body. When the body is finished, Python will re-evaluate the condition. If it is still true, the body is performed again! The process repeats itself. At some point, the condition needs to become false, so that we can eventually exit the loop.

The most basic kind of loop we need to write is how to teach the computer how to count. The following code will print the numbers from 1 to 5.
print("I’m going to count to 5:")

n = 1
while n <= 5:
    print(n)
    n += 1

print("I’m finished counting.")

Many of the while loops we will write will have the same basic structure of this example. And once you have written one loop, it is easy to adapt it to similar other problems. For instance, if we can count to 5, then we can easily modify the above code so that it can count to 1000 instead of 5. How would we make this change?

Another simple modification we can make to the above loop is to count by fives:

print("Let’s count by fives from 0 to 100.")

n = 0
while n <= 100:
    print(n)
    n += 5

Discussion:

1. How would we modify the code so that it also prints out the sum of the numbers that it prints? In other words we want the sum of the multiples of 5 from 0 to 100 inclusive.

2. How would we modify the above code so that it prints the numbers in reverse order, i.e. from 100 down to 0?

3. Try this experiment: What errors result if your program is careless about indentation?

4. What numbers are printed by this Python code?
   n = 8
   while n < 12:
       print(n)
       n += 1
5. What is the output of this program?

```python
n = 20
while n >= 5:
    print(n)
    n -= 1
print("After the loop, n equals", n)
```

Important observation: With a loop there is very often some “counter” variable that keeps changing on every iteration. In the above examples, it was this variable `n`. Notice that before the loop, `n` received some `initial value`, such as zero. Then, when we introduced the while loop, we established some `limit` on the value of `n`, such as 100. Finally, at the end of the loop body, we `increment` the counter variable by some amount. Almost always this increment is 1, but in the above example we saw that it could be something else like 5.

So, when you write loops, you should remember: `initial value, limit value, increment`

Since all the while loops we have seen have had the same basic structure, it is useful to keep in mind a basic pattern. Suppose you want a loop to do something exactly 20 times. Here is how you would do it using a while loop:

```python
n = 0
while n < 20:
    # do whatever you need the loop to do on each iteration
    n += 1
```

The way the above code is written, it is guaranteed to perform the body exactly 20 times. If you wanted to do it 30 times, just change the 20 to 30.

Application: Prime numbers

A prime number is a positive integer that has exactly 2 factors, namely 1 and itself. Examples of prime numbers are 2, 3, 5, 7, 11, 13, etc.

How can we tell if a number `n` is prime? Here is an algorithm to do it.

- For each possible `divisor` from 1 to `n`, see if it divides evenly into `n`.
- We need to `count` how many times we find a `divisor` that divides evenly into `n`. Store this in another variable.
- When finished, see if the count is 2.

Let’s illustrate the algorithm with some examples.
Is 5 a prime number?

- The possible divisors to test are 1, 2, 3, 4, 5.
- Do they divide evenly into 5?  1 \rightarrow yes, 2 \rightarrow no, 3 \rightarrow no, 4 \rightarrow no, 5 \rightarrow yes
- We counted 2 divisors. It is prime.

Is 6 a prime number?

- The possible divisors to test are 1, 2, 3, 4, 5, 6.
- Do they divide evenly into 6?  1 \rightarrow yes, 2 \rightarrow yes, 3 \rightarrow yes, 4 \rightarrow no, 5 \rightarrow no, 6 \rightarrow yes
- We counted 4 divisors. It is NOT prime.

Here is the Python program to determine if a number is prime. It follows exactly from the algorithm just described above.

```python
n = int(input("Enter a positive integer: "))

# The divisor goes from 1 to n, inclusive, just like we are able
# to count from 1 to n. Also, start count at 0.
count = 0
divisor = 1
while divisor <= n:
   if n % divisor == 0:
      count += 1
   divisor += 1

if count == 2:
   print("Your number is prime.")
else:
   print("Your number is not prime.")
```

Discussion:

1. Where is the variable `count` used in this program?

2. Where is the variable `divisor` used in this program?

3. What is the difference between the variables `divisor` and `n`?

4. If the value of `n` is 9, then what are the values of `count` and `divisor` at the end of the program?
More about loops

Earlier we wrote a loop to some simple counting and adding. Now, let’s create a more useful loop: let’s add five input numbers. We want to allow the user to enter 5 numbers, and calculate their sum. We’ll need a loop, and inside the loop we need to do the following:

• Ask the user for the next number.
• Continually add this next number to the sum.
• Keep track of how many numbers so far have been read.

The information that we need to keep track of tells us what variables we want.

Important observation: If you want to add or count something, make sure your variable starts at zero.

Here is the program.

```python
# add5.py – Let’s ask the user for 5 numbers, and then
# their sum. It’s critical to think about the
# variables we need.

count = 0
sum = 0
while count < 5:
    nextNumber = int(input("Enter a number: "))
    sum += nextNumber
    count += 1

print("The sum of your numbers is ", sum)
```

Discussion: What does this code accomplish?

```python
# add5.py – Let’s ask the user for 5 numbers, and then
# their sum. It’s critical to think about the
# variables we need.

count = 1
sum = 0
while count <= 100:
    sum += count
    count += 1
print(sum)
```
Common loop mistakes

We all make mistakes. Be patient and forgiving with yourself and others. When composing programs, we need to be careful and methodical, because one small mistake can have a profound effect on the output. There are three easy ways to mess up a loop, and I’ll illustrate them here with the task of trying to print the numbers from 1 to 10 inclusive.

In each example, see if you can determine exactly why the code is in error, and how you would fix it.

Mistake #1 – Condition initially false. In this case, the loop will not execute at all. It gets skipped.

```python
num = 1
while num > 10:
    print(num)
    num = num + 1
```

Mistake #2 – Infinite loop. Here, the condition is always true; never has a chance to become false.

```python
num = 1
while num <= 10:
    print(num)
```

Mistake #3 – Off by one error. This means we have one iteration too many or too few. I will illustrate both ways of making this error.

```python
num = 1
while num < 10:
    print(num)
    num = num + 1
```

Error checking of input

Now that we know how to write loops and if-statements, there is a lot we can do. As you know, very often in our programs, we need the user to enter some input. But what if the input doesn’t make any sense? The easiest thing is for the program to just accept the invalid input and ultimately spit out some worthless answer. But it would be more useful if the program can warn the user of a problem with the input, and then give the user another chance. This is called error checking.

Let’s say we want the user to enter a positive number. If the user enters zero or a negative value, we need to loop until the user supplies us with a positive number.
The key to error checking is to use a Boolean variable. I like to have a variable called \texttt{needInput}. This variable is initially set to \texttt{True}, because we definitely need input from the user. If the user supplies us with good input, then the value of \texttt{needInput} can be set to \texttt{False}, and this will be our signal that we don’t need to ask again for the input. But if the input is invalid, we need to print some kind of error message and have the user try again.

Here is the code to do error checking. You will find it very useful in this class and in the future!

\begin{verbatim}
needInput = True
while needInput:
    value = int(input("Enter positive number: "))
    if value > 0:
        needInput = False
    else:
        print("Sorry, try again.")
\end{verbatim}

Exercise:

1. How would you perform error checking if we wanted the user to enter an odd number? In other words, rewrite the above code to accomplish this, making the appropriate changes, even though the overall structure will be the same.

2. How would you perform error checking if we wanted the user to enter just a single digit from 0 to 9?

The \texttt{for} loop

Python provides a second way to write a loop, using the keyword \texttt{for} instead of \texttt{while}. Here is the format of a \texttt{for} loop:

\begin{verbatim}
for <variable> in <list>:
    # steps to perform on each iteration
\end{verbatim}

The most common way to write a \texttt{for} loop is to count, like this example:

\begin{verbatim}
for i in range(5, 15):
    print(i)
\end{verbatim}

Python has a very useful built-in function called \texttt{range} that creates a list of numbers that we can incorporate into our \texttt{for} loop. The \texttt{range} function takes 2 parameters. The general meaning of \texttt{range(a, b)} is to return us a list of numbers starting at \texttt{a} and going as high as \texttt{b – 1}, inclusive.
For example, `range(3, 7)` returns this list: 3 4 5 6

So, if we want to print the numbers from 1 to 10 inclusive, we need to call the `range` function and make sure that it will return us a list containing exactly these numbers. If we said `range(1, 10)`, this will give us the list 1 2 3 4 5 6 7 8 9. The number 10 would be omitted. We must instead say `range(1, 11)`.

