**Conversational Python**

***An Introduction to Computer Science Written in Conversational Banter***

***as Opposed to Typical Stogy Old Textbook Tone***

**By Jason B. Shepherd, Ph.D.**

Table of Contents

[Chapter 1: Basic I/O, Variables, Types, Statements, and Expressions 3](#_Toc462995300)

[Section 1.1: the print statement 3](#_Toc462995301)

[Section 1.2: the input statement 4](#_Toc462995302)

[Section 1.3: assignment statements 7](#_Toc462995303)

[Section 1.4: values, types, expressions, and statements 11](#_Toc462995304)

[Section 1.5: Calling Functions 17](#_Toc462995305)

[Section 1.6: Comments 20](#_Toc462995306)

[Chapter 1 Exercises 21](#_Toc462995307)

[Chapter 2: Conditional Statements 25](#_Toc462995308)

[Section 2.1: the if statement and the Boolean type 25](#_Toc462995309)

[Section 2.2: elif and Boolean Operators 28](#_Toc462995310)

[Section 2.3: Nuances of Boolean Operators 33](#_Toc462995311)

[Chapter 2 Exercises 34](#_Toc462995312)

[Chapter 3: Loops and String Manipulation 36](#_Toc462995313)

[Section 3.1: while loops 36](#_Toc462995314)

[Section 3.2: manipulating and printing strings 42](#_Toc462995315)

[Section 3.3: for loops 49](#_Toc462995316)

[Section 3.4: more string manipulation with loops 52](#_Toc462995317)

[Section 3.5: nested loops 58](#_Toc462995318)

[Chapter 3 Exercises 60](#_Toc462995319)

[Chapter 4: Functions 63](#_Toc462995320)

[Section 4.1: Introduction to Functions 63](#_Toc462995321)

[Section 4.2: Parameters 63](#_Toc462995322)

[Section 4.3: Return Values 63](#_Toc462995323)

# Chapter 1: Basic I/O, Variables, Types, Statements, and Expressions

## Section 1.1: the print statement

Let us start with our first program. Open **IDLE**. You will see the Python Shell window. Choose **File 🡪 New File** from the menu. You now have two windows on the screen. One is still the Python Shell window, and the other is your new file. Go ahead and save this new file. It’s a good habit to always save your file right away to give it a filename. Call this file **hello.py** and save it somewhere *other than* the Python34 folder; perhaps you could make a new folder called **cs1** under your **Documents** folder, for example.

You are now ready to start typing **statements** into the file **hello.py**. These statements will make up a program. Each statement gives the computer an instruction. A statement might tell the computer to put something on the screen (which is called “printing”), it might prompt the user to type something, it might tell the computer to remember some information so that we can recall it and use it later on in the program, etc.

Any time from here forward that you choose the menu option **Run 🡪 Run Module**, your computer will hand your statements to the **Python interpreter**. The Python interpreter will take each statement, one at a time, and change the statement into machine code (which is basically a bunch of 1’s and 0’s that tell the computer to turn electrical circuits on and off… beep beep boop boop). If any of your statements put words or numbers on the screen, those will show up in the Python Shell window.

Let’s try to write a program that makes some text show up in the Python Shell window.

So, let’s type the following into your new **hello.py** file.

print("Hello")

Run your program using the menu option **Run 🡪 Run Module**.

What do you see? You should see in your Python Shell window the word *Hello* on a separate line. If you don’t, ask your friendly neighborhood programmer for help.

Cool. Why does this work, and can we break it and learn something in the process? The **print** statement allows us to put words and numbers on the screen, that is, in the Python Shell window. You type print, and then in parentheses, you put what you want to appear on-screen. What are those double quotes doing there? Let’s get rid of them and then run the program to see what happens. Replace your code with this:

print(Hello)

Notice the double quotes are gone. Run your program using the menu option **Run 🡪 Run Module**.

Oh no! We’ve broken our computer! Well, not really. The word **print** means something to Python—it’s part of the language—but *Hello* doesn’t mean anything to Python. The double quotes tell Python to actually print the text “Hello” onto the screen.

Let’s delete the one line we have in our file, and replace it with the following three lines.

print("Hello")

print("How are you?")

print("Goodbye.")

Run your code again and you will see these three lines appear on-screen in order.

What if you change the order of the statements?

print("Hello")

print("Goodbye.")

print("How are you?")

Notice we switched the order of the last two statements. Python will only *execute* your statements in the order you give them. Consider the following analogy: programs are to computers as recipes are to cooking. The order of the statements matter just like the order of steps in a recipe matter.

## Section 1.2: the input statement

Okay, let’s change our code again – just one line this time.

print("Hello, Steve.")

This code attempts to make our program more personalized, but it makes the bold (and unfortunate) assumption that the user’s name will always be Steve. What if we wanted to ask users what their names are, and then greet the user by name? How many statements would we need? We would need two: 1.) to ask for the name, and 2.) to greet the person using that name. Here’s a good first attempt.

print("What is your name? ")

print("Hello, name.")

What do you see? The code does ask for the user’s name (that’s good), but it does not give the user the ability to type anything in (that’s bad).

Let’s introduce a new type of statement called **input**. All programs take *input* from the user (possibly from the keyboard, a mouse, or something else) and produce *output* (usually information is *printed* to the screen, but the information could be placed elsewhere, too, like placed in a file or sent over the Internet to a Web site or something). The **input** statement will allow the user to type something in.

Let’s try this. Change the first **print** statement to an **input** statement.

input("What is your name? ")

print("Hello, name.")

Run this code using **Run 🡪 Run Module**. What happens?

Cool! The user (you) can now type in your name. But, then the program fails to address the person (you, again) using that name. Bummer.

We need to make the program *remember* the person’s name in the first statement so that it can be used later in the second statement.

Let’s change the first statement from this

input("What is your name? ")

to this

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

See the difference? We’ve put “**firstname =**” in front of the **input** command.

Here’s how it works. The **input** command retrieves the text the user types in from the keyboard. Then, we have to *store* it somewhere so that we can use it later on in the program. The word **firstname** is a **variable**. Variables are kind of like Post-It notes for the computer to help it remember numbers and text that are important to us. **Variables** store **values**.

We can choose to name the variable *almost* whatever we want. There are some rules for what you can and can’t name a variable. Variables must start with a letter or an underscore (\_). After that, they can include any letters, numbers, or underscores, but no other symbols. Variables cannot have spaces in their name. You also cannot name a variable one of the **reserved words** in Python. That is, there are commands that mean something to Python, like **if** or **while**. Always name the variable so that you’ll remember its name. Instead of

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

We could have typed

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

or

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

but for now we’ll stick with

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

Now, what about the second line? It’s still

print("Hello, name.")

We need to *use* the variable **firstname** to retrieve the value we stored. Change the **print** line to this

print("Hello,", firstname)

Run it. Voila!

Now, instead of putting one thing inside the parentheses after **print**, we’re listing two things, separated by a comma. The first is a text string “Hello,”. The second is the variable that stores the name. Print will print both of those things, separated by a space.

What if we don’t want spaces between the things we print? We’ll get to that later, too.

Awesome. Try some stuff. Try to break your code. Don’t worry, you can always change your code back. Our code so far should look like this.

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

print("Hello,", firstname)

Here’s one way to break your code. Change the second line by removing *first* from the variable name.

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

print("Hello,", name)

Run this code. What happens? Why do you think this happens?

Once you create a variable named **firstname**, it’s called **firstname** for the duration of the program. In the first line, we’re creating **firstname**. Then, in the second line, we try to use a variable called **name**, but there is no such variable named **name**. Python rightly vomits red text all over the screen.

It would be like if your name was Jorge and I tried to get your attention by yelling “Hey, Fred!” You wouldn’t know I was trying to get your attention.

Let’s try one more thing. Let’s go back to our original code, which looked like this.

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

print("Hello,", firstname)

Change the variable **firstname** in the second line by capitalizing the first letter. In other words, change **firstname** to **Firstname**, like this.

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

print("Hello,", Firstname)

Now run it and see what happens. You get an error, don’t you? **Firstname** and **firstname** are different variables. Python is a *case-sensitive*language. So we don’t make mistakes with mixing uppercase and lowercase in Python, we typically stick to lowercase.

## Section 1.3: assignment statements

As it turns out, we now know a lot about Python even though we likely don’t realize it. We know three kinds of statements.

1. **print** statements
2. **input** statements
3. *assignment* statements

An assignment statement is a statement that creates or updates the value of a variable. Our input statement in the previous example was also an assignment statement because it created the variable named **firstname**.

Let’s look at more examples of assignment statements. Consider the following. I live on an acreage, and we have a barn where cats tend to gather. We didn’t have to buy any cats; they just show up. It’s good they are around because they eat mice and we don’t like mice.

Suppose we want to store the number of cats we have in our barn at any given time, and suppose we currently have six cats. To store this information in a Python variable, we would type the following.

cats = 6

To experiment with what’s happening here, let’s add two more lines so that your code now looks like this.

cats = 6

print(cats)

print("cats")

Can you guess what will happen when you run this code? It is very important to be able to read code line by line to figure out in your mind what will happen. Later on, when you’re programming and something doesn’t work right, you’ll need to look at your code line by line to make sure that it makes logical sense. This code will print

6

cats

There are two print statements listed above. Since there are two print statements, we can reasonably assume there will be two lines that appear on the screen. The first statement makes a new variable named **cats** and assigns it the value **6**. The second statement prints the value stored in cats. The third statement prints the text string *cats* since there are double quotes around it.

If we hadn’t created the variable **cats** before printing the value of **cats**, Python would have puked red text again saying that it doesn’t know what cats is.

It’s worth noting that the first statement line **cats = 6** does not print anything to the screen. It’s just an assignment statement. It’s only job is to **define** a new variable. It does not print anything. Only **print** statements actually print anything on the Python Shell window.

Okay, let’s try to *break things* again! What happens if we switch the order of **cats** and the number **6** in the first line, like this.

6 = cats

Run it. Kablooey! Now we’ve learned a rule about assignment statements. Think of the **=** sign as being more like a left arrow 🡨. We say

cats = 6

you can read it like

cats 🡨 6

in that the value **6** is being *assigned to* the new variable **cats**. You cannot change the order of **cats** and **6**.

See? Computers are picky.

