ch 1

Writing programs (or programming) is a very creative and rewarding activity. You can write programs for many reasons, ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem.

Programmers add an operating system and a set of applications to the hardware and we end up with a Personal Digital Assistant that is quite helpful and capable of helping us do many different things.

If we knew this language, we could tell the computer to do tasks on our behalf that were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

For a computer, the opposite is true, reading and understanding text from a piece of paper is hard for a computer to do but counting the words and telling you how many times the most used word was used is very easy for the computer:

Our “personal information analysis assistant” quickly told us that the word “to” was used sixteen times in the first three paragraphs of this chapter

The Central Processing Unit (or CPU) is the part of the computer that is built to be obsessed with “what is next?” If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask “What next?” three billion times per second. You are going to have to learn how to talk fast to keep up with the CPU

. • The Main Memory is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the information stored in the main memory vanishes when the computer is turned off.

• The Secondary Memory is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).

• The Input and Output Devices are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we interact with the computer. •

These days, most computers also have a Network Connection to retrieve information over a network. We can think of the network as a very slow place to store and retrieve data that might not always be “up”. So in a sense, the network is a slower and at times unreliable form of Secondary Memory.

You need to be the person who answers the CPU’s “What next?” question. But it would be very uncomfortable to shrink you down to 5mm tall and insert you into the computer just so you could issue a command three billion

First, you need to know the programming language (Python) - you need to know the vocabulary and the grammar. You need to be able to spell the words in this new language properly and know how to construct well-formed “sentences” in this new language.

The definition of a program at its most basic is a sequence of Python statements that have been crafted to do something. Even our simple hello.py script is a program. It is a one-line program and is not particularly useful, but in the strictest definition, it is a Python program.

Syntax errors These are the first errors you will make and the easiest to fix. A syntax error means that you have violated the “grammar” rules of Python. Python does its best to point right at the line and character where it noticed it was confused. The only tricky bit of syntax errors is that sometimes the mistake that needs fixing is actually earlier in the program than where Python noticed it was confused. So the line and character that Python indicates in a syntax error may just be a starting point for your investigation.

2

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and “Hello, World!” These values belong to different types: 2 is an integer, and “Hello, World!” is a string, so called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. The print statement also works for integers. We use the python command to start the interpreter.

Well, that’s not what we expected at all! Python interprets 1,000,000 as a commaseparated sequence of integers, which it prints with spaces between. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn’t do the “right” thing.

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value. An assignment statement creates new variables and gives them values:

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you’ll see why later).

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what’s wrong with class? It turns out that class is one of Python’s keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names. Python reserves 35 keywords: and del from None True as elif global nonlocal try assert else if not while break except import or with class False in pass yield continue finally is raise async def for lambda return await

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called input that gets input from the keyboard1 . When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter,

3

A boolean expression is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

True and False are special values that belong to the class bool; they are not strings: >>> type(True) >>> type(False)

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10. n%2 == 0 or n%3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3. Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y. Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as “true.” >>> 17 and True True This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it until you are sure you know what you are doing.

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

if x < y: print('x is less than y') elif x > y: print('x is greater than y') else: print('x and y are equal')

One conditional can also be nested within another. We could have written the three-branch example like this: if x == y: print('x and y are equal') else: if x < y: print('x is less than y') else: print('x is greater than y

When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

4

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name. We have already seen one example of a function call: >>> type(32)

The name of the function is type. The expression in parentheses is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type function, is the type of the argument. It is common to say that a function “takes” an argument and “returns” a result. The result is called the return value

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

Python has a math module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

This statement creates a module object named math. If you print the module object, you get some information about it:

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called log that computes logarithms base e. The second example finds the sine of radians. The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be deterministic. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them fruitful functions. Other functions, like print\_twice, perform an action but don’t return a value. They are called void functions. When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

5

More formally, here is the flow of execution for a while statement: 1. Evaluate the condition, yielding True or False. 2. If the condition is false, exit the while statement and continue execution at the next statement. 3. If the condition is true, execute the body and then go back to step 1. This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. For the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times. The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration. Here is an example of a loop that copies its input until the user types “done”, but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body: friends = ['Joseph', 'Glenn', 'Sally'] for friend in friends: print('Happy New Year:', friend) print('Done!') In Python terms, the variable friends is a list1 of three strings and the for loop goes through the list and executes the body once for each of the three strings in the list resulting in this output: 1We will examine lists in more detail in a later chapter. Happy New Year: Joseph Happy New Year: Glenn Happy New Year: Sally Done!

Often we use a for or while loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

In this loop we do use the iteration variable. Instead of simply adding one to the count as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable total, it contains the “running total of the values so far”. So before the loop starts total is zero because we have not yet seen any values, during the loop total is the running total, and at the end of the loop total is the overall total of all the values in the list.