*General observation*: Very often in Python you will notice that whenever we use some interval of numbers, such as from 1 to 10 in some built-in function like `range`, you should typically expect the range to begin with the first number, and then end with the last number minus 1.

So, to print 1-10 using a `for` loop, we would say:

```python
for i in range(1, 11):
    print(i)
```

One convenient feature of the `for` loop is that the increment statement for our counter variable is omitted. If we had used a `while` loop instead of a `for` loop, we would have to explicitly include a statement “i = i + 1” at the end of the body of the loop.

It is generally a good practice to use the variable `i` to be the counter of a `for` loop. The letter `i` stands for “index”.

While we are on the subject of the `range` function, there is a variant of the `range` function that accepts three parameters, like this: `range(0, 100, 5)`. The third parameter is the stride. It would allow us to count by increments other than 1. The stride can even be negative to allow us to count backwards. However, we won’t need to do this very often.

Discussion:

1. **What does this code accomplish?**
   ```python
   sum = 0
   for i in range(2, 7):
       print(i)
       sum = sum + i
   print(sum)
   ```

2. **What does this code accomplish?**
   ```python
   sum = 0
   for i in range(1, 6):
       sum = sum + i
   print(sum)
   ```

3. **What does this code accomplish?**
   ```python
   for i in range(3, 8):
       print(i ** 2)
   ```
4. What does this code accomplish?
   count = 0
   for i in range(1, 100):
       if i % 7 == 0:
           count += 1
   print(count)

5. Write code containing a loop that will print the multiples of 4 from 4 to 48 inclusive.

6. Write code containing a loop that will count how many positive integers are evenly divisible into the number 50.

7. Write code containing a loop that will print all 3-digit numbers that end in 7. When you are done, you might notice that there is a lot of output. So, modify your program so that it prints only 12 numbers per line.

8. Ask the user to enter a digit. Then, print all 2-digit numbers that end with this digit.

9. The range function returns a list, and does not have to be associated with a for-loop. For instance, assuming that n is an integer, the if-statement
   if n in range(5, 15):

   is equivalent to this if-statement
   if _______ <= n <= _______

   Fill in the blanks.

Nested loop

A nested loop is simply a loop within a loop. These are actually quite common in real computer programs because they allow the computer to do a lot of work with a small number of program statements.

A nested loop is useful whenever we need to process multi-dimensional information. In this course, we will not encounter this very often. But here are some general examples:

- An image consists of rows and columns of pixels.
- A video is essentially a list of images, so this adds a third dimension of time to our images.
- Games often have a 2-D board.
- Airline and hotel reservations. For example, a seat on an airplane is given by a row number and which seat in that row.
In real life, our brains process nested loops all the time! Here are typical examples:

- For each week... for each day of the week... for each hour of the day
- For each building... for each room in the building

I need to caution you that we should not confuse a nested loop with consecutive loops. Just because you see two loops does not automatically mean that one loop must be inside the other. Consider the following examples. In the first case, the two loops are simply consecutive. In the second case, the two loops indeed nest.

# Example #1
```python
count = 0
for i in range(0, 10):
    count += 1
for j in range(0, 20):
    count += 1
print(count)
```

# Example #2
```python
count = 0
for i in range(0, 10):
    for j in range(0, 20):
        count += 1
print(count)
```

What is the output of each loop situation here?

As another example, let’s suppose we wanted to print a field of star/asterisk (*) characters having 10 rows and 20 stars per row. Here is how it would be done:

```python
for i in range(0, 10):
    for j in range(0, 20):
        print("*", end="")
    print()
```

Notice that the nested loop is set up exactly the same way as in Example #2 above. This is because we want 10 outer iterations and 20 inner iterations. We want to print $10 \times 20 = 200$ stars. Since we wish to print the stars right next to each other within the line, we add a second argument to print: `end=""`.

This overrides the default behavior of print, which would have printed a newline character after the star. In this case, "" means print nothing at all between the stars.
Also notice the second print statement in the code. It is positioned immediately after the inner loop, but it is located within the outer loop. This makes sure that we print the newline after each row of stars. It is critically important that we indent the code correctly in this example.

Discussion: Make the following modifications to the above source code that printed the field of stars. Describe what effect each change has on the output.

1. Change `end=" "` to `end = " | ". (Change it back before continuing to the remaining questions.)

2. Indent the final `print()` statement further to the right so that it is directly underneath the first `print` statement.

3. Unindent the final `print()` statement so that it vertically lines up with the word `for`.

**Break and continue**

The Python keywords `break` and `continue` are special statements that allow you to interrupt the normal flow of a loop. They are powerful statements, and most loops that we will write will not need to use them. A common mistake for novices is to use `break` or `continue` when it is not necessary.

The purpose of the `break` statement is to immediately abandon the loop. We would do this when we realize that we have completely finished doing the work that the loop needed to do, and that further iterations are not necessary.

The purpose of the `continue` statement is to abandon or skip just the current iteration of the loop. Here is an analogy. Let's say you want to play all of the songs on a CD or record. That is a loop. But let’s suppose that you don’t like the third song. You want to skip it. When you get to the third song, you would want the mechanism to immediately “continue” to the next song.

You can think of the difference between `break` and `continue` like this -- one morning you become suddenly sick at work, and you ask the boss if you can leave work early and come back tomorrow. That is a “continue” situation. On the other hand, you might decide to quit your job. That would be “break.” Break has a stronger effect than continue.

Here is a simple example of using the `continue` statement. We can print the numbers 1-10 but skip the number 3:

```python
for i in range(1, 11):
    if i == 3:
        continue
    print(i)
```
If we had used the word `break` instead, the code would only have printed the numbers 1 and 2. We use `break` more often than `continue`. The `break` statement is often used when we want to search for something. Suppose you left your umbrella inside your friend’s house, but you can’t remember which room of the house you left it. You begin a methodical search of the house, going from room to room. Suppose your friend’s house has 10 rooms. Will you have to search every room? Probably not. You would stop searching once you find the umbrella. The pseudocode would look like this:

```
for each room in my friend’s house:
    Look for the umbrella in this room
    if I see the umbrella,
        Grab it
        break
```

Without the `break` statement, we would keep searching for the umbrella all over the house even after having found it! So, we see that using the `break` statement can make our code more efficient.

As with any programming construct, the `continue` and `break` statements can be misused or overused (i.e. used when they are not necessary). It turns out that we can always write a loop without the words `continue` and `break`. In fact, many programmers tend to avoid these two kinds of statements. Here is how the umbrella-finding example would be written in pseudocode without making use of the `break` statement:

```
found = False
while found == False and there are still rooms to search in:
    Go into the next room
    if I see the umbrella,
        Grab it
        found = True
```

Discussion:

1. In the last example above, why do we need the statement `found = True`?

2. Earlier, you saw the example of a loop that uses a `continue` statement to skip the number 3 while printing the numbers 1-10. Rewrite this loop (without changing the output, of course) so that it does not use the `continue` statement.
STRINGS

Objectives:

- How text data is internally represented as a string
- Accessing individual characters by a positive or negative index
- String slices
- Operations on strings: concatenation, comparison, “in”
- Using Python’s built-in string functions

Not all data is numerical. The purpose of the string data type is for us to keep track of text information. A string is a collection of any number of characters. And a character is basically anything that you can type on the keyboard, including letters, digits, punctuation symbols, and blank spaces, tabs and newlines. (A newline is a special character that results when you hit the Enter/Return key and signals your computer that it should go to the next line of a text document.)

In Python, we denote a string constant by enclosing it in either single quotes ‘like this’, or inside double quotes “like this”. By convention, most people prefer to use double quotes. You may use either single or double quotes for your string constants, but be consistent.

Python allows variables to hold string values, just like any other type (Boolean, int, float). So, the following assignment statements are valid:

```python
name = "Nick Charles"
name = "N"
name = ""
```

In the last example above, there was nothing between the quotation marks, and this is called the empty string, not to be confused with strings containing only invisible characters such as " " or "     ".

Accessing a character

Since a string is a collection of characters, it makes sense to want to access individual characters or parts of the string. Python uses the brackets [ ] to access portions of a string. Before I show you all the syntax for using the brackets, it is important to understand how the individual characters of a string are numbered.