So, on the left-hand side of the equals sign in an assignment statement, we can give the name of a variable. If the variable doesn’t exist, it is created new. If the variable *does* already exist, the value of the variable is updated (or, overwritten – update and overwrite are synonymous). So what can we put on the right-hand side of the equals?

On the right-hand side, we can put any **expression** that produces a value. Here are some examples of **expressions**.

6

2 + 4

6 \* 1

8 - 2

12 / 2

12 – (3 \* 2)

The asterisk (\*) or star does multiplication. The forward slash (/) performs division. The double forward slash (//) performs whole number division by dropping the remainder. For example, 5 // 2 produces the value 2. The percent (%) gives us the remainder. This operation is also known as the *modulus*, or simply *mod*. For example, 5 % 2 produces the value 1. The +, -, \*, /, //, and % are known as **operators**. Operators take two expressions (which produce values) and produces a new value. Parentheses can be used to group expressions much like we do in mathematics. Also as in mathematics, operators have precedence. Expressions produce a value by evaluating the expression left-to-right while performing parenthetical expressions first, then multiplication and division, and finally addition and subtraction.

With this in mind, we could re-write this assignment statement

cats = 6

like the following, just for the heck of it.

cats = 12 – (3 \* 2)

To recap, the left-hand side (abbreviated **LHS**) of an assignment statement is the name of a variable. The variable will either be created (that is, **defined**) if it doesn’t already exist or updated (that is, overwritten) depending on whether the variable existed previously. The right-hand side (abbreviated **RHS**) is an expression that, when evaluated by Python, produces a value.

With all this in mind, let’s try something. What happens when you change our code so that it looks like this.

cats = 6

cats = cats + 1

print("You have", cats, “cats.”)

Remember how assignment statements work. They work in two steps.

1. We determine the value produced on the RHS.
2. That value is assigned to the variable on the LHS.

What is the value on the RHS of **cats = cats + 1**? When we reach this statement, **cats** is **6**. Thus, **cats + 1** is the same as **6 + 1**, which is **7**. Therefore, the assignment statement **cats = cats + 1** is essentially the same as saying **cats = 7**.

Visualize how this assignment statement works.

cats = cats + 1

cats = 6 + 1

cats = 7

Remember, the equals sign performs assignment of the value produced by the RHS to the variable on the LHS. Therefore, the equals sign is more like a left arrow that the mathematical equals sign that says “the thing on the LHS and the thing on the RHS have the same value.” The fact that you can type **x = x + 1** in Python looks yucky to mathematicians. Poor, poor mathematicians…

We will say more about the relationship between values, expressions, and statements in the next section.

## Section 1.4: values, types, expressions, and statements

Let’s modify the code example from the previous section. It is, after all, silly because you start with a fixed number of cats when, in fact, we could start with any number of cats. Let’s ask users for how many cats they have in their barn. Then, our program should calculate the number of cats they’ll have in the barn in a month’s time. For the sake of argument, let’s suppose the number of cats will increase by 4.

Such a problem would look like this on the Python Shell window.

How many cats do you have? **10**

In a month’s time, you will have 14 cats!

The **10** is bolded to show that the user is the one who types in the value 10.

Can you write this program? Give it a try. You know how to get input. You know how to produce output. You know how to create variables and perform calculations using expressions. Try to write this program.

(One attempt at a solution follows, but try to shield your eyes and don’t look at it right away. You won’t learn very well if you don’t try and fail every now and then.)

Here is an attempt that you might have made.

cats = input("How many cats do you have? ")

cats = cats + 4

print("In a month’s time, you will have", cats, "cats.")

This seems reasonable and logical, but it doesn’t work! You end up with an error. Let’s take a very close look at the error. It’s very important to understand how to read error messages. If you know how, they often tell you exactly what’s wrong.

Traceback (most recent call last):

File "/Users/shep/cs1/code/cats.py", line 2, in <module>

cats = cats + 4

TypeError: Can't convert 'int' object to str implicitly

The error states that the problem is on line 2 of your code. That’s helpful, but keep in mind line 2 is only where the error was *detected*. It’s possible that what caused the error occurred earlier in the code than line 2.

Note what the error message says: “Can’t convert ‘int’ object to str implicitly.” What does this mean? In order to understand what it means, we need to revisit some of the concepts we touched on in the previous section.

A **statement** can consist of one or more expressions. An **expression** is a piece of code that produces a **value**. Every value has a **type**. Consider the following example.

carrots = (7 + 3) \* 2

This is one assignment statement whose right-hand side (RHS) consists of two expressions. The first expression is

7 + 3

7 and 3 are values known as **integers**, which is just a fancy word for “whole number.” Sometimes in Python, an integer is called an **int**, for short. 7 is a value and its type is int. 3 is value and its type is int. Since 7 is an int and 3 is an int, add them together gives us the value 10, which is also an int. Thus, we could write the following about this expression.

|  |  |  |
| --- | --- | --- |
| Expression | Value | Type |
| 7 + 3 | 10 | int |

Remember, an expression is a piece of code that produces a value. Every value has a type.

If you’re not sure what the type of an expression is, you can find out by typing the expression into the Python Shell. If I entered 7 + 3, the Python Shell would output 10. If I entered **type**(7 + 3), the Python Shell would output “<class ‘int’>”. Try out the **type** command to see what different expressions and different values have as their type. Try type(7), type(7.52), and type("Hello").

The next expression for us to consider in our current example is (7 + 3) \* 2. We can see from this expression that expressions can consist of other expressions. We already know that 7 + 3 is an expression whose value is 10 and whose type is int. Thus, we can determine the following about this expression.

|  |  |  |
| --- | --- | --- |
| Expression | Value | Type |
| (7 + 3) \* 2 | 20 | int |

From this, we can determine that the variable carrots will have as its value 20 and its type will be int.

There is another number type named **float**. Floats are used to represent numbers that have a fractional part. The value 5.2 is an example of a float. Float values can be expressed in scientific notation as well. The value 3e2 is equivalent to 3 x 102, which is 300, for example. The value after the ‘e’ is the power of ten to which we multiply the first number.

Any time we type a specific value like 5 or 5.25 in a program, we call that value a **literal**. That is, 5.25 is a **float literal** because it is “literally” the value 5.25. It’s important to have the word “literal” in your programmer vocabulary.

Now let us consider a different example.

firstname = "Kanye"

lastname = "West"

fullname = firstname + " " + lastname

These three statements all involve text values. Text values have a special type called **string**. In Python, a string is called a **str** for short. If, in the Python Shell, I were to enter type("Kanye"), I would see the output “<class ‘str’>”. If I entered the first statement firstname = "Kanye" and then afterwards entered type(firstname), I would see the output “<class ‘str’>”.

There are four expressions in these three statements. They are

|  |  |  |
| --- | --- | --- |
| Expression | Value | Type |
| "Kanye" | "Kanye" | str |
| "West" | "West" | str |
| firstname + " " | "Kanye " | str |
| firstname + " " + lastname | "Kanye West" | str |

Some operators can work on strings, too. When we use the plus (+) on two string values, it “smashes” the two strings together to form a new string. Here, we are taking the first name and putting a space on the end of it. Then, we are appending the last name onto that new string that consists of the first name and trailing space. There is a geeky name for “smashing” two strings together to make a new string, and that name is **concatenation**. We would say that the plus (+) **concatenates** two strings.

Remember, an expression is a piece of code that produces a value. Every value has a type. Every variable has a value and a type.

Types and values are tremendously important in Python, and this is illustrated by our problematic code from earlier in this section. Recall that we had written the following.

cats = input("How many cats do you have? ")

cats = cats + 4

print("In a month’s time, you will have", cats, "cats.")

When we ran this code, we got an error on line 2 that told us “Can't convert 'int' object to str implicitly”. When line 2 tries to do the plus operation, it gets confused because 4 is an int and it thinks that cats is a str rather than an int. Think about it. Can you guess why? Look back at line 2. Look back at line 1.

The **input** command in line 1 retrieves characters entered from the keyboard. These characters could be letters, symbols, and/or numbers. Since the value placed into **cats** by **input** could be any of these things, the type of the value **input** gives us is **str**. Suppose the user typed a **5** at the prompt “How many cats do you have?” The initial value of the variable **cats** will be the **str** value **"5"** rather than the **int** value **5**.

It doesn’t make sense to Python to add a string and an integer. After all, what is reasonable to assume about the type and value of an expression like “Hello” + 32? In the above example, we need to *convert* cats from a string to an integer so that we can treat cats as an integer. Let us add a new line of code after the **input** statement.

cats = input("How many cats do you have? ")

**cats = int(cats)**

cats = cats + 4

print("In a month’s time, you will have", cats, "cats.")

The **int** command converts the value stored in cats to a string, and then it overwrites the value of cats to this new int value. Converting a value from one **type** to another is called **casting**.

When we write code, it is bound to have errors we need to correct. Sometimes, those errors make the program crash and we see an actual error message in red text on the screen. Other times, however, we don’t get a nice error message. Instead, the program appears to behave erroneously. Erroneous code is called a **bug**, and it is up to us to “debug” the program.

|  |
| --- |
| At some point, we’ll have a nice detour in this box about the origin of the term “debug.” It’s an amusing historical tale. For now, read this: <http://www.wired.com/2013/12/googles-doodle-honors-grace-hopper-and-entomology/> |

To understand bugs, consider the same code we started with earlier.

cats = input("How many cats do you have? ")

cats = cats + 4

print("In a month’s time, you will have", cats, "cats.")

Let’s change the code so that instead of adding 4 to the number of cats, we will double the number of cats by multiplying by 2. After all, when it comes to feral barn cats, this is a more accurate representation of what happens to cat populations.

cats = input("How many cats do you have? ")

cats = cats \* 2

print("In a month’s time, you will have", cats, "cats.")

The type of the variable **cats** will once again be a string unless we cast it to an integer. Try running this code without performing the cast. Type ‘3’ for the number of cats. What happens?

33 cats?! Good grief! As it turns out, Python allows string values to be repeated using the asterisk/star (\*) operator. The expression we’ve inadvertently performed is "3" \* 2, which is the same as "3" + "3", which is the same as "33". If we properly cast **cats** to an **int** before multiplying, we get the proper result and we have “debugged” the program.