To find the largest value in a list or sequence, we construct the following loop: largest = None print('Before:', largest) for itervar in [3, 41, 12, 9, 74, 15]: if largest is None or itervar > largest : largest = itervar print('Loop:', itervar, largest) print('Largest:', largest) When the program executes, the output is as follows: Before: None Loop: 3 3

6

For most people, the first letter of “banana” is “b”, not “a”. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero. >>> letter = fruit[0] >>> print(letter) b So “b” is the 0th letter (“zero-th”) of “banana”, “a” is the 1th letter (“one-th”), and “n” is the 2th (“two-th”) letter. You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get: >>> letter = fruit[1.5] TypeError: string indices must be integers

len is a built-in function that returns the number of characters in a string: >>> fruit = 'banana' >>> len(fruit) 6 To get the last letter of a string, you might be tempted to try something like this: >>> length = len(fruit) >>> last = fruit[length] IndexError: string index out of range The reason for the IndexError is that there is no letter in “banana” with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a while loop:

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second: >>> 'a' in 'banana' True >>> 'seed' in 'banana' False

7

When we want to read or write a file (say on your hard drive), we first must open the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file mbox.txt, which should be stored in the same folder that you are in when you start Python. You can download this file from [www.py4e.com/code3/mbox.txt](http://www.py4e.com/code3/mbox.txt)

If the open is successful, the operating system returns us a file handle. The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

In Python, we represent the newline character as a backslash-n in string constants. Even though this looks like two characters, it is actually a single character. When we look at the variable by entering “stuff” in the interpreter, it shows us the \n in the string, but when we use print to show the string, we see the string broken into two lines by the newline character.

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

As your file processing programs get more complicated, you may want to structure your search loops using continue. The basic idea of the search loop is that you are looking for “interesting” lines and effectively skipping “uninteresting” lines. And then when we find an interesting line, we do something with that line.

8

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items. There are several ways to create a new list; the simplest is to enclose the elements in square brackets (“[" and “]”): [10, 20, 30, 40] ['crunchy frog', 'ram bladder', 'lark vomit'] The first example is a list of four integers. The second is a list of three strings. The elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0: >>> print(cheeses[0]) Cheddar Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n − 1, where n is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

The + operator concatenates lists: >>> a = [1, 2, 3] >>> b = [4, 5, 6] >>> c = a + b >>> print(c) [1, 2, 3, 4, 5, 6] Similarly, the \* operator repeats a list a given number of times:

The slice operator also works on lists: >>> t = ['a', 'b', 'c', 'd', 'e', 'f'] >>> t[1:3] ['b', 'c'] >>> t[:4] ['a', 'b', 'c', 'd'] >>> t[3:] ['d', 'e', 'f']

Python provides methods that operate on lists. For example, append adds a new element to the end of a list: >>> t = ['a', 'b', 'c'] >>> t.append('d') >>> print(t) ['a', 'b', 'c', 'd']

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops: >>> nums = [3, 41, 12, 9, 74, 15] >>> print(len(nums)) 6 >>> print(max(nums)) 74 >>> print(min(nums)) 3 >>> print(sum(nums)) 154 >>> print(sum(nums)/len(nums))

9

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type. You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item. As an example, we’ll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings. The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

This line creates an item that maps from the key 'one' to the value “uno”. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included. But soft what light through yonder window breaks It is the east and Juliet is the sun Arise fair sun and kill the envious moon Who is already sick and pale with grief We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary. You will see that we have two for loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called nested loops because one of the loops is the outer loop and the other loop is the inner loop. Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating “more quickly” and the outer loop as iterating more slowly. The combination of the two nested loops ensures that we will count ever

Since the Python split function looks for spaces and treats words as tokens separated by spaces, we would treat the words “soft!” and “soft” as different words and create a separate dictionary entry for each word. Also since the file has capitalization, we would treat “who” and “Who” as different words with different counts. We can solve both these problems by using the string methods lower, punctuation, and translate. The translate is the most subtle of the methods. Here is the documentation for translate: line.translate(str.maketrans(fromstr, tostr, deletestr)) Replace the characters in fromstr with the character in the same position in tostr and delete all characters that are in deletestr. The fromstr and tostr can be empty strings and the deletestr parameter can be omitted. We will not specify the tostr but we will use the deletestr parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers “punctuation”:

10

A tuple1 is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Tuples are also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

The first loop builds a list of tuples, where each tuple is a word preceded by its length. sort compares the first element, length, first, and only considers the second element to break ties. The keyword argument reverse=True tells sort to go in decreasing order. The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in reverse alphabetical order, so “what” appears before “soft” in the following list. The output of the program is as follows:

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

14

Like many aspects of programming, it is necessary to learn the concepts of object oriented programming before you can use them effectively. You should approach this chapter as a way to learn some terms and concepts and work through a few simple examples to lay a foundation for future learning. The key outcome of this chapter is to have a basic understanding of how objects are constructed and how they function and most importantly how we make use of the capabilities of objects that are provided to us by Python and Python libraries.

Instead of focusing on what these lines accomplish, let’s look at what is really happening from the point of view of object-oriented programming. Don’t worry if the following paragraphs don’t make any sense the first time you read them because we have not yet defined all of these terms. The first line constructs an object of type list, the second and third lines call the append() method, the fourth line calls the sort() method, and the fifth line retrieves the item at position 0. The sixth line calls the \_\_getitem\_\_() method in the stuff list with a parameter of zero.

We read the URL into a string and then pass that into urllib to retrieve the data from the web. The urllib library uses the socket library to make the actual network connection to retrieve the data. We take the string that urllib returns and hand it to BeautifulSoup for parsing. BeautifulSoup makes use of the object html.parser1 and returns an object. We call the tags() method on the returned object that returns a dictionary of tag objects. We loop through the tags and call the get() method for each tag to print out the href attribute. We can draw a picture of this program and how the objects work together.

One of the advantages of the object-oriented approach is that it can hide complexity. For example, while we need to know how to use the urllib and BeautifulSoup code, we do not need to know how those libraries work internally. This allows us to focus on the part of the problem we need to solve and ignore the other parts of the program.