Inside a string, each of its characters lives inside a cell, and these cells are numbered sequentially from zero. Thus, if s is a string variable and s = “horse”, then the individual characters are located as follows:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>‘h’</td>
<td>‘o’</td>
<td>‘r’</td>
<td>‘s’</td>
<td>‘e’</td>
</tr>
</tbody>
</table>
To access an individual character of a string, simply put its index inside brackets after the name of the string. So, s[0] refers to the first character of the string variable s, which in this case is ‘h’. Similarly, s[1] is ‘o’, s[2] is ‘r’, s[3] is ‘s’ and s[4] is ‘e’. Right away, you should notice something about the size of the string and its index values. In this example, our string has 5 characters, and they are located at indices 0..4 (i.e. 0 through 4 inclusive). No matter how long a string is, its first character is always at index 0. The index of the last character is always 1 less than the length of the string.

Think of an index like an apartment number. The letter ‘r’ is located at index 2.

Interestingly, Python even allows us to use negative index values when using the bracket notation. In this case, a negative index number means we are counting from the right end of the string. Thus, the index –1 refers to the last (i.e. rightmost) character of the string. The index –2 is the second from the end, and so on. The nice thing about negative indexing is that you don’t need to know the length of the string. The index –1 refers to the last character no longer how big the string is.

The following table illustrates positive and negative indices for the string “doghouse”.

<table>
<thead>
<tr>
<th>Positive index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>‘d’</td>
<td>‘o’</td>
<td>‘g’</td>
<td>‘h’</td>
<td>‘o’</td>
<td>‘u’</td>
<td>‘s’</td>
<td>‘e’</td>
</tr>
<tr>
<td>Negative index</td>
<td>–8</td>
<td>–7</td>
<td>–6</td>
<td>–5</td>
<td>–4</td>
<td>–3</td>
<td>–2</td>
<td>–1</td>
</tr>
</tbody>
</table>

This idea of numbering from both ends is common to us in real life. Imagine you are giving directions to find someone’s house along a street, or an office along a hallway. For example, let’s say that along some hallway there are eight offices. Your office could be, say, the third door on the left, if I come down the hallway from a certain direction. However, if I came from the other direction, you would instead say it’s the sixth door on the right. This situation is exactly the same as looking for the letter ‘g’ in the word “doghouse” above, except that in Python the default numbering starts at zero instead of one.

**String slice**

Sometimes one character is not enough. A string slice is a collection of several consecutive characters from a string. Again, we use the bracket notation, and specify two values separated by a colon. If s is a string, then the notation s[a:b] means that we want (b – a) characters from the string starting at index a.

For example, if s = “doghouse”, then s[2:5] means that we want 5 – 2 = 3 characters starting at index 2 of the string. This means we have “gho”. A convenient way to figure out a string slice is to remember that the notation s[a:b] is analogous to the way we wrote for loops with “for i in range(a, b)”. Then, we see that s[2:5] will give us the characters s[2], s[3], and s[4] only. The result of a string slice is another string.
For added flexibility, Python allows us to omit the numbers a and/or b in the notation s[a:b]. Omitting the first number (a) means that we start the string slice at the beginning of the string. Omitting the second number (b) means that we finish the string slice at the end of the string. This is helpful when we don’t know exactly how many characters are in the string, or we don’t wish to count them.

Examples: if s = “doghouse”, then

- s[:3] means we want the first 3 characters of s, which gives us “dog”.
- s[3:] means we want everything starting at index 3, which gives us “house”.

Incidentally, a string slice can also have an optional third argument representing the stride. In general, the notation s[a:b:c] means to begin the string slice at index a, stop just before index b, and include only the characters at indices that are c apart. In other words, this string slice will include the characters at index a, a+c, a+2c, etc. until we reach index b. For example, if we have s = “doghouse”, then s[:3] means to give every third character starting at the beginning, which gives us “dhs”.

Continuing our analogy between string slices and the range function, consider this example:

```python
for i in range(5, 20, 4):
    print(i)

s = "abcdefghijklmnopqrstuvwxyz"
print(s[5:20:4])
```

Here, the `for` loop will print the numbers 5, 9, 13 and 17. Similarly, the string slice will contain the characters at positions 5, 9, 13 and 17 within our string, resulting in “fjnr”.

*Concatenation* is another fundamental operation on strings. It means to merge two strings into each other. We use the + sign. It works for string variables, string constants, or any combination. Example: “ab” + “cde” results in the string “abcde”.

**Discussion:**

1. In the string s = “abcdefghijklmnopqrstuvwxyz”, which letters are located at s[4] and s[–4]?
2. If s = “Salt Lake City”, then what string slice equals just the word “Lake”?
3. What is the output of the following code?
s = "dolphin"
print (s[1] + s[4])
print (s[4] + s[1])
print (s[1:4])

4. Let s = “cat” and t = “house”. Create an expression using s and t that equals “house cat”.

Changing a string

There is a technical point about strings in Python. String objects cannot be changed. If you have a string variable, and you just want to change one of its characters, you cannot simply write an assignment statement to just a character. You have to assign to the entire string.

Going back to s = “doghouse”, if we subsequently decided to start s with a capital letter D, the following statement is illegal: s[0] = ‘D’. To change one letter of a string, we can do the following:

- Create a slice of the string that excludes the first character.
- Create a new string starting with the letter “D”, and concatenate with the old string slice.

The resulting assignment statement becomes: s = “D” + s[1:]

Here is another example. Suppose s is a string containing several words, and we want to modify s by appending a period at the end of the string. Here is how to do it:

s = s + "."

Incidentally, the above assignment statement can also be rewritten using a shortcut assignment operator +=. Just as + has different but analogous meanings for numbers and strings, so too for +=. If x and y are strings, then the statement x += y means to change x by concatenating y onto the end of x. The string y is unchanged. For example, this code:

word = "dog"
word += "s"

results in the word "dogs" being stored in the variable called word.

Important operations on strings

You have just seen concatenation of string, but there is more.

We can compare two strings using relational operators, and thereby compare them just like we compare numbers. Python relies on Unicode values of each character to determine alphabetical order. For instance, ‘a’ is considered the “lowest” letter of the alphabet, and ‘z’ is the “highest” letter. So, ‘a’ < ‘b’,
‘b’ < ‘c’, and so on. Another way to think of < is like this: if s1 and s2 are strings, then s1 < s2 means that s1 should appear earlier in the dictionary than s2. For example, “cat” is less than “dog”.

Long ago, someone made the arbitrary decision that capital letters are “less than” lowercase letters. Consequently, “Cat” < “cat”. Also, the absence of a character is less than any real character. For example, “dog” is less than any 4-character string that begins with “dog”. In particular, “dog” < “dogs”.

Another useful operator is the Python keyword `in`. It is used to see if a letter or substring exists in a string. It returns True or False. For example, “x” in “aeiou” returns False.

Important string functions

Functions are written by people to save us time and effort in coding. Our Python installation has several useful built-in functions for strings. Here are some of the most commonly used string functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td>How many characters in s</td>
<td><code>len(“dog”)</code> returns 3</td>
</tr>
<tr>
<td><code>s.find(&lt;letter or pattern&gt;)</code></td>
<td>Returns index of first occurrence of the letter or pattern</td>
<td><code>s = “dog”</code> s.find(‘o’) returns 1</td>
</tr>
<tr>
<td><code>s.upper()</code></td>
<td>Returns s with all of its letters converted to uppercase. Does not change s.</td>
<td><code>s = “Dog”</code> s.upper( ) returns the string “DOG”</td>
</tr>
<tr>
<td><code>s.lower()</code></td>
<td>Analogously returns s with all of its letters converted to uppercase. Does not change s.</td>
<td><code>s = “Dog”</code> s.lower( ) returns the string “dog”</td>
</tr>
<tr>
<td><code>s.replace(&lt;old char&gt;, &lt;new char&gt;)</code></td>
<td>Returns a new string where every occurrence of the old char is replaced with the new char. Does not change s.</td>
<td><code>s = “strings”</code> s.replace(“s“, “oa”) returns the string “oatringoa”</td>
</tr>
<tr>
<td><code>s.count(&lt;letter or pattern&gt;)</code></td>
<td>Returns how many times the substring appears in s.</td>
<td><code>s = “the cat in the hat”</code> s.count(“at”) returns 2</td>
</tr>
</tbody>
</table>

There is no function to reverse a string, but we can accomplish this if we use –1 as the third parameter of a string slice. In other words, s[:: –1] is the reversal of the string s.

Exercise: Suppose we just got a word from the user, and it is now stored inside a string variable s. How would we determine the following?

1. If it contains the letter T, capital or lowercase
2. The number of times the capital letter T appears
3. The location of the first capital T
4. The location of the second capital T, assuming that it exists
5. If it contains a digit
6. If it contains at least 2 vowels
7. The number of vowels
8. The locations of all the vowels
9. If it has more than 5 characters
10. The fifth character
11. The last 3 characters
12. If the first and last characters are the same
13. If all of its letters are capitalized

More questions about strings: You should be able to work each problem out by hand without using the computer, but feel free to check your answer with the Python system.