It is important to understand what operators are available to us. Examine the Tables 1.1 and 1.2 for a more comprehensive list.

|  |  |
| --- | --- |
| **Operator** | **Usage** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| // | Integer division  Returns the whole number result only from division.  Example: 5 // 2 gives us 2 rather than 2.5. |
| % | Modulus, or mod  Returns the remainder from division.  Example: 5 % 2 gives us 1 since 5 divided by 2 is 2 remainder 1 |
| \*\* | Exponentiation  Returns the result of raising a number to a power.  Example: 2 \*\* 3 gives us 8 |

**Table 1.1:** Arithmetic Operators

|  |  |
| --- | --- |
| **Operator** | **Usage** |
| + | Concatenation  Example: "ab" + "cd" gives us "abcd" |
| \* | Repetition  Example: "-" \* 5 gives us "-----" |
| % | Formatting  The string format operator allows us to insert one string value into the middle of another string, which is known as the format string. The format string can contain any number of **format specifiers**, which are placeholders for the values to be inserted. See Table 1.3 for a list of format specifier examples.  Example 1:  Suppose we have a variable forks that contains the integer value 5.  "There are %d forks on the table." % forks  would produce the string "There are 5 forks on the table."  Example 2:  Suppose we have two variables first and last that contain string values "Bob" and "Barker".  "Hi, %s %s." % (first, last)  would produce the string "Hi, Bob Barker." Note that if we have multiple values to be inserted into the format string, we enclose them in parentheses and separate them with commas.  Example 3:  Suppose we have a variable ch that contains a single character ‘!’  "OMG%c%c%c" % (ch, ch, ch)  would produce the string "OMG!!!”. |

**Table 1.2:** String Operators

|  |  |
| --- | --- |
| **Specifier** | **Description** |
| %d | the value to be inserted is an integer |
| %5d | the value is aligned to the right across a 5-character column |
| %-5d | the value is aligned to the left across a 5-character column |
| %f | the value to be inserted is a float |
| %10.2f | the value to be inserted is a float aligned to the right across a 10-character column, and we should use two decimal places after the decimal point only |
| %s | the value to be inserted is a string |
| %10s | the value is aligned to the right across a 10-character column |
| %-10s | the value is aligned to the left across a 10-character column |
| %c | the value to be inserted is a single character |

**Table 1.3:** Examples ofString Format Specifiers

## Section 1.5: Calling Functions

Earlier, we said that we know three types of statements.

1. **print** statements
2. **input** statements
3. *assignment* statements

We have since learned other statements, for example, using **type** to determine the type of an expression or **int** to cast a str to an int. We have also been using the word *command* to refer to words like **print**, **input**, and **type** that seem to have important meaning to Python. In fact, these commands are actually called **functions**. It’s important to learn to speak like a programmer when you are writing code.

|  |  |
| --- | --- |
| **Instead of…** | **Programmers say…** |
| *use* a function | **call** a function |
| function *produces* a value | function **returns** a value |

For example, if we have the following code.

answer = input("Do you wish to continue (y/n)? ")

Programmers would say they are **calling** the input function, and the input function will **return** a string value.

All functions return a value, even something like **print**. Just for fun (wheee!), type the following into the Python Shell window.

var = print("Hello.")

Now, type var and press ENTER. Hmm, normally when we type the name of a variable or we type an expression into the Python Shell window, it tells us its value. We get nothing. Try to type the expression type(var) into the Shell. Aha! The variable var has a special type called NoneType. Every function call is an expression that returns some value, even if at least that value belongs to NoneType.

We can now simplify the list of statements we know about to these.

1. function call statements
2. assignment statements

From now on, we will refer to things like **print**, **input**, and **type** as functions rather than as commands.

The expressions that are placed between the parentheses after the function’s name are called **arguments**. Some functions take no arguments, some functions take one argument, and some other functions can take several arguments. Arguments are separated with commas. Arguments tell the function how to do its job. The code below shows different examples of how to call functions with differing numbers of arguments.

age = input("What is your age? ")

age = int(age)

age = age + 1

print("In one year, you will be", age, "years old.")

Lines 1 and 2 demonstrate calling a function with one argument. Line 4 has three arguments. The first is a string value, the second is a string variable, and the third is another string value.

Because functions *return* values, we can call one function and immediately give its return value to another function. Look at lines 1 and 2 above again. Since casting can be performed on any expression, we could do both the **input** and the **int** cast on one line, like this.

age = int(input("What is your age? "))

The **input** function is called first, and the result returned by **input** is then given to **int**, which casts the result from a string to an integer. Students are often puzzled by what appears to be “double right parentheses” at the end of the above statement. Note that the last right parenthesis matches the left parenthesis for **int**, and the next to last right parenthesis matches the left parenthesis for **input**. This is shown visually below.

age = int( input("What is your age? ") )

If the user were to type **18**, the code above would be executed and “transformed” through the following steps.

age = int(input("What is your age? "))

age = int("18")

age = 18

Just for fun (again: whee!!), let’s try out the string formatting operator (%) in the print statement above. Instead of this,

print("In one year, you will be", age, "years old.")

We could do this.

print("In one year, you will be %d years old." % age)

Here, the value of age gets inserted into the format string in place of the %d. There is no advantage to one way or the other, per se. There will be lots of ways to write code, though you should try to write code so that is *readable*. If you write code that is easy to read, it will be easier to change. Some programmers like the second way because it is easy to see the format of what the output will be, and it can be easier to control where the spaces go in the output.

Started with four lines of code.

age = input("What is your age? ")

age = int(age)

age = age + 1

print("In one year, you will be", age, "years old.")

Then, we “tweaked” the code so that, ultimately, it looked like this.

age = int(input("What is your age? "))

age = age + 1

print("In one year, you will be %d years old." % age)

As you program, you will develop your own coding style. You will want to decide which of the two blocks of code (or a combination of them) looks the most readable to you.

## Section 1.6: Comments

As we go forward in learning Python, our programs will get longer and more intricate. It may be helpful to annotate our code with short comments to remind us what our code does. Here is a (somewhat silly) example.

# Get the user’s age.

age = input("What is your age? ")

age = int(age)

# Tell the user his or her age one year from now.

age = age + 1

print("In one year, you will be", age, "years old.")

The lines that start with a # symbol are called comments. Any line that starts with # will be ignored by Python. Again, this is a somewhat silly example because our code is relatively simple and probably does not require comments.

Programmers will also use comments at the beginning of a code file to document what the program does. Here is an example.

# Program: tictactoe.py

# Programmer: Susan McConnell

# Description:

# This program allows users to play Tic Tac Toe against

# a computer opponent. Users choose the row and column

# to place their ‘X’ or ‘O’ on the game grid in each turn.

Because placing a # symbol at the start of a line hides the code from Python, another use of comments is to hide old code. Sometimes, we want to save old code without deleting it. This can occur when we’re not sure if new code we’re trying out is going to work, and so we may not want to lose our old code in case we need to go back to it later. For example:

#age = input("What is your age? ")

#age = int(age)

age = int(input("What is your age? "))

Python will not execute the first two lines because they are “commented out.” It will, however, execute the last line.

Another use for comments is to help us program. As human beings, we do not naturally think in code. Even experienced programmers struggle to think purely in terms of programming language code. One good way to program is to write comments first in plain English to help us organize our logic and thoughts, and then we write Python code beneath the comments. For example, we might start with:

# Get the user’s age.

# Print how old they’ll be in a year.

Then, we can fill in the details.

# Get the user’s age.

age = input("What is your age? ")

age = int(age)

# Print how old they’ll be in a year.

age = age + 1

print("In one year, you will be", age, "years old.")

Comments end up being very important later on in the book when we start creating our own functions (yes, we get to make our own functions eventually). Practice writing comments when you write your own code.

## Chapter 1 Exercises

1. Write a program that creates as its output a face on the screen by arranging different symbols, letters, and/or numbers. Here is an example.