14. Suppose `s.find("g")` returns 3. What can we conclude about the string s?

15. Suppose `s.count("g")` returns 3. What can we conclude about the string s?

16. What is the output of the following code?
   ```
   word1 = "well"
   word2 = "done"
   word1 += word2
   print (word1)
   ```

17. What is the string contained in s2 when the following code finishes?
   ```
   s1 = "dolphin"
   s2 = ""
   for c in s1:
       if c in "aeiou":
           s2 += c
   ```

18. What is the output of the following code?
   ```
   s1 = "dolphin"
   s2 = ""
   s3 = ""
   for c in s1:
       if c in "aeiou":
           s2 += c
       else:
           s3 += c
   print (s2, s3)
   ```
LISTS

After int, float, str and boolean, we are now ready to learn a fifth data type: the list.

Lists are great! The list is probably the most versatile data type in Python. The major benefit of the list is that it allows us to store many values and associate them with the same variable. For instance, suppose your program needed to keep track of 30 numbers. Does it make sense to store them in 30 different variables? We would have to come up with distinct names for these 30 variables. Instead, we can put all these numbers into a single list. The magic of the list is that the basic representation and syntax is like a string.

For example, we can create a variable called data, and store the numbers 5, 4, 7 and 3 in it as a list like this:

\[
data = [5, 4, 7, 3]
\]

There are two things to note here. First, individual elements of a list are separated from the others by commas. Second, the brackets are used to indicate the beginning and end of the list. So, the bracket symbols are doing double duty, because as you saw with strings earlier we also use brackets to reference individual pieces of information.

After having initialized our list, we can refer to the individual numbers just like we did for strings.

\[
data[0] \text{ is } 5
\]
\[
data[1] \text{ is } 4
\]
\[
data[2] \text{ is } 7
\]
\[
data[3] \text{ is } 3
\]

And just like strings, we can use the [ ] notation to grab various parts of a list. Suppose L is a list.

- Negative indices allow us to count from the right end. \(L[-1]\) refers to the last item in a list.
- Slices to refer to part of a list. \(L[2:3]\) refers to the third and fourth elements, and this expression can itself be treated as a list.
- Slices with a third argument indicate a stride. \(L[5:50:10]\) means the part of the list containing just the items at indices 5, 15, 25, 35, and 45.
- Slices with arguments omitted. \(L[-5:]\) means we want just the last 5 elements from \(L\).

There are two ways to create a list in a Python program. The first way is to “hard code” it, as you just saw above. This is very simple, but it suffers the disadvantage that if you wanted to initialize your list to contain different data instead, you would have to re-edit the program before running it again.

The second method for initializing the list is to append data into it. I much prefer this method, because I can test my program on any data I want without having to change my program. Here is what you need to do to initialize a list this way:
• Begin with an empty list. For example, with the statement L = []
• Use the built-in append() function to add one element at a time to the list.
• Use a loop to append multiple values, and obtain the list values from input.

Here is an example of what this looks like.

```python
# We begin by asking the user for how many values.
size = int(input("How many values for list? "))

L = []
for i in range(0, size):
    L.append(int(input("Enter a value: ")))

# Let's output the list
for item in L:
    print(item)
```

Here is another possible approach. Sometimes, the user does not know the size of the list in advance. In this case, the user can signal the end of the input with a special "sentinel" value such as -1, assuming that -1 is not a legitimate input value for the list.

```python
L = []
while True:
    value = int(input("Enter a value, -1 to quit: "))
    if value == -1:
        break
    L.append(value)

for item in L:
    print(item)
```

Discussion:

There are two ways to print out the elements of a list. Experiment with both, shown below, and describe the difference in the output.

```python
for item in L:                       print(L)
    print(item)
```
Important list functions

Suppose that \( L \) is a list. Here are some things we may want to do with the list. Usually, our list contains numbers. There are functions to perform basic statistics. And then there are functions that make useful modifications to the list. You have seen the \texttt{append( )} function already.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{len(L)}</td>
<td>Returns the number of values in the list</td>
</tr>
<tr>
<td>\texttt{max(L)}</td>
<td>Returns the highest number in the list</td>
</tr>
<tr>
<td>\texttt{min(L)}</td>
<td>Returns the lowest number in the list</td>
</tr>
<tr>
<td>\texttt{sum(L)}</td>
<td>Returns the sum of the numbers in the list</td>
</tr>
<tr>
<td>\texttt{L.append(value)}</td>
<td>Inserts the new value at the right end of the list</td>
</tr>
<tr>
<td>\texttt{L.insert(index, value)}</td>
<td>Places the value at a certain index of the list. The existing element at that index and everyone to its right get shifted over to the right to make room for the new value.</td>
</tr>
<tr>
<td>\texttt{L.remove(x)}</td>
<td>Removes the first object having the value ( x ). Items to its right shift to the left.</td>
</tr>
<tr>
<td>\texttt{L.reverse( )}</td>
<td>Changes ( L ) to its reversal</td>
</tr>
<tr>
<td>\texttt{L.sort( )}</td>
<td>Changes ( L ) to its ascending order</td>
</tr>
<tr>
<td>\texttt{L.index(value)}</td>
<td>Returns the first location of a value in the list. For example, if ( L = [ 5, 4, 7, 4 ] ), then ( L.index(4) ) is 1. A run-time error results if the value doesn’t exist.</td>
</tr>
</tbody>
</table>

Note that most of the functions listed above only make sense if the elements of \( L \) are all of the same type. Numbers can be compared with other numbers, strings can be compared with other strings, but numbers and strings cannot be compared. However, in practice our lists will generally contain values all of the same type, as in a list of only numbers or only of strings.

We also need to be careful concerning the proper way to copy lists. This is one way in which lists do not quite behave like other variables. For example, if \( a \) is an integer variable, and you wanted to copy this value into a new variable \( b \), this is easy: you would enter the statement \( b = a \).

However, this technique does not work for lists. If \( A \) and \( B \) are lists, then \( B = A \) has a special meaning. It means that we want \( B \) to refer to the same list as \( A \). The statement \( B = A \) does not copy anything. Instead the same list now has two different names (aliases). This is not what we want!

Here is the correct way to copy a list \( A \) into a new list \( B \). This technique should remind you of how we initialized lists from input. First, create a new empty list \( B \). Then, traverse the list \( A \): for each value in \( A \), we append this value onto the end of \( B \). Here is the code:
B = [ ]
for value in A:
    B.append(value)

To review the meaning of =, let’s look at a simple example involving integers.

a = 1
b = a
a = 2

In this example, the second statement places the value 1 (the current value of a) into b. The final value of a is 2, and the final value of b is 1.

Let’s repeat this example using lists. Comments beside each statement explain the effect.

A = [1]    # Create a new list with the number 1 in it; call it A
B = A      # Make B refer to the same list.
A = [2]    # Create a new list with just 2 in it; call it A

In this example, the final value of A is [2] and the final value of B is [1]. The third statement creates an entirely new list and gives it to the variable A. The original list [1] is still referred to by B.

Finally, consider this example:

A = [1]    # Create a new list with 1 in it; call it A
B = A      # Make B refer to the same list.
A.append(2)   # Append 2 to the list referred to by A and B.

In this case, there is only one list! Both A and B refer to the same list, [1, 2].

Now, let’s practice with lists. Assume that L is a list of numbers. Explain or show how to find ...

1. How many values are positive
2. The sum of just the positive values
3. The last value
4. The largest value
5. The second largest value ... Can you do it without changing the list?
6. The median
7. The location of the first zero

8. The location of the second zero

It’s important to note that the individual elements of a list do not have to be numbers. They can be anything, including strings. For example, the following code creates a list of strings and prints each string out, one per line.

```python
food = ["hamburger", "hot dog", "pizza"]
for s in food:
    print(s)
```

Discussion: How can we modify the above loop so that it counts how many times the letter ‘g’ appears in the list of strings?

An interesting problem that we can solve later has to do with a heterogeneous list, for example a list that has both numbers and strings. We can skip the strings when doing arithmetic on the list.

The **enumerate function**

For a list, the purpose of the enumerate function is to allow us to visit all the elements of a list, and be able to see both the value as well as its index.