\\|//

o o

v

~~~

1. What is the output of the following program?

a = 1

b = a \* 2

c = 2 \* b + 1

print(a)

print(b)

print(c)

1. What is the output of the following program?

a = 2

b = a \* 2

c = b \*\* a

a = c % 2

print(a)

print(b)

print(c)

1. What is the output of the following program?

a = 5

b = a // 2

c = a / 2

a = a % 2

print(a)

print(b)

print(c)

1. What is the output of the following program if the user enters a **2** at the first prompt and a **3** at the second prompt?

x = input("Enter a whole number: ")

y = input("Enter another whole number: ")

z = x + y

print(z)

Be careful. Try to type this program into IDLE and see what you get.

1. Write a program that asks users for two whole numbers. It should then add them and print the result. Here is an example of what the output should look like.

Enter a whole number: **2**

Enter another whole number: **3**

2 + 3 = 5

1. What is the output of the following program?

s = "John"

t = "Smith"

r = s + t

print(r)

1. What is the output of the following program?

s = "John"

t = "Smith"

print("%s, %s" % (t, s))

1. What is the output of the following program?

s = "John"

t = "Smith"

r = "%s, %s" % (t, s)

print(r)

1. What is the output of the following program?

s = "John"

t = "Smith"

r = "%s, %s"

print(r % (t, s))

1. What is the output of the following program? Write down the answer *exactly* how it would appear on-screen.

print("%s %s" % ("Item", "Price"))

print("%s %f" % ("Soda", 1.75))

print("%s %f" % ("Pizza", 2.00))

print("%s %f" % ("Hot Dog", 1.50))

print("%s %f" % ("Crab Legs", 30.99))

1. What is the output of the following program? Write down the answer *exactly* how it would appear on-screen.

print("%-10s %10s" % ("Item", "Price"))

print("%-10s %10.2f" % ("Soda", 1.75))

print("%-10s %10.2f" % ("Pizza", 2.00))

print("%-10s %10.2f" % ("Hot Dog", 1.50))

print("%-10s %10.2f" % ("Crab Legs", 30.99))

1. What is the output of the following program? Write down the answer *exactly* how it would appear on-screen.

print("%-10s %10s" % ("Item", "Price"))

fmt = "%-10s %10.2f"

print(fmt % ("Soda", 1.75))

print(fmt % ("Pizza", 2.00))

print(fmt % ("Hot Dog", 1.50))

print(fmt % ("Crab Legs", 30.99))

1. What is the output of the following program? Write down the answer *exactly* how it would appear on-screen.

print("%-10s %10s" % ("Item", "Price"))

fmt = "%-10s %10.2f"

print(fmt % ("Soda", 1.75))

print(fmt % ("Pizza", 2.00))

#print(fmt % ("Hot Dog", 1.50))

#print(fmt % ("Crab Legs", 30.99))

print(fmt % ("Pretzel", 1.50))

print(fmt % ("Nachos", 2.25))

1. Write a program that asks the user for a single character. The program should then greet them in block letters "HI" consisting solely of that character. Here is an example of a running program.

Enter a character: **#**

## ## #####

###### #

## ## #####

Here is another example of a running program if the user were to type a different character.

Enter a character: **@**

@@ @@ @@@@@

@@@@@@ @

@@ @@ @@@@@

# Chapter 2: Conditional Statements

## Section 2.1: the if statement and the Boolean type

Do you remember our sample program involving cats in Chapter 1? Let’s re-write it here.

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

Feral barn cats can multiply rapidly if not spayed or neutered, so this is not that ridiculous of an example. Suppose we want our program to make the observation “That’s a lot of cats!” if the barn owner ends up with more than 20 cats. Our first attempt at this, based on what we know so far, might look like this.

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

print("That’s a lot of cats!")

Python programs always execute in sequence. That is, first Python executes line 1, then line 2, line 3, and finally line 4. With that in mind, line 4 will always execute no matter what. This is absurd given what we are trying to accomplish, and even more absurd if the user types 0 for the number of cats initially.

What we need is a way to say “only print if the number of cats is 20 or more.” Python allows us to do this by using something called an **if** statement. Let’s modify our code as follows.

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

if cats > 19:

print("That’s a lot of cats!")

Notice how we’ve added a line just before our final print statement. Note also how our print statement is now tabbed over one tab-stop. Now try out your program using several different inputs for cats. It works!

To understand more about why it works and how **if** statements work in general, let’s add one more line of code (for now).

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

if cats > 19:

print("That’s a lot of cats!")

print("Thank you for using our program.")

Try out your program using several different inputs for cats again. Notice how no matter how many cats you enter, the program always prints “Thank you for using our program.” Now, let’s tab the last line over one tab-stop so that it lines up with the previous print statement.

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

if cats > 19:

print("That’s a lot of cats!")

print("Thank you for using our program.")

Try out your program again.

Notice how now the program only prints “Thank you for using our program.” if the value of cats is greater than 19. This tells us something about how **if** statements work. When the expression cats > 19 is true, any statements that are indented beneath the **if** statement are executed, but they are only executed if the expression cats > 19 is true. A sequence of statements separated together by indentation (i.e., tabs) is called a **block** of statements.

Let’s try one more thing to understand how tabs/indentation work. What happens if we de-indent the first line under the **if**?

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

if cats > 19:

print("That’s a lot of cats!")

print("Thank you for using our program.")

Run your code. Kablooey! The error indicates that whenever Python sees an **if**, there should be at least one indented section of code beneath it.

Earlier, we referred to cats > 19 as an expression. If it is an expression, then it must produce a value and must have a type, right? Let’s go to our Python Shell window and type the following: type(cats > 19). The result we are given is something called a **bool**. The word “bool” stands for Boolean. The word Boolean refers to George Boole, an English mathematician who formalized mathematical logic. Valid Boolean values are either **True** or **False**.

Whenever the value of the Boolean expression given to an **if** statement is **True**, the indented block of code beneath it will be executed. If it is **False**, the block will be skipped.

Based on what we know about variables having types, we might conclude that we can create Boolean variables, too, and we’d be right! Examine the following code.

cats = int(input("How many barn cats do you own? "))

cats = cats \* 2

print("In a six-months, you will have %d cats!" % cats)

too\_many = cats > 19

if too\_many:

print("That’s a lot of cats!")

print("Thank you for using our program.")

Now, suppose we want to say “You should get more cats” if the barn owner does not yet have 20 cats. That is, we want to say this message if the Boolean expression is **False**. We can adjust our code by changing this:

if cats > 19:

print("That’s a lot of cats!")

to this:

if cats > 19:

print("That’s a lot of cats!")

else:

print("You should get more cats.")

Any **if** block can have an **else** block to handle situations where the **if** condition is **False**, but having an **else** block is optional.

Boolean expressions can be formed using **logical operators**. We’ve already seen greater-than (>). Table 2.1 shows the rest of the logical operators in Python. Note that, for example, >= denotes greater-than-or-equal-to, presumably because the symbol ≥ does not exist on your computer keyboard. Also, note that comparing two values for equality uses the == double-equals symbol. We already use the equals (=) symbol for assignment statements so == is used to compare two values as part of a Boolean expression.

|  |  |
| --- | --- |
| **Symbol** | **Description** |
| == | Equals |
| != | Not equals |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |

**Table 2.1:** Logical Operators

## Section 2.2: elif and Boolean Operators

Different states in the U.S. have different rules for when one is allowed to obtain a license to drive an automobile. One common practice is to allow individuals who are 16 or older to get a license. Suppose we wished to write a Python program that tells people if they are eligible to obtain a driver’s license in their state. We might prompt users for their respective ages, and then we could respond appropriately. Let’s use the following code as our starting point.

age = int(input("What is your age? "))

if age >= 16:

print("You are old enough to take your license exam.")

else:

print("Perhaps you should ride a bike for now.")

The situation in some states is more involved than this. In fact, let us suppose that if users are 14 or older, they can obtain a learner’s permit where they are able to drive if accompanied by an adult. If they are 16 *and* have had a driver’s education class, then they can take the license exam. Got it? Let’s ignore driver’s education for right now to simplify things, and let’s try to program it using what we know so far.

age = int(input("What is your age? "))

if age >= 14:

print("You may obtain a learner’s permit.")

else:

if age >= 16:

print("You are old enough to take your license exam.")

else:

print("Perhaps you should ride a bike for now.")

That’s a lot of indentation! Fortunately, the indentation makes the logic of our code a bit easier to follow, which is a nice feature of Python code. When we reach the first **else**, our code beneath the **else** starts at a new indentation level.

Try running this code using a variety of inputs. Do you notice a problem?

It appears that no matter what age we enter, we can never get it to tell us we’re old enough to take the license exam. Our Boolean conditions are not in the correct order. Even if we type that we are 17 years old, for example, the first condition is **True**, so it prints “You may obtain a learner’s permit.” Let’s fix the order of our code to place the most restrictive Boolean expression first.

age = int(input("What is your age? "))

if age >= 16:

print("You are old enough to take your license exam. ")

else:

if age >= 14:

print("You may obtain a learner’s permit.")

else:

print("Perhaps you should ride a bike for now.")

This is much better.

You might imagine that if you have a lot of different conditions you wish to handle in your program, your code many eventually look like a very long series of stair steps. This would make our code a bit difficult to read and maintain, so Python provides us with a statement for making a series of else-if conditions. This statement is called **elif**, for short. We can re-write the code above as follows.

age = int(input("What is your age? "))

if age >= 16:

print("You are old enough to take your license exam. ")

elif age >= 14:

print("You may obtain a learner’s permit.")

else:

print("Perhaps you should ride a bike for now.")

print("Done.")

Note that we’ve also added a statement that prints “Done” for illustrative purposes.

Having **elif**’s work the same way as having an **else** block followed by an **if** block. If the first condition is **True**, we print that we are old enough to take the exam. Then, none of the other conditions are checked. Instead, Python drops down to the “Done” print statement. If the first condition is **False**, then, and only then, is the next condition checked. If that condition is **True**, we print that they can obtain a learner’s permit and we drop down to the “Done” print statement. If that condition was **False**, our code descends to the **else** block.

As always, **else** blocks are always optional, as are **elif** blocks. You can have any number of **elif** blocks, but **if**/**elif**/**else** blocks must always come in that order: **if** first, then any **elif**’s, and finally an **else**.

Now let’s consider driver’s education. Individuals must be 16 years old *and* have had a driver’s education class to take the license exam. So, we’ll need to ask them if they’ve taken the exam by using another **input** statement.

age = int(input("What is your age? "))

**drivers\_ed = input("Have you passed driver’s education? (y/n) ")**

if age >= 16:

print("You are old enough to take your license exam. ")

elif age >= 14:

print("You may obtain a learner’s permit.")

else:

print("Perhaps you should ride a bike for now.")

print("Done.")

Notice how we are prompting users to enter either “y” or “n” for their answer to the driver’s education question. Also notice that we do not need to cast the value returned from the **input** function to another type like we did in the previous **input** statement (the one that asks for the age). The variable drivers\_ed can remain a string, because “y” and “n” are string values.