Usually when we traverse a list, we are ignorant of the index. Here is an example.

```python
L = [ 5, 7, 4, 2, 3, 8, 1 ]
for value in L:
    if value % 2 == 1:
        print("I found the odd number ", value)
```

The above code will find all of the odd numbers in the list. But where do these numbers live in the list? In other words, what are their indices or apartment numbers? This is why we need the enumerate function. Our code then becomes:

```python
L = [ 5, 7, 4, 2, 3, 8, 1 ]
for index, value in enumerate(L):
    if value % 2 == 1:
        print("I found the odd number", value, "at index", index)
By using `enumerate()`, we simultaneously know both the location and the contents of each number in the list. Had we not used `enumerate()`, we would not know the locations.

Exercise: Modify the above code so that it also:

1. Finds the sum of all the odd numbers in the list.
2. Places all the odd numbers into a new list.
3. Places the indices of the odd numbers into a third list.
4. If the original list is `L = [5, 7, 4, 2, 3, 8, 1]`, then what values are contained in the two new lists?

And now try a more general question about lists:

5. How would you find the identity and location of the largest number in a list?

One final note about lists. The list is not the only way to aggregate data in Python. The Python language includes additional data types such as tuple, set, and dictionary. But we will not study them in this course.
In computer programming, we classify errors into three categories. The word “bug” is also used as slang to refer to any of these mistakes.

**Syntax error** – This means that you have entered a statement that the computer does not understand. If the machine is not 100% certain of what you are trying to do, it will give you this kind of error message, and refuse to run the program. Even the most trivial typos can give rise to a syntax error.

A syntax error is usually the easiest kind of mistake to fix. The word syntax basically means grammar, so a syntax error typically means that your program contains at least one statement that violates the rules of the Python language. For example, misspelling the name of a keyword, missing or extra operator or punctuation symbol. If your program contains a syntax error, the Python system will immediately tell you about it when you try to run your program. Because the computer does not understand your program, it won’t even start to run it.

**Run-time error** – This means that the computer understands your program, but it aborts while it’s running because you are attempting to perform some operation that is impossible. In this case, the program will immediately halt during execution, and you will see an error message.

Common sources of run-time errors include trying to open a file that does not exist, dividing by zero, taking the square root of a negative number, etc. As the name suggests, a run-time error occurs while the program is running. As with syntax errors, the Python system will tell you what kind of error you have, and where in the program it thinks the error has occurred.

**Logical error** – This means that the program runs to completion, but the output is incorrect.

Logical errors usually result when we mistype a formula or when we print the wrong value. Logical errors are usually the most difficult kind of error to fix because the computer does not give you an error message. You must logically figure out where the error came about.

Exercise: The following code attempts to determine if a list contains an odd number. What is wrong with the following code, and how would you fix it?

```python
list = [8, 5, 2, 8, 3, 6, 1, 9, 4]