Now, let’s adjust our logic. We’ll only show the part of the code that deals with taking the license exam for now. This code:

if age >= 16:

print("You are old enough to take your license exam. ")

Can become this code:

if age >= 16:

if drivers\_ed == "y":

print("You are old enough to take your license exam. ")

This code says that in order to print “You are old enough to take your license exam” first age >= 16 must be **True**, and then drivers\_ed == "y" must be **True**, too.

There is another way to do this. We can use something called a **Boolean operator**. Instead of this:

if age >= 16:

if drivers\_ed == "y":

print("You are old enough to take your license exam. ")

We can say this:

if age >= 16 **and** drivers\_ed == "y":

print("You are old enough to take your license exam. ")

When we join two Boolean expressions with **and**, the resulting expression is **True** only if both of expressions are each **True**. If either age >= 16 is **False** or drivers\_ed == "y" is **False**, then the whole Boolean expression age >= 16 and drivers\_ed == "y" is **False.**

There are three Boolean operators in Python, which are given in Table 2.2.

|  |  |
| --- | --- |
| **Symbol** | **Description** |
| and | Suppose a and b are Boolean expressions.    If a == True and b == True, then a and b == True.  If a == True and b == False, then a and b == False.  If a == False and b == True, then a and b == False.  If a == False and b == False, then a and b == False.  In other words, the expression a and b is only True if both a and b are True. |
| or | Suppose a and b are Boolean expressions.    If a == True and b == True, then a and b == True.  If a == True and b == False, then a and b == True.  If a == False and b == True, then a and b == True.  If a == False and b == False, then a and b == False.  In other words, the expression a or b is True if whenever either a or b is True, or both of them are True. |
| not | Suppose a is a Boolean expression.  If a == True, then not a is False.  If a == False, then not a is True. |

**Table 2.2:** Boolean Operators

Let us look at another example where these Boolean operators become useful. Suppose we want to write a program that tells people if they have an increased risk for heart disease. There are a number of known risk factors, but for now we’ll only consider three. Without Boolean operators, we might write the program as follows.

bp = input("Do you have high blood pressure (y/n)? ")

smoke = input("Do you smoke (y/n)? ")

hist = input("Do you have a family history of heart disease ↵

(y/n)? ")

if bp == "y":

print("You have an increased risk of heart disease.")

elif smoke == "y":

print("You have an increased risk of heart disease.")

elif hist == "y":

print("You have an increased risk of heart disease.")

else:

print("You do not have an increased risk for heart disease.")

Note that each of the statements that follow the if/elif conditions all print the same message. Instead, we can use the Boolean operator or to simplify the code.

bp = input("Do you have high blood pressure (y/n)? ")

smoke = input("Do you smoke (y/n)? ")

hist = input("Do you have a family history of heart disease ↵

(y/n)? ")

if bp == "y" **or** smoke == "y" **or** hist == "y":

print("You have an increased risk of heart disease.")

else:

print("You do not have an increased risk for heart disease.")

Lastly, we can use the **not** operator to improve the readability of code. Recall how, in our driver’s license example, we asked if the user had taken a driver’s education course. With this scenario in mind, consider the following.

age = int(input("What is your age? "))

drivers\_ed = input("Have you taken driver’s ed (y/n)? ") == "y"

if 14 <= age and age <= 16 and **not** drivers\_ed:

print("You should consider taking a driver’s ed course.")

This example includes a few interesting code constructions. The second line is different from what you’ve seen before. In the code found earlier in this section, we let drivers\_ed be a string variable whose value should be "y" or "n". In this code, drivers\_ed is a Boolean variable. Why?

Notice that the **input** part of the statement is followed by a logical operator, the double-equals (==). Thus, we are comparing what the user types to the string "y". If the user typed "y", then the RHS of the assignment statement is True, so drivers\_ed gets the value True. If the user types anything else, drivers\_ed becomes False.

Next, we have code that checks to see if the user age’s is still typically in the range that students take a driver’s education class. If so, and they’ve not had driver’s education, they are advised to do so via a **print** statement. The only way the **print** statement will happen is if the age is between 14 and 16, and they’ve not had driver’s education.

Keep in mind that the programming practice in line 2 might not be a good idea. Your author has only written code this way to show you that it *can* be done. The latter part of the expression (== “y”) might be hard to notice, and because it is hard to notice, it may make the code more difficult to read and therefore maintain. In the future, we may want to check what the user typed in to make sure it’s what we expect (i.e., a “y” or a “n”). Checking inputs is something we’ll discuss in Chapter 3.

## Section 2.3: Nuances of Boolean Operators

Student will sometimes write code like this.

age = int(input("What is your age? "))

if age == 14 or 15:

print("You can get a learner’s permit.")

Run this code and type 18. Why does it print "You can get a learner’s permit”?

Remember than anything on the LHS and RHS of a Boolean operator must be itself a Boolean expression. In other words, the LHS and RHS must both be True or False. In the above example, there is an expression on either side of the double-equals. One is

age == 14

and the other is

15

The expression age == 14 is False. The expression 15 is True. **Anything non-zero in Python is treated as True.** Therefore, the Boolean expression is transformed as follows.

age == 14 or 15

False or 15

False or True

True

Instead of

age == 14 or 15

a programmer should write

age == 14 or age == 15

Bottom line: be very careful when using and/or.

## Chapter 2 Exercises

1. Suppose x is an integer variable. What is the difference between the expression x = 2 and the expression x == 2?
2. Given the following code:

x = 3

y = 6

What is the type and value of each of the following expressions?

x != 0

y >= 3

x == y

not (x == 0)

x > 1 and y < 5

2 < x and x < 5

7 < y or y < 10

1. Identify the error in the following code.

num = float(input("Enter a real number: "))

if num < 0.0:

print("%.2 is negative." % num)

else:

print("The number is 0.0.")

elif num > 0.0:

print("%.2 is positive." % num)

1. Identify the error in the following code.

num = float(input("Enter a real number: "))

if num = 0.0:

print("The number is 0.0.")

elif num < 0.0:

print("%.2 is negative." % num)

else:

print("%.2 is positive." % num)

1. What is the output of the following code?

major = "Music"

if major == "Computer Science" or "Math":

print("Calculus is required for your major.")

elif major == "Music":

print("Music Theory is required for your major.")

1. Consider the following code.

if a > 2:

if b < 3:

print("Statement 1")

else:

print("Statement 2")

else:

if b > 3:

print("Statement 3")

else:

print("Statement 4")

What is the output of the program if a and b were given the following values prior to the start of the code?

a = 3

b = 4

a = -2

b = 2

1. What is the output of the following code?

r = 10

if 5 < r and r < 15:

print("1")

if 5 < r or r < 15:

print("2")

r = 100

if 5 < r and r < 15:

print("3")

if 5 < r or r < 15:

print("4")

# Chapter 3: Loops and String Manipulation

## Section 3.1: while loops

Sometimes we’ll want to repeat an action in code. One example might be that we repeatedly ask the user for input until we run out of that input. Suppose we write a program that calculates the amount of sales tax on different items at a store. We might do the following.

tax = 0.07 # that is, 7 percent

print("Enter the amount of each item.")

print("Enter 'quit' to end the program.")

amount = input("Amount of item? ")

while amount != "quit":

amount = float(amount)

total = (amount \* tax) + amount

print("The total for that item will be %.2f." % amount)

amount = input("Amount of item? ")

print("Thank you for using this program.")

We’ve seen many of these ideas before. We are getting input from the user. If the input is not the word “quit,” we cast the amount to a float and compute the total price of the item with tax included.

What is special about the above code is the keyword **while**. **while** works similar to **if** in many ways. When we use **if**, we give a condition immediately after the word **if**, and if the condition is **True**, Python executes the code that is indented beneath it. **while** works the same way, however, once we reach the end of the indented block, the code “loops” back up to the condition again. If the condition is again **True**, we reenter the indented block of code. Thus, we can make code repeat itself as long as a condition is met. A programming language “construct” that allows code to be repeated is known as a **loop**.

We call amount != "quit" in this example the **loop condition**. We call the indented block of statement that follows the loop condition the **loop body**. We say that the statements in the indented block are *inside* the loop body. The final **print** statement in the code is not indented and is therefore *outside* the loop body.

Try out the code above if you haven’t already. Does it behave the way you expect?

Now, notice that we ask for input twice in this program: once just before we try to enter the loop, and once at the end of the indented block inside the loop body. Why? The value returned by the initial **input** statement gets us into the loop body, though it doesn’t have to. It is possible that the user types “quit” right away, in which case the program bypasses the loop and prints our “thank you” message. At the end of the loop body, we need to ask the user for the next input value, so that we have a new amount. Otherwise, the program would just continue to calculate the same total value over and over again and the program would never have the chance to end.

To see this for yourself, delete the last line of the loop body, i.e., the **input** statement that is the last line of the indented block of statements. Instead of deleting it, you could just comment it out by putting a # symbol at the start of that line. Run your code and type an initial amount, say, 4.00. Your code will endlessly compute sales tax with no end in sight! Press the “Control” and “C” keys simultaneously to abort your program (we will write this Ctrl+C in the future because that’s how programmers write such things).

With this in mind, we will often see loops written using the following pattern.

x = some initial value

while condition involving x:

code to be repeated

x = a new value of x

We call each new time through the body of the loop an **iteration**. We may refer to the first iteration, second iteration, next iteration, final iteration, etc., when discussing the behavior of a loop.

To get a sense of the mechanics of how **while** loops work, beginning programmers often consider “toy” examples that showcase a number of “gotchas” when it comes to programming loops. Consider the following.

x = 1

while x <= 5:

print(x)

x = x + 1

print("Done.")

Run it. This prints the following.

1

2

3

4

5

Done.

We don’t always need to have a non-indented statement following a loop. We are only doing this in the examples so it is clear in the output that we left the loop body.

In each iteration, we print the value of x and then increase x by 1. Adding 1 to a value is such a common thing in programming that it has a special name: **increment**. That is, we are **incrementing** x by adding 1 to it. When x is 5, we print the 5 and then add 1 more to make x become 6. When we evaluate the expression x <= 5, we get 6 <= 5, which is False, so our code falls down to the non-indented section of code.

It is a good thing to observe that, in this example, the last value we print is 5, but the value of x when we exit the loop is 6.

Let’s look at another “toy” example. We will change our first example, which was the following.

x = 1

while x <= 5:

print(x)

x = x + 1

print("Done.")

Instead, we will switch the order of the loop body statements, like this.

x = 1

while x <= 5:

x = x + 1

print(x)

print("Done.")

Run it. Our output is different from the first example.

2

3

4

5

6

Done.

Can you explain why this is?

In this new example, we are incrementing x *before* we print it. Thus, the first value we print is not 1 but 2. The order of statements matters in programming, and it definitely matters in loops.

Consider another example.

x = 1

while x <= 5:

print(x)