for number in list:
    if number % 2 == 1:
        found = True
    else:
        found = False
```

Some years ago, on the TV game show *Who Wants to be a Millionaire*, the million dollar question asked “Which insect shorted out an early supercomputer and inspired the term ‘computer bug’?” The possible choices were: moth; roach; fly; Japanese beetle. The answer was a moth.
Remember: An error is simply a mistake in a program. Computer programs must be written with great care because they are brittle. A trivial change in the code can make a big difference in how the program behaves. It is very easy to make a mistake. One small change can fix or ruin a program. Don’t make it a rush job. For example, when editing, it is easy to omit a statement, duplicate a statement, type statements out of order, or indent statements improperly.
FILE INPUT AND OUTPUT

Ultimately, one goal of computer programming is to write interesting and useful programs, and then test them on realistic input data. For example, counting all the words in a book, or finding all the words that have exactly 9 letters in them, or the words that have 3 vowels, etc. “Interesting” programs tend to be those that have large amounts of input and/or output.

Output

If you have a program that needs to print out a lot of output, it’s a good idea to instead write all the output to a file so that you can view it later. By “a lot” of output, I would say more output than you can easily see on a screen. When we perform file output, we actually have our computer program create a new file to contain the output. It turns out that a file is its own data type, meaning that we can create a variable in a Python program to refer to an actual file on the hard drive. So, now he have learned six data types: int, float, str, boolean, list, file.

Here are the steps to accomplishing file output, followed by an example program.

1. First, create the output file by using the built-in open( ) function. You generally should do this early in the program. You are free to choose whatever names you want for the name of the file, and the name of the variable referring to the file within your program. For example:
   ```python
   outFile = open("output.txt", "w")
   ```

   Note that the 2nd parameter to the open( ) function is the letter “w” which stands for write. We are telling Python that we want to write to this file.

2. When you are ready to print something to the file, use the built-in write( ) function instead of the usual print( ) function. You need to specify the file variable as well as what string to write to it. As an example:
   ```python
   outFile.write("a modern major general\n")
   ```

   Note that it is a good idea to end the string with \n, which is the newline character. This will allow any subsequent output to appear on the next line.

3. At the end of the program, tell Python to close the file, using the built-in close( ) function:
   ```python
   outFile.close()
   ```

   By closing the file, we ensure that all of the output actually gets written to the file. Because of the way that the operating system usually handles files, if we forget to “close” the file, the last part of our output may get omitted from the file.
# output.py – Demo of file output
# Can we handle a lot of numbers?

# Create an output file.
outFile = open("numbers.txt", "w")

# Put the calculated values into the file.
for i in range(14, 23, 2):
    outFile.write(str(i) + "\n")

# We’re done with the file.
print("I am done writing to the file.")
outFile.close()

Exercise:

1. In the above example program, what numbers appear in the output file?

2. In the above program, when writing the number i to the file, why was it necessary to call str(i)?
   In other words, why couldn’t we say outFile.write(i + "\n")?

3. Write a program that prints the numbers 1-100 on the screen, one number per line.

4. Write a second program that prints the numbers 1-100 to a file. Call the file counting.txt.

Input

If your program needs a lot of input (e.g. adding 10 numbers), then rather than having the user type so much input each time the program is run, it is more convenient to use file input. The steps for accomplishing file input are analogous to what you saw for file output.

1. First, open the input file.
   inFile = open("input.txt", "r")

2. We read lines from the input file one by one. Each line is a string, just like the interactive input() function would provide us.
   for line in inFile:
       # do whatever you need with the input

3. When done reading input, close the file.
   inFile.close()
So, now that you have seen both file input and file output, you should notice that the structure is the same. There are three steps: opening the file, reading or writing the file, and then closing the file.

Exercise:

1. Create a text file called input.txt, and enter 10 numbers, one per line. This will be our test input.

2. Based on the steps listed above, write a program that reads input.txt and finds the sum of all the numbers in the input file. Your program should be written in such a way that it does not matter how long the input file is. In other words, don’t tell your program to read only the first 10 numbers in the input file.

3. How would you find the average of the numbers in input.txt?
PROBLEM SOLVING

A computer system consists of two essential parts: its hardware and software. “Hardware” generally refers to the tangible pieces that you can touch: the actual physical components such as the CPU (central processing unit), memory, I/O devices, and the network. “Software” on the other hand consists of the programs stored on the computer that we want to run. The software is what gives the machine its behavior, and allows it to do useful work (or play). Software includes the operating system, which is responsible for maintaining the hardware in good working order. Besides the operating system is the application software that we use on a regular basis like Web browsers and Microsoft Office.

To illustrate the difference between hardware and software, think of a restaurant. In one sense, when we “go to a restaurant,” we are thinking of the building location that houses the restaurant. In another sense, when we are “inside the restaurant,” we are enjoying the food and its atmosphere, which are a product of the restaurant as a business. Someday the restaurant (business) might move to a new location (building). It’s the same food, employees, clientele and atmosphere, but in a different location. The old location has been taken over by a different business, though its exterior still looks the same.

In this course and in several other courses offered by the computer science department, you learn how to write your own software, so that you can have the computer do exactly what you want and solve your specific problems. We are generally not interested in designing new computer hardware – that is closer to the subject of computer engineering, rather than computer science.

The heart of any computer program is its algorithm. The algorithm tells us in English how we go about solving the problem. As mentioned before, there are three important features of any successful algorithm:

- **Unambiguous:** This means that anybody should be able to follow your directions.
- **Detailed:** The algorithm must be precise in its description of how the input, output, variables, and operators are to be used.
- **Deterministic:** The order in which the steps are taken must be absolutely clear. When one step is completed, it should be obvious what the next step in the recipe should be.

**Problem-solving strategy**

Whenever we wish to solve a problem in computer science, it is important to adhere to a general problem-solving strategy. The following table shows how I like to remember it, with these five steps. Beside each step, I have indicated a common obstacle that could occur at that point in the process.
<table>
<thead>
<tr>
<th>#</th>
<th>Problem-solving steps</th>
<th>Common obstacle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Understand the requirements of the problem, including its inputs and outputs.</td>
<td>The requirements are vague or ambiguous, or otherwise we don’t understand what the problem is asking us to do. In this case, we seek clarification.</td>
</tr>
<tr>
<td>2</td>
<td>Write the solution in English “pseudo-code”. This step is the most important, and this is where we need to invest most of our time.</td>
<td>We understand the problem, but we don’t know how to solve it!</td>
</tr>
<tr>
<td>3</td>
<td>Write code in a programming language, such as Python.</td>
<td>We know how to solve the problem by hand, but we don’t know enough of the programming language to adequately convey our solution.</td>
</tr>
<tr>
<td>4</td>
<td>Compile the program. Note that the word “compile” means for the computer to translate your program from whatever language you typed it in, into machine (binary) language, which is the native language of the computer.</td>
<td>After we typed in the program, it does not compile because there are syntax errors. We need to re-edit the program.</td>
</tr>
<tr>
<td>5</td>
<td>Run and test the program.</td>
<td>The program compiles, but when we run it we (sometimes) get a run-time error or the output is incorrect. This probably means we made a mistake back in step 2 because our algorithm is flawed or incomplete.</td>
</tr>
</tbody>
</table>

Beginners often have trouble with this procedure because they try to do steps 2 and 3 at the same time. It is easier to type the computer code once you already have a good idea on how to solve the problem. Otherwise, you will spend lots of time editing your code and making mistakes.

After a program has been tested and it seems to work for all conceivable inputs, then we can decide to refine it further. For example, we may want to make the output more attractive, or make the input process more user-friendly. Or we may want to generalize the algorithm to solve a larger variety of input situations.

Let’s look at some examples of algorithms.

Example #1: The Euclidean algorithm. Given two numbers, it finds their greatest common divisor.

Algorithm:

1. Let m = the larger number, and let n = the smaller number.
2. Let r = the remainder after dividing m/n.
3. If r = 0, then our answer is n, and we are done. But if r ≠ 0, let m = n, n = r, and go to step 2.
Let’s try out this algorithm. Do you understand the steps? Will the procedure work? The algorithm here is rather tersely stated. It said nothing about I/O, and if we ever wanted to write this into a computer program, we would have to fill in those details.

Exercise: Work out the Euclidean algorithm above with the numbers 32 and 84. Does the algorithm give the correct answer?

Example #2: Convert time zones. Given the time here, add 9 hours to output the corresponding time in Moscow.

Algorithm:

1. Ask the user to enter the hour, the minute, and whether it is a.m. or p.m.
   Throughout this algorithm, the minutes will be unchanged.
2. Convert this civilian hour to a military hour:
   If it’s 12 a.m., the military hour is 0
   If it’s 12 p.m., the military hour is 12
   Otherwise, if it’s p.m., the military hour is 12 + the civilian hour.
   And if it’s a.m., the military hour is the same as the civilian hour.
3. Ready for the conversion. Add 9 to the military hour.
   After doing so, if the military hour is 24 or higher, subtract 24 because it is referring to a time in the next day.
4. Convert the hour back to civilian format. In each case, print the minute between the hour number and the a.m./p.m.
   If the military hour is 0, output 12 a.m.
   If the military hour is 12, output 12 p.m.
   Otherwise, if the military hour > 12, subtract 12 and say “p.m.”
   Else, print the military hour unchanged, and say “a.m.”

Exercise:

1. How would this algorithm change if we wanted to convert to a time zone in the other direction? For example, Los Angeles is 3 hours behind.

2. How would this algorithm change if we wanted to convert to a time zone that is not a whole number of hours different from ours? For example, the time in Newfoundland is 1.5 hours ahead.

Example #3: Allow the user to play a simple guessing game.
For example, we could have the user guess a number between 1 and 100. We need to set the maximum number of guesses to be $\log_2 100$, which is about 7. The secret number can be hard-coded in the program, or better yet, we can use a built-in function to select a random number. However, it may be easier to test our algorithm if we hard code the correct answer.

Algorithm:

1. Let the secret number be 42.
2. Let the maximum number of allowed guesses to be 7.
3. Initialize the number of actual guesses accrued to be 0.
4. While the number of guesses < maximum guesses, loop:
   a. Ask the user for a guess
   b. Increment the number of guesses made
   c. If the guess matches the secret number:
      Win! Congratulate the user.
      Break from the loop, as the game is over.
   d. Otherwise, at this point, we know the guess is wrong.
      If the guess > secret number,
      Tell the user the guess was too high.
      Otherwise,
      Tell the user the guess was too low.
5. If we get this far, that means the user has run out of guesses and has lost the game.

Discussion:

How does the above algorithm determine that the player has won or lost the game?

When you write your own algorithms, it is not necessary to number your steps. After all, as you compose your algorithm, you might suddenly realize that you need to insert a step between two existing steps, and it would be inconvenient to have to renumber the steps.

Some advice

The best way to learn how to solve problems is to practice. But it doesn’t hurt to take a break for a few minutes and listen to some advice. 😊

To motivate a solution to a new problem, it sometimes helps to look at existing example solutions. After a while, as you accumulate some experience, you will begin to say to yourself, “This problem looks familiar.” Most problems are not solved entirely from scratch. You can adapt a technique that you saw from an earlier problem. It’s just like making sandwiches. If you know how to make a peanut butter and
jelly sandwich, it’s not hard to figure the recipe for a peanut butter and banana sandwich. And the procedure for a turkey sandwich is almost the same as for a ham sandwich.

Use built-in features of Python to simplify your solution. You can search the online documentation, and maybe you will find a very useful function. We have seen examples of this already. For example, there is already a built-in function in Python that will find the sum of the numbers in a list.

One major skill in computer problem solving is being able to read a problem description, and identifying the major “nouns” and “verbs.” The nouns often become the variables in the program. The verbs could be various operations or functions.

It has been said that any skill takes about 10 years to master. In a college course you only have a few months. You won’t be expected to solve every possible problem at this stage in your career. So, be patient with yourself and have some fun.

Problem solving is not easy. There is no formula that works in all cases. Difficulties can arise at any stage in the process. Ask for help if you get stuck.

It’s easier to find a mistake in the (English) design of a program than in the (Python) implementation. This is why we spend so much time in step #2 in the problem-solving procedure. It’s a common mistake for people to rush to step #3 and type the code before having a complete algorithm.

Input/output (I/O) is an important part of computer programs. To create an effective program, we sometimes have to use the right kind of I/O.

- Examples of input include: numbers, text, files, a URL, a button that the user can click, and image and sound
- Examples of output include: numbers, text, files, images and sound

The output is what the user sees when running your program. You need to take care that the output is in the exact specification that the user expects. For example, suppose you want to print an amount of money. An output of $3.57777777 does not look good. Actually, in this course, this situation won’t come up, but in the next course you will learn how to format output.

**Practice our problem-solving strategy**

Find a partner for this exercise. Your group will be given one or more of these problems to solve. Please write a very clear list of steps to solve the problem. You will discuss the solutions in class. Try your best, and don’t be concerned if your first attempt at a solution is not correct. Please note that you are not writing computer programs here. You only need to write an algorithm, which is a step-by-step solution in English.

Hint: I think you will find that several problems will have this basic structure: a loop containing some kind of if statement, so that you can “interrogate” each number in a list or each letter of a string.
1. Given a list of numbers, print the locations (i.e. indices) of the numbers that are multiples of 5. Also tell the user how many numbers it found.

2. The Fibonacci sequence is this list of numbers: 1, 1, 2, 3, 5, 8, 13, 21, etc. The first two numbers are 1 and 1, and each subsequent number in the list is the sum of the previous two. Create a list containing the first 20 Fibonacci numbers.

3. Ask the user for a string, and print out every fiftieth character of the string. In other words, the characters at index 0, 50, 100, 150, etc.

4. Ask the user for the current time in Greenville. Determine the time in Seattle (3 hour time difference).

5. Solve a quadratic equation. Ask the user for the values of a, b, c in ax^2 + bx + c = 0, and determine the two answers. You may assume that the roots are real.

6. You are given a list of 10 integers. Go through the list and see if it contains any values divisible by three. If so, print out all such values, along with their locations in the list.

7. You are given a list of 5 numbers. Determine the second largest value in the list. Don’t change the list.

8. You are given a list of 5 numbers. Determine if this list is sorted in ascending order.

9. You are given a list of numbers. Find the smallest positive number in the list, as well as its location.

10. You are given a list of numbers. Determine how many values are unique. For example, the list of numbers [1, 7, 4, 1, 4] has three unique values.

11. Ask the user to enter a sequence of numbers. Store these numbers in a list. Stop reading input once you encounter the same number entered twice in a row.

12. Ask the user to enter a sequence of numbers. Store these numbers in a list. Stop reading input once you encounter a number that has already been entered.

13. Ask the user for a word. Determine if the word contains a double letter. A double letter means that you have 2 consecutive characters that are the same. Hint: a word is simply a list or sequence of characters. If you feel more comfortable with numbers, this problem is the same as checking to see if 2 consecutive numbers in a list are the same.

14. From a deck of cards, deal yourself 7 cards. Determine how many of these cards are face cards (Jack, Queen, King).
15. Given a poker hand, determine if it’s a full house.

16. Output the first twenty powers of 2. \((2, 4, 8, 16, \ldots, 2^{20})\) Also find the sum of these numbers.

17. Ask the user to enter the lengths of 3 sides of a triangle. Determine if this triangle is equilateral (all sides equal), or isosceles (two sides equal) or scalene (no sides equal).

18. Ask the user to enter the 3 sides of a triangle. Determine which side is the longest and call this side “c”, and call the other 2 sides “a” and “b”. Determine if this is a right triangle by testing if \(c^2 = a^2 + b^2\).

19. Ask the user to enter a number. Print all the divisors of this number. For example, if the user chooses 10, then print out the numbers 1, 2, 5 and 10. Also tell the user how many divisors.

20. Print out all the prime numbers between 1 and 1000.

21. Given a number of seconds (which could be a large number), output the equivalent time in minutes:seconds, or hours:minutes:seconds as appropriate. For example, if the input is 200, then we would say this is 3 minutes and 20 seconds.

22. Suppose s is a string variable containing a time, such as “6:17”. There is an integer on either side of a colon. Assume this represents minutes and seconds. Output the total number of seconds this represents. For example, in this case 6:17 equals 377 seconds.

23. Ask the user to enter the date, in the form of month/day. Determine if this is a valid date. For example, 1/45 is invalid. You may assume it is not a leap year.

24. A computer system has a rule about user names. They must be 8 characters long, and must contain at least 2 letters and at least 2 digits. Given a proposed user name, test if it’s valid.

25. Ask the user for an integer, and determine if it’s a 4-digit number or not.

26. Given a positive integer, how would we figure out how many digits it has?

27. Given a positive integer, how would we find the sum of the digits?

28. Ask the user to enter a string. Determine if this input is a valid identifier in Python. In other words, the first character is a letter or underscore, and each other character is a letter, underscore or a digit.

29. Suppose s is a string variable. How would you capitalize all of the vowels in this string?
30. Suppose s is a string containing a first and last name. Print out this name, with the last name followed by the first name. For example, if s == “Al Capone” you should print “Capone Al”.

31. A bank would like to devise a scheme for assigning account numbers so that one of the digits, say the last digit, is a “check digit.” The purpose is to help prevent someone accidentally referencing an invalid account number. The check digit should be computable from the rest of the digits of the number. And it should not be as trivial as saying “all account numbers must end in 3.” Suggest a scheme for computing the check digit, and write an algorithm that will calculate it when given an account number.

32. (Challenging problem) I would like to schedule a meeting in Room 106 in Riley Hall. Scan the current course schedule. Find out when the room is being used for a class.

33. Suppose a company compiles a report on all of its employees. The report is formatted so that there are always 4 employees listed per page. The pages are numbered sequentially, starting at 1. The employees are number sequentially, starting at 1, and appear in ascending numerical order throughout the report. Write formulas that will determine the following:
   a. If there are n employees, how many pages will be in the report?
   b. On which page will we find the report for employee number n?
   c. Page n contains the reports for which employees? In other words, give the lowest and highest numbered employee on page n.
   d. Let’s generalize these formulas. How would your formulas change if we formatted the report to have P employees per page?

34. (Challenging problem) Suppose s is a string containing numbers separated by commas. For example, s might be the string “5,12,8,3,50”.
   a. Create a list called commaLocation that contains the indices of all of the commas in s. Your algorithm should work no matter how many numbers are in s.
   b. Next, create a list L that consists of the numbers inside s that are separated by the commas. Use the values in commaLocation to help you create string slices.
   c. Print the sum of the numbers in L.

35. Ask the user for a car’s city and highway mileage (miles per gallon), and use this information to compute the average MPG. The definition of “average” MPG assumes that 55 percent of all miles are city miles, and the remaining 45 percent are highway miles. Hint: Assume that we drive 100 miles. Therefore, we drive 55 miles in the city and 45 on the highway. The average MPG is the total number of miles driven (100) divided by the total number of gallons consumed in the city plus the highway. In your solution, you need to calculate the number of gallons consumed driving 55 miles in the city, and the number of gallons consumed driving 45 miles on the highway.
FUNCTIONS

Objectives:

• Why functions exist
• What happens when we enter and leave functions
• Parameters and return values
• How to create and use a function

As you know, a computer program is a list of instructions for the computer to perform in order to solve some problem. But, what happens if we have “a lot” of instructions? It’s just like what happens when any business or group of people has a lot of work to do – they get organized and delegate the work.

A restaurant kitchen is a busy place. Some chefs specialize on just one type of food, such as sauces or desserts. A menu is often a recipe of smaller recipes. The same thing occurs in the software industry. A large computer program is written by several people, and each team member is responsible for writing a portion.

Functions provide us a way to organize a solution into logical pieces that communicate with each other. A function is basically one self-contained part of a computer program. Over the years, computer scientists have used various words to describe this concept: besides the word “function” you may also hear of a “procedure,” “sub-program,” or “method.” They all essentially mean the same thing, and the nuances are not important at this point.

Another purpose of a function in a computer program is to encapsulate some code that might need to be used several times. We avoid having to retype the same code in more than one place in the program.

Here is an analogy to illustrate: When you hear a song, there is usually a portion that gets repeated several times, and this turns out to be the most memorable part of the song. It’s called the chorus. And when you see the lyrics of the song printed out or as a computer file, you can see the words for the chorus printed near the top of the lyrics. However, between each of the verses of the song, there may be a special notation [Chorus] in brackets. This means “the chorus goes here.” Why did the person who typed the lyrics decide to write [Chorus] instead of typing out the lyrics of the chorus again?

The reason is to save printed space. It is unnecessary to retype the chorus because it doesn’t change during the song. When you are first learning the song, and you encounter [Chorus] in the lyrics, all you have to do is glance back at the top of the sheet to find the words to the Chorus. And then what do you do when you are finished with the chorus a second time? You need to remember where you left off in the rest of the song.

Exactly the same thing is going on in computer programs that contain functions.
Basic concepts regarding functions

Actually, you have already been using Python’s built-in functions. Now we get to write our own. The first thing to understand about functions is where they fit into your program.

In Python, we *define* functions at the top of a program. After defining all the functions we wish to write, this is followed by the “main program.” When you run a program, it begins with the first statement of the main program. We only enter a function if and when it is called. Most of the time, a function is called from somewhere in the main program. But it is possible for one function to call another.

When you write a function, it is important to preface it with a comment, explaining its purpose, and how it works, just like you would include a general comment about an entire program.

Functions often communicate data with the main program. Usually we pass *parameters* to a function. Parameters are like a form of “input” to a function. Inside the function, we are free to create any “local” variables to assist us in our calculations. Finally, a function usually needs to *return* some data back to the place where the function was called. A function can have any number of parameters, but it can only return one value. However, in Python, we can return a list, so this is our loophole in case we actually need to return more than one number.

There is one subtlety to note about parameters inside a function. Any changes we make to them inside the function will have no effect once the function is finished. Consider this example. The function and main program are written here side by side for clarity, though in reality the function would be on top.