print("Done.")

Here, we have removed the line that increments x. You should be able to guess what will happen when we run this program (hint: be ready to press Ctrl+C).

Since we have removed the line that increments x, x never changes, and so the program just prints 1 over and over again. Loops that are never-ending are called **infinite loops**.

Let’s try another example.

x = 0

while x <= 4:

print(x)

x = x + 1

print("Done.")

All we’ve done is changed the initial value of x and the value of x that causes us to not reenter the loop. The output is:

0

1

2

3

4

Done.

Let’s change the code again.

x = 0

while x < 4:

print(x)

x = x + 1

print("Done.")

Now instead of <=, we have <. This makes the code end the loop one value of x earlier. The output is:

0

1

2

3

4

Done.

Now, let’s change the condition so that it uses > instead of <. The code now reads:

x = 0

while x > 4:

print(x)

x = x + 1

print("Done.")

What is the output? Why?

The output will be, as you may or may not have expected, is:

Done.

The variable x is set to zero initially. In the next line, the expression x > 4 becomes 0 > 4, which is False. Thus, the code bypasses the loop body entirely. There is no “rule” that says we have to do the loop body at least once. Computers are stupid. They only do what you tell them to. In this case, because the condition is False initially, we never do the loop body and go straight to the non-indented line of code past the loop body.

All of our “toy” examples thus far have involved incrementing x, that is, adding 1 to x. However, we can change our variable however we wish. We could do this.

x = 2

while x <= 8:

print(x)

x = x + 2

print("Who do we appreciate?")

In this code, we add 2 to x each time. The output of this code is:

2

4

6

8

Who do we appreciate?

We could also write code like this:

x = 5

while x > 0:

print(x)

x = x - 1

print("Blastoff!")

In this code, we subtract 1 from x each time. This is called **decrementing** x. The output of this code is:

5

4

3

2

1

Blastoff!

Toy examples are a great way to understand how loops work from a “mechanical” perspective, but let’s see how we can use them to solve problems. Recall our very first example in this section, the one dealing with reporting the price of an item after sales tax. Our next example is also related to money. Suppose we open a savings account, one that pays 5% interest per year into the account as long as we don’t withdraw any money that we deposit. Let’s say we start the account by depositing $10,000.00. How many years will it take to double our investment?

The question “How many years will it take to double our investment?” tells us a lot about the code we would need to write. We would need to keep track of our balance as it grows by 5% each year. We would need to keep track of the number of years that have elapsed so far in our code. We would also need to *check* to see if the balance has doubled. If, in describing what our program is supposed to do, we say we need to *check* something, we are often talking about a Boolean condition found in an **if** statement or a **while** loop.

Try to program this example on your own first. You can’t learn to be a good programmer if you don’t try things regularly by starting with a blank screen or a blank sheet of paper. If you get really stuck, then (and only then) glance at the answer below.

Okay, let’s look at a solution (Warning: Our initial solution will have flaws, as initial solutions often do).

balance = float(input("Enter starting balance: "))

years = 0

while balance < 2\*balance:

balance = balance \* 0.05 + balance

years = years + 1

print("The balance after %d years is $%.2f." % ↵

(years, balance))

print("It will take %d years to double your investment." % years)

This is a good first attempt. Here, we are letting the loop do the work of adding to the balance year after year, each time checking to see if the balance has doubled and if we can leave the loop. However, our logic is flawed a bit. Consider the loop condition balance < 2\*balance. A variable’s value will never be greater than twice itself (unless that value is negative, but let’s steer clear of *negative* values in bank accounts!).

The problem here is that we’re trying to use the variable balance for two very different purposes simultaneously. On one hand, we are using balance to keep track of the current balance as it changes year after year. On the other hand, we’re pretending that it still holds the *starting* balance from our initial investment. We should really keep the starting balance in a separate variable.

With this in mind, we can modify the previous code as follows.

**startbalance** = float(input("Enter starting balance: "))

**balance = startbalance**

years = 0

while balance < 2\***startbalance**:

balance = balance \* 0.05 + balance

years = years + 1

print("The balance after %d years is $%.2f." % ↵

(years, balance))

print("It will take %d years to double your investment." % years)

## Section 3.2: manipulating and printing strings

Given our newly found exposure to loops, now is a good time to revisit strings. It turns out, there is a fair amount of nifty stuff we can do with strings using loops. Suppose we create a new string variable named s in the following way.

s = "abc def"

Note that the string contains a space between the substrings "abc" and "def". We can inspect individual characters by using square brackets and character’s position in the string. Consider the following example.

print(s[0])

print(s[1])

print(s[2])

print(s[3])

print(s[4])

print(s[5])

print(s[6])

print(s[7])

If we run this code, we see the following output.

a

b

c

d

e

f

Traceback (most recent call last):

File "/Users/shep/cs1/code/ch3strings.py", line 9, in <module>

print(s[7])

IndexError: string index out of range

The syntax for retrieving a single character from a string s is s[index] where index is an integer representing the position of the character we wish to retrieve. The first character (in this case "a") is at position 0 rather than position 1. That may take a little getting used to.

Let’s re-list the code again, this time with comments to explain each line.

s = "abc def"

print(s[0]) # print "a"

print(s[1]) # print "b"

print(s[2]) # print "c"

print(s[3]) # print " " (which ends up as just a blank line)

print(s[4]) # print "d"

print(s[5]) # print "e"

print(s[6]) # print "f"

print(s[7]) # Kablooey! There is no character at index 7.

Between the square brackets, you can place one index or you can give a range of indices, like this.

s = "abc def"

print(s[0:2])

print(s[0:3])

print(s[3:6])

print(s[3:])

print(s[:4])

This code yields the following output.

ab

abc

de

def

abc

We can observe that the first print does not print characters at indices 0 through 2. Rather, it prints characters 0 and 1 but not 2. We might conclude from this that when we have two indices in square brackets following a string, the first number is the first position *inclusive* and the second number is the last position that will be selected up to, but not including, or *exclusive*. Note that the indices are separated by a colon (:). In this situation, we will call the colon the **slicing operator**.

Lines 4 and 5 show us that the first and second indices of the slicing operator are optional. If one is omitted, Python assumes that the we mean “go to the ends of the string.” When we use square brackets to select one or more characters from a string, we say that we are obtaining a **substring** of the original string.

What appears to be indentation in lines 3 and 4 of the output is actually the single space found at index 3 in the string s. It is difficult to tell if there is a space at the end of line 5. One way to tell is to append an additional character to the output to see the space. For example:

print(s[:4] + "$")

The output is:

abc $

Can you see the space in the output above?

There are quite a few things we can do with strings. Another interesting feature of strings is that they have their own set of special functions. Consider the following example.

t = "I like cheese."

shout = t.upper()

print(shout)

Most of the functions we have seen thus far (like print, input, etc.) do not have a period/dot in front of them. Calling a function which operates specifically on a value or a variable uses **dot-notation**,like t.upper().

On the Internet, typing something in all capital letters is a convention that indicates shouting. In effect, the upper() function can be used on a string value (which is often contained in a variable) to return an all-uppercase version of that string. The output of this code is:

I LIKE CHEESE.

It is *very* important to note that upper() does not change the contents of the string variable t. If I do this:

t = "I like cheese."

shout = t.upper()

print(shout)

print(t)

I get this as output:

I LIKE CHEESE.

I like cheese.

Functions like upper() are not restricted to being called on variables. They may be called on any string value, like this:

shout = "I like cheese.".upper()

print(shout)

This works because Python transforms the first statement as shown below.

"I like cheese.".upper()

"I LIKE CHEESE."

A common mistake is to omit the parentheses at the end of upper(). Remember that upper is just the name of the function; the parentheses are what calls the function. If we remove the parentheses, we get output that looks strange.

t = "I like cheese."

print(t.upper)

This produces something like:

<built-in method upper of str object at 0x1019807e8>

This is Python’s way of saying “yes, upper is the name of a function. If you want to actually use that function, put parentheses at the end.” Let’s add the parentheses to the end of the function call.

t = "I like cheese."

print(t.upper())

And now, it works.

Observe that when we “shout” by using upper(), the punctuation displayed is a period. Shouldn’t we change that to an exclamation point (!)? Consider the following code.

t = "I like cheese."

shout = t.upper().replace(".", "!")

print(shout)

The function replace is another string function. It takes two arguments. The first is a pattern to find in the string, and the second is a string we use to insert instead.

Consider how Python executes that second line.

shout = t.upper().replace(".", "!")

shout = "I like cheese.".upper().replace(".", "!")

shout = "I LIKE CHEESE.".replace(".", "!")

shout = "I LIKE CHEESE!"

This example works because t is a string, so calling t.upper() returns another string. Because t.upper() is a string, we can call replace on it to return yet another string. Calling a series of functions in one line is called **function chaining**.

Note that the original string value of t still hasn’t changed. If we wanted to actually change t, we would need to use an *assignment statement*, like this:

t = t.upper().replace(".", "!")

Now, t is changed to the newly returned string value.

We can even chain calls to replace, like this:

t = t.upper().replace("LIKE","LOVE").replace(".", "!")

As you might guess, and in the interest of completeness, in addition to upper there is a lower function for returning all lowercase versions of strings.

Now, let us explore the relationship between strings and loops. Consider the following code.

s = "abc def"

index = 0

while index < 7:

print(s[index])

index = index + 1

This produces the output:

a

b

c

d

e

f

Why?

The variable index starts at 0. Each time in the loop body, we print the value of the expression s[index], which in the first loop iteration is s[0] or "a". The code adds 1 to index , so now index is 1 (it was 0 previously). The code loops back around to the beginning of the loop. We print the value of s[index] again, which this time is s[1] or "b". We continue in this fashion until index is 7, at which point we exit the loop.

This is yet another “toy” example intended to get us to see the relationship between loops and strings, which is a powerful relationship we will exploit very shortly.

Let’s make one change to this example. We will change only the value of the string variable s.

s = "protagonist"

index = 0

while index < 7:

print(s[index])

index = index + 1

The output is now:

p

r

o

t

a

g

o

Oh no! Where is the rest of the string? Why did it stop after 7 characters?

Unfortunately, we have **hard-coded** the loop condition as index < 7. From a logic standpoint, the 7 is supposed to refer to the length of the string, which is also 1 more than the last index in the string. Recall from the previous example that the last index of s was 6 and the number of characters (the *length* of the string) was 7. In this example, the length of the string is 11, not 7.

Any time we hard-code a value, like 7, it makes our code brittle and less resistant to change. Let us replace this line:

while index < 7:

with this line:

while index < len(s):

The len function returns the number of characters in the string.

Alternatively, we could have done this:

s = "protagonist"

index = 0

lastindex = len(s) - 1

while index <= lastindex:

print(s[index])

index = index + 1

It is very important to see the relationship between the last index and the length of the string. Because indices start at 0, not 1, the last index is always the length minus 1.

Let’s see how well this is all sinking in. Can you write code to print the characters of a string, one on each line, but in reverse order this time? In other words, if my string is "abc", can you write code that prints:

c

b

a

Give it a try.

Let’s check your work. With some experimentation, you might have been able to get close to a solution that looks like this. (If not, keep trying and practicing.)

s = "abc"

index = len(s) - 1

while index >= 0:

print(s[index])

index = index - 1

Note that we are setting the initial index variable to the last index of string (again, length minus 1), and in our loop body we are subtracting 1 rather than adding 1.

In Python, as it is in other programming languages, there is often more than one way to write a program. Another approach to printing a string in reverse could take advantage of the fact that you can put *negative* indices in square brackets to obtain characters in a string starting from the end rather than from the beginning. For example, suppose we have the following code.

foo = "abc"

print(foo[-1])

print(foo[-2])

print(foo[-3])

This prints:

c

b

a

With this in mind, we could write new code that produces the same output as our earlier code, but in a different way.

foo = "abc"

index = -1

while index >= -len(foo):

print(foo[index])

index = index - 1

Note the loop condition. If len(foo) in this case is 3, then -len(foo) is -3.

## Section 3.3: for loops

Loops of the following form are very common. Suppose n is an integer variable defined earlier in a program.

k = 0

while k < n:

print(k)

k = k + 1

Loops of this form are some common, in fact, that there is a shorter version of the loop known as a **for** loop. The above loop could be re-written as a for loop as follows.

for k in range(0, n):

print(k)

Gone is the explicit declaration of the variable k. Gone also is the increment statement k = k + 1. The range expression gives k an initial value of 0 in this example, and then it allows the loop to continue until k == n.

Given our previous example of printing the characters of a string:

s = "protagonist"

index = 0

while index < len(s):

print(s[index])

index = index + 1

We can now re-write this code using the shorter for loop.

s = "protagonist"

for index in range(0, len(s)):

print(s[index])

For loops can also countdown or count in multiples just as easy as while loops can. Suppose we want to print a string in reverse order like we did earlier. We had the following.

s = "cba"

index = len(s) - 1

while index >= 0:

print(s[index])

index = index - 1

Instead, we can add a third argument to range which specifies what we add to the loop variable to change it prior to each iteration of the loop. Thus, we can do this.

s = "cba"

for index in range(len(s)-1, -1, -1):

print(s[index])

That’s a lot to mentally unpack. Let’s take it a piece at a time. Let’s take the first bolded expression in range below.

for index in range(**len(s)-1**, -1, -1):

This sets the last index in the string (len(s)-1) as the initial value of the variable index. The next expression in range is:

for index in range(len(s)-1, **-1**, -1):

This says that when index goes past 0 and reaches -1, we should exit the loop. Finally, we have:

for index in range(len(s)-1, -1, **-1**):

This -1 specifies how index should be changed prior to each iteration. This means that index will have 1 subtracted from its value.

Let’s see how we are doing here. Suppose I have the following code.

for i in range(1, 9, 2):

print(i)

What is the output of this code?

We would expect to see the following on-screen.

1

3

5

7

Can you explain why?

The variable i gets the value 1 to start the loop. In each iteration, the value of i is printed. Just before each iteration, we add 2 to i. Therefore, i will have the values 1, 3, 5, and 7 within the loop body. When i reaches 9, the for loop will end.

Let us try to use our knowledge of for loops to write a program that reverses a string. In other words, given user input, reverse the input and print it to the user. If the user enters “abc def”, then the program should output “fed cba”.

Here is our attempt.

s = input("Enter a string: ")

rev = ""

for i in range(len(s)-1, -1, -1):

rev = rev + s[i]

print("The reverse of the string is: %s." % rev)

s is the input string variable and rev will be the string variable that holds s’s reverse. We take each character from s using a for loop and we concatenate it onto the end of rev.

Consider how rev changes with each iteration of the loop, as shown in the following table.

|  |  |  |
| --- | --- | --- |
| **rev** | **i** | **s[i]** |
| "f" | 6 | "f" |
| "fe" | 5 | "e" |
| "fed" | 4 | "d" |
| "fed " | 3 | " " |
| "fed c" | 2 | "c" |
| "fed cb" | 1 | "b" |
| "fed cba" | 0 | "a" |

This type of table is called a **trace table**. It serves as a tool for programmers to make sense of how values change from iteration to iteration of a loop.

Let us demonstrate another way to solve this problem that uses an alternate form of the for loop. Suppose we wish to iterate through the characters of a string s with the intent of reversing it. We might write the code as:

s = input("Enter a string: ")

rev = ""

for index in range(0, len(s)):

rev = s[index] + rev

print("The reverse of the string is: %s." % rev)

Notice how we’ve changed the logic from our previous attempt. In the previous code, we traversed the string from the back to the front, and for each character, we appended it to the end of the reversed string rev. In this code, we are doing the opposite. We are traversing the string from front to back and prepending the character to the front of the reversed string rev. Both methods solve the same problem and produce the same output.

As it turns out, we don’t even need to concern ourselves with having an index variable. Python has a version of the for loop syntax that operates exclusively on strings. Consider the following adjustments to our prior code.

s = input("Enter a string: ")

rev = ""

for ch in s:

rev = ch + rev

print("The reverse of the string is: %s." % rev)

In this form of the Python for loop, the loop variable ch gets a different character from the string s each time the loop condition is checked. Notice that there is no more index variable or range function.

## Section 3.4: more string manipulation with loops

We have been asking users for input in many of our previous examples. We have always assumed users will type the correct information, but what happens when they do not. Suppose we ask users to enter their age. We have been doing this as follows.

age = int(input("Enter your age: “))

If the users’ fingers slip and they enter “1r” instead of “14”. The int() function will crash because “1r” cannot be cast to an integer. This happens because we have prematurely tried to cast the string to an integer without checking the input first. We could do this instead.

age\_str = input("Enter your age: ")

while not age\_str.isdigit():

print("That is not a valid age.")

age\_str = input("Enter your age: ")

age = int(age\_str)

isdigit() is always called *on* a string value and it refers specifically to that value. The “dot” (.) separates the word isdigit from the string it refers to, in this case the variable age\_str. age\_str.isdigit() returns True if all of the characters in age\_str are numbers, otherwise it returns False.

The loop in this case is used to perform **input validation**. That is, the program will not leave the loop until the user types a response in the correct format.

Let us examine another example that uses loops to operate on strings. Suppose we ask users for their full names using an input statement.

fullname = input("Enter your full name: ")

The variable fullname could contain "Javier Sanchez", "Mary Johnson", or just about anything, though in most cases, there will be a first name and a last name separated by a space. Let us try to extract the first name from fullname. To do this, we’ll need to use the slicing operator. Depending on the length of the first name, the range of the string slice will be a little different.

# Suppose fullname == "Javier Sanchez"

# 01234567...

firstname = fullname[0:6]

# Or, suppose fullname == "Mary Johnson"

# 01234567...

firstname = fullname[0:4]

Since we can’t possibly know the length of the first name ahead of time, we will need to find the index of the space in order to slice the appropriate substring. If we are able to put the index of the space into a variable named space, then we can extract the first name as follows.

firstname = fullname[0:space]

If fullname is "Javier Sanchez" then space would be 6, if fullname is "Mary Johnson" then space would be 4, and so on. Finding the index of the space requires us to search the string character by character, which is something we have already done, both with while loops and with for loops.

Here is our strategy. We will use a loop to examine each character in fullname, one at a time. If we encounter the space, we will set the value of space to the current index of the loop, and then we will break out of the loop. After all, once we’ve found the space, our work is done; there is no need to keep looping.

Thus, our code now looks like this.

fullname = input("Enter your full name: ")

for index in range(0, len(fullname)):

if fullname[index] == ' ':

space = index

break

firstname = fullname[0:space]

print("Your first name is %s." % firstname)

Try out this code on several full names that you can think of. Try to find **edge cases** that may not work. Edge cases are exceptional cases of program input that could cause your code to fail. One such edge case in this problem is full names that do not have spaces, i.e., the names of individuals who only have one name. Consider celebrities’ names such as Prince, Cher, Sting, Madonna, etc. Those names are their full names and first names simultaneously. Try typing "Prince" in as the input to your program.

Enter your full name: **Prince**

Traceback (most recent call last):

File "/Users/shep/cs1/code/ch3fullname.py", line 6, in <module>

firstname = fullname[0:space]

NameError: name 'space' is not defined

Can you explain this error to yourself? Why is space not defined in line 6? Where in the code do we create the space variable normally?

In this case, space is first created inside the loop. This is generally bad programming practice. If the variable is used by code outside of a loop, the variable should also be defined outside the loop. So, let’s define space before we reach the loop, but what should be its initial value? Let’s choose a value that doesn’t look like a normal index. Normal indices would be 0, 1, 2, 3, etc., so let’s choose -1 as the initial value of space. Our code can now be written as follows.

fullname = input("Enter your full name: ")

space = -1

for index in range(0, len(fullname)):

if fullname[index] == ' ':

space = index

break

if space == -1:

firstname = fullname[0:space]

else:

firstname = fullname

print("Your first name is %s." % firstname)

Finding a character or a pattern in a string is such a common operation, Python gives us a way to do it without writing a loop every time. We can simplify the code above using the find() function as follows.

fullname = input("Enter your full name: ")

space = fullname.find(" ")

if space == -1:

firstname = fullname[0:space]

else:

firstname = fullname

print("Your first name is %s." % firstname)

find() is always called *on* a string value and it refers specifically to that value. The “dot” (.) separates the word find from the string it is searching, in this case the variable fullname. The argument is the single character or multiple character pattern to search for. find() returns the index where that pattern starts in the string.

Here are some examples of using find().

print("abcde".find("b")) # prints 1

print("abcde".find("cd")) # prints 2

print("abcde".find("cash")) # prints -1, which indicates the

# pattern was not found

print("abcde".find("abcde")) # prints 0

Now that we know how to find a pattern (and specifically, a space) in a string, consider what that allows us to do. Suppose we have a sentence that we wish to break into individual words. How are words separated in the English language? If you said “using spaces” then you are correct (for the most part).

Let’s try to do something like this through an example. If you spend time on the Web, you might have learned that Internet citizens have tabbed September 19th as “International Talk Like a Pirate Day.” Imagined pirate-speak draws from many sources in popular culture. Fictional pirates might say “Ahoy!” instead of “Hello!” or they might use the word “matey” to describe a friend. If we wanted to translate a single English word into its Pirate equivalent, we might do something like this.