```python
# main program                 def fun(n):
    a = 4                              n = n + 1
    fun(a)                             print(n)
print(a)
```

In this example, since \( a = 4 \), the value 4 is passed to the function into the parameter \( n \). So, inside the function, the value of \( n \) is initially 4. Then we increment it to 5 and print this out. When the function returns to the main program, \( a \) is still 4. So, when we print \( a \), we print the number 4. Incrementing \( n \) inside the function had no effect on \( a \) in the main program.

<table>
<thead>
<tr>
<th>Main program calls a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>We may pass to it some parameter(s).</td>
</tr>
<tr>
<td>We enter the function.</td>
</tr>
<tr>
<td>Parameters coming in will act like local variables.</td>
</tr>
<tr>
<td>When we are finished, we may return a value.</td>
</tr>
<tr>
<td>After the function returned, if there was a return value, we need to use it here.</td>
</tr>
</tbody>
</table>
Let’s look at some examples. In each case, the function being defined appears in boldface.

Example #1

# A program that can count multiple times.

# count_up – This function will print the numbers
# from 1 up to some given maximum.

def count_up(maximum):
    for i in range(1, maximum + 1):
        print(i)

# Main program: let’s count to 10, then to 20.
count_up(10)
count_up(20)

Example #2

# Let’s find the sum of some ranges of integers.

# find_sum – find the sum from low..high

def find_sum(low, high):
    sum = 0
    for i in range(low, high + 1):
        sum += i
    return sum

# Main program
print(find_sum(10, 25))
print(find_sum(4, 72))
print(find_sum(75, 150))

Discussion: For each of the two above example programs, answer these questions.

1. In the example program above, what is the name of the function?
2. How many times is the function called?
3. Does the function take any parameters? If so, how many, and what is/are the type(s) of the parameters?
4. Does the function return a value? If so, what kind of value?
Please note that there are two parts to writing any function:

First, we need to define the function itself. We accomplish this by using the Python keyword `def`. As we design the function, we need to consider the following.

- Does the function need parameters? If so, how many, and what should they be called?
- What calculation does the function need to do? Is any I/O necessary?
- What should the function return, if anything?

The second step is to actually use the function elsewhere in the program. In other words, somewhere in the main program we should call the function. Here are some considerations.

- What values should we pass to the function?
- What should we do with the answer that the function returns to us? For example, it turns out that a function call is often on the right side of an assignment statement.

Example #3: We convert a Celsius temperature to Fahrenheit using a function.