english\_word = input("Enter a word: ")

if english\_word == "hello":

pirate\_word = "ahoy"

elif english\_word == "friend":

pirate\_word = "matey"

else:

# The word has no known pirate translation.

# Leave it in English.

pirate\_word = english\_word

print("%s in pirate is %s." % (english\_word, pirate\_word))

What if we wanted to change an entire English sentence into Pirate instead of just translating a single word? How would we accomplish that?

Let’s keep this as simple as possible, so for now let’s ignore “edge cases” like uppercase letters and punctuation. Let’s only worry about translating all-lowercase words separated by spaces.

You might not have any idea how to accomplish this in code, but there are a number of ways to get started. One way is to realize that every computer program tends to take input and transform it into output. If we envision a program as taking variables and changing their values in order to reach a goal, we can write down how we want those variables to change, and then write down the code to perform the changes.

Let’s think about what variables we would want to have to make an English-to-Pirate translator program work. Suppose we have two variables, one named sentence and the other named word. sentence contains the English sentence the user types, and word will store each word as we extract it from sentence. Each time we extract an English word into word, the sentence should shrink.

Again, if we are thinking about a program as something that makes changes to input variables, we can envision a program looping to change sentence and word repeatedly. Here is an example.

|  |  |
| --- | --- |
| word: "" | sentence: "hi there my friend" |
| word: "hi" | sentence: "there my friend" |
| word: "there" | sentence: "my friend" |
| word: "my" | sentence: "friend" |
| word: "friend" | sentence: "" |

Think of this process in a loop. Each time through the loop, we find the index of the space in sentence and put that index into a variable named space. We use the slicing operator to get the first word in the sentence from 0 to space. Then, we use the slicing operator again to “shrink” sentence so that sentence only contains the remaining words.

This would allow us to translate each word, one at a time. Let’s write some code.

sentence = input("Enter an English sentence: ")

word = ""

print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

space = sentence.find(" ")

while space != -1:

word = sentence[0:space]

sentence = sentence[space+1:]

print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

space = sentence.find(" ")

Let’s try it out.

Enter an English sentence: **hello there my friend**

word: "" sentence: "hello there my friend"

word: "hello" sentence: "there my friend"

word: "there" sentence: "my friend"

word: "my" sentence: "friend"

This is pretty good, but what about the final word in this example: “friend”. We will need to add some code after the loop to take care of the last word, like this:

sentence = input("Enter an English sentence: ")

word = ""

print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

space = sentence.find(" ")

while space != -1:

word = sentence[0:space]

sentence = sentence[space+1:]

print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

space = sentence.find(" ")

if sentence != "":

word = sentence

sentence = ""

print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

The last block of code checks to see if there’s still text residing in sentence.

Now all that’s left is to do the translating. We will take each word and append its translation to a new string which we’ll call pirate. We can get rid of our print statements, too, since they were only there to aid our understanding of what the code does.

sentence = input("Enter an English sentence: ")

word = ""

pirate = ""

space = sentence.find(" ")

while space != -1:

word = sentence[0:space]

sentence = sentence[space+1:]

if word == "hello":

pirate = pirate + "ahoy" + " "

elif word == "friend":

pirate = pirate + "matey" + " "

else:

pirate = pirate + word + " "

space = sentence.find(" ")

if sentence != "":

word = sentence

sentence = ""

if word == "hello":

pirate = pirate + "ahoy" + " "

elif word == "friend":

pirate = pirate + "matey" + " "

else:

pirate = pirate + word + " "

print("Your sentence translated to pirate is:")

print(pirate)

An example of a running program is:

Enter an English sentence: **hello there my friend**

Your sentence translated to pirate is:

ahoy there my matey

We can add more translations by adding more elif conditions to the if-blocks of statements, though we must add them both to the while loop code and the if-block that follows the while loop. Having to add this logic in two different places makes the code more difficult to maintain, since we might add an elif to the earlier part of the code but then forget to add it to the later part of the code. In Chapter 4, we will examine **functions**, which will improve the maintainability of this code dramatically.

## Section 3.5: nested loops

Some interesting and practical things happen when we use a loop as the body of another loop. Consider the following code, and try to guess as to its output.

for outer in range(0, 3):

for inner in range(0, 3):

print("outer = %d, inner = %d" % (outer, inner))

Let’s follow the code step-by-step as we would with any code. The first statement we encounter is for outer in range(0, 3). Since this is first time we’ve encountered this statement, this code sets outer to 0 and we enter its loop body. The first statement of the loop body is, of course, another loop. This loop statement is for inner in range(0, 3). So, this sets inner to 0 and we enter its loop body. This means that at this point in the code, we are actually inside two loops: an inner one, which is inside an outer one. The first (and only) statement in the inner loop prints the values of outer and inner so that we can see how the values change throughout the program.

We call a loop-inside-a-loop construct a set of **nested loops**. That is, the inner loop is nested inside the outer loop.

Before we continue discussing how these loops work, run the code to see the output.

outer = 0, inner = 0

outer = 0, inner = 1

outer = 0, inner = 2

outer = 1, inner = 0

outer = 1, inner = 1

outer = 1, inner = 2

outer = 2, inner = 0

outer = 2, inner = 1

outer = 2, inner = 2

We can get a pretty good idea of how this works from looking at the output. Once we enter the outer loop, we encounter the inner loop right away. Python will stay in the inner loop until it is completed, and then it will exit to the outer loop. The outer loop will increment outer by one, and since outer is still within its range, it will enter the outer loop body, which, once again, will reach the inner loop. Since we left the inner loop once already, reaching the inner loop this time is like reaching it for the first time. The statement for inner in range(0, 3) sets inner to 0 once again, and the process repeats itself.

We often use nested loops to write programs in Python. For one example, suppose we wish to write a program that repeatedly asks for students’ names and exam scores. The output should be students’ names followed by their exam average. The following solution shows how to do this using nested loops.

student = input("Student name? ('quit' to end) ")

while student != "quit":

sum = 0.0

count = 1

score = input("Score %d? ('quit' to end) " % count)

while score != "quit":

score = float(score)

sum += score

count += 1

score = input("Score %d? ('quit' to end) " % count)

# count is one more than it should be since we

# increment count before we quit.

count -= 1

if count > 0:

average = sum / count

print("%s's average is %.2f" % (student, average))

else:

print("No grades entered for %s." % student)

student = input("Student name? ('quit' to end) ")

One thing to note about this code, besides the nested while loops, is the shorthand assignment statements. Instead of typing

count = count + 1

we can type

count += 1

This type of shorthand assignment construct works for most operators in Python.

Beyond that observation, we note that the outer while loop is responsible for getting each student’s name. The inner while loop retrieves each score and adds it to a sum. Once we exit the inner loop, we can calculate the average using the sum. We must be careful to make sure that there are any scores at all. If count were 0 and we divided by 0 without checking, our program would crash.

The output of this code might look something like the following depending on what the user types.

Student name? ('quit' to end) **David Chan**

Score 1? ('quit' to end) **95.5**

Score 2? ('quit' to end) **80.5**

Score 3? ('quit' to end) **87.0**

Score 4? ('quit' to end) **quit**

David Chan's average is 87.67

Student name? ('quit' to end) **Sarah McDowell**

Score 1? ('quit' to end) **quit**

No grades entered for Sarah McDowell.

Student name? ('quit' to end) **quit**

## Chapter 3 Exercises

1. Write a program that asks for a word, phrase, or sentence. The program should then print whether the input is a palindrome. (Do a Web search to see what a palindrome is.)
2. Write a program that repeatedly asks for a number until the user enters “quit.” The program should print the sum and average of all the numbers.
3. What is the output of the following program.

for i in range(2, 10, 2):

print(i)

1. What is the output of the following program.

for i in range(5, 0, -1):

print(i)

1. What is the output of the following program.

for i in range(1, 6, 2):

for j in range(1, 3):

print("%d %d" % (i, j))

1. Write an English to Pirate translator like the one found in Section 3.4, only have your program repeatedly ask for sentences until the user presses ENTER/RETURN without typing anything. You can still ignore capitalization and punctuation.
2. Do #3 again, only this time properly handle capitalization and punctuation.
3. Write a program that translates English sentences to Pig Latin. To form the Pig Latin equivalent of a word, remove the first consonant sound of the word and append it to the end, followed by “-ay”. Thus, “cat” becomes “at-cay” and “ship” becomes “ip-shay.” The latter example demonstrates that consonant sounds can be blended consonants. If an English word begins with a vowel, the word is simplify restated with “-way” appended to it. In other words, “apple” becomes “apple-way.” Your solution should handle capitalization and punctuation.

Example of a working program:

Enter English (press 'ENTER' only to quit): Hey there, Delilah!

The Pig Latin equivalent is: Ey-hay ere-thay, Elilah-day!

Enter English (press 'ENTER' only to quit): Welcome to the Apple Store.

The Pig Latin equivalent is: Elcome-way o-tay e-thay Apple-way Ore-stay.

Enter English (press 'ENTER' only to quit):

Thank you for using the Pig Latin translator!

1. Write a program that asks for a number and then prints that number of rows of 5 asterisks. For example, if the user enters 4, the program would print the following.

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

1. Write a program that asks for two numbers *x* and *y*. The program should print a rectangle of asterisks consisting of *x* rows and *y* columns. For example, if the user enters 3 and 8, respectively, the program would print the following.

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

1. Write a program that asks for a number and then prints a triangle of asterisks where the base has that number of asterisks. For example, if the user enters 4, the program would print the following.

\*

\*\*

\*\*\*

\*\*\*\*

1. Write a program that asks for a number and then prints an inverted triangle of asterisks where first row contains that number of asterisks. For example, if the user enters 4, the program would print the following.

\*\*\*\*

\*\*\*

\*\*

\*

1. Python programs are capable of generating random numbers (well, "pseudo"-random numbers – more about that in the next chapter). One example of how to do this is as follows.

import random

x = random.randint(1,10)

print("Here is a random integer between 1 and 10: %d" % x)

If you run this program over and over again you’ll see the number change. We can use random integers to *simulate* real-life occurrences, including games of chance and movement of animals.

Consider one program that demonstrates the use of random numbers. A drunkard stumbling around in a grid of streets picks one of four directions (north, south, east, or west) at each intersection, and then he moves to the next intersection.

Write a program that simulates the drunkard’s walk. Represent each intersection location as (x,y) integer pairs and have the drunkard start at (0,0). Have the drunkard walk 100 intersections and print the intersection location where he ends up. Run this program many times. Does the drunkard tend to stay close to (0,0) or does he end up moving far away?

# Chapter 4: Functions

## Section 4.1: Introduction to Functions

## Section 4.2: Parameters

## Section 4.3: Return Values