```python
# Conversion function
def convert_to_f(celsius_temp):
    return 9/5 * celsius_temp + 32

# -----------------------------
# Main program
# 
c = float(input("Enter a Celsius temperature: "))

f = convert_to_f(c)

print("The Fahrenheit temperature is ", f)
```
Parameters and return value

A function may or may not need parameters. And it may or may not return a value. Therefore, there are 4 possible scenarios that we should consider. They are all plausible, but each one describes a completely different purpose of the function.

1. Parameter(s) and return value
   Send data, receive data
   This is the most common scenario.
   The function is calculating something for the main program.

2. Parameter(s) but no return value
   Send data, but receive no data
   Most likely because we want to output something or write to a file while inside the function. The function is not trying to perform a calculation that we need later.

3. No parameter(s), but a return value
   Send no data, but receive data
   Most likely because the function is getting data from the user, an input file, or is generating random data – and this is needed by the rest of the program.

4. Neither parameter(s) nor return value
   Send no data, receive no data
   No data to be sent in either direction. Usually this means we are just printing a message, or we are doing some isolated step of an overall algorithm.

What does a function call look like? It depends on whether the function you are calling returns a value. If not, then the function call just sits by itself. Otherwise, you need to do something to the return value. “Do something” means that the function call needs to be on the right side of an assignment, or it needs to be nested inside another function call as a parameter.

Here are some examples of what function calls look like:

<table>
<thead>
<tr>
<th>Function call</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(25)</td>
<td>Go to the function called fun() and take the number 25 with you. Apparently, there is no return value to bring back.</td>
</tr>
<tr>
<td>x = fun(25)</td>
<td>The value returned by fun(25) is put into the variable x.</td>
</tr>
<tr>
<td>print(fun(25))</td>
<td>The value returned by fun(25) is printed.</td>
</tr>
<tr>
<td>fun2(fun(25))</td>
<td>The value returned by fun(25) is passed to a second function fun2().</td>
</tr>
</tbody>
</table>

In the above examples, notice that the first one is different from the other three. All we do is pass the number 25 to a function called fun. We don’t expect the function to return anything. If it did, it would be thrown away! In the other three examples, we do something with the return value.

It is a common mistake to write a function call like “fun(25)” when the function in fact returns a value.

The following example program illustrates the mistake of discarding a return value. It contains a function that performs a simple calculation.
# for a given value, return 7 less than 3 times the square.
def f(x):
    return 3 * x ** 2 - 7

# main program
value = int(input("Please enter a number: "))
f(value)
print("After applying the function, the answer is ", result)

This program will not run at all. It contains an error because the variable result was never assigned. The way to fix the error is to change the statement f(value) to this:

result = f(value)

Recap

Let’s review some important facts about functions:

- The parameters and return value are optional. It all depends on the purpose of the function.
- Variables declared inside a function are “local” only, and cannot be used outside the function. Similarly, changes to variables inside a function have no effect outside the function.
- You can only return one value from a function, but it can be anything, including a list.
- We have already been using built-in functions like print().
- The function is introduced with the word def, and the body of the function is indented. Unindenting signals Python that the function is over.

Any of the practice problems we saw earlier can be written as a function instead of a program. Try it! Let me suggest:

- Put the algorithm’s calculations inside the function. The function should not do I/O itself. The function’s “input” is really the parameters, and its “output” is the result it returns. If you’d like, you can write a separate function to obtain the interactive input from the user and/or a separate function to display the output.
- The main program should pass the appropriate (input) data to the function. And most importantly, it needs to use the function’s return value to output to the user.

Exercise:

1. Consider the following Python program. Trace its execution by hand and explain what happens. What is the output?
def fun(x):
    return 2*x - 1

def fun2(x, y):
    return fun(x) + y

def getNumber():
    value = int(input("Please enter an integer: "))
    return value

# main program
a = 3
b = fun(a)
print(b)

c = fun2(a, b)
print(c)
d = getNumber()
print(d)

2. Consider the following Python program. Explain what it accomplishes.

def compute(n):
    return n + 2

# main program
L = [12, 87, 5, 13, 94]

for number in L:
    print("The result for ", number, " is ", compute(number))

3. Let’s create a short program that includes the use of a function. The program will ask the user to enter the freezing and boiling points of a substance. We need to convert these values from Fahrenheit to Celsius.

First, we need a conversion function. The parameter is the Fahrenheit temperature. The return value is the corresponding Celsius temperature.

The main program will do the following steps: It should ask the user for the freezing and boiling temperatures in Fahrenheit. Next, it needs to call the convert function on the freezing
temperature to convert it to Celsius. Then it should do the same for the boiling temperature. Finally, output both Celsius temperatures.

4. Design a program that converts an amount of money from US dollars into euros. Use a function to perform the actual conversion, and do all the I/O in the main program. Assume that 1 Euro is worth $1.16.

5. Go back to the 30+ practice problems you saw earlier. Select one of these problems, and write a program to solve it, using a function to perform the essential computation.

6. We can find hierarchical information and multiple “function instances” in everyday life. For example, the nutrition label for Publix’s Cool Mint Cookie frozen yogurt shows several levels of nesting. Literally, the ingredients read as follows.

Cultured lowfat milk, sugar, corn syrup, nonfat dry milk, whey protein concentrate, cultured dairy solids, chocolate fudge [sugar, peanut oil, cocoa (processed with alkali), whey, salt, soy lecithin], choco-coated mint cookies ** {cookie [wheat flour, sugar, partially hydrogenated soybean oil and/or cottonseed oil, cocoa processed with alkali, soy lecithin (an emulsifier), salt, sodium bicarbonate], sugar, coconut oil, cocoa processed with alkali, cocoa, nonfat milk, palm kernel oil, milkfat, natural flavors, soy lecithin, oil of peppermint, color added (yellow 5 lake, blue 1 lake)}, vegetable mono and diglycerides, natural flavors, locust bean gum, guar gum, sodium citrate, calcium sulfate, carrageenan and peppermint extract.

** manufactured in a facility that produces peanuts, tree nuts and milk products.

Copy and paste the above ingredient list to a new blank text file. Separate the list out so that only one ingredient or subingredient appears per line. Indent the ingredients to show the levels of nesting. The above ingredient list uses various types of grouping symbols such as parentheses, brackets and braces. Place each grouping symbol by itself on a line to make the hierarchy easier to read.

Which foodstuffs are mentioned more than once in the list of ingredients? This would be analogous to a function that is called from more than one place in a computer program.

Congratulations! This is the end of the first unit of the course. As we turn our attention to specific applications of problem solving, you will have more opportunities to practice your craft in the lab and learn a little more Python as needed along the way.