Chapter 2 Writing Simple Programs

Objectives

- To know the steps in an orderly software development process.
- To understand programs following the input, process, output (IPO) pattern and be able to modify them in simple ways.
- To understand the rules for forming valid Python identifiers and expressions.
- To be able to understand and write Python statements to output information to the screen, assign values to variables, get information entered from the keyboard, and perform a counted loop.

2.1 The Software Development Process

As you saw in the previous chapter, it is easy to run programs that have already been written. The harder part is actually coming up with a program in the first place. Computers are very literal, and they must be told what to do right down to the last detail. Writing large programs is a daunting challenge. It would be almost impossible without a systematic approach.

The process of creating a program is often broken down into stages according to the information that is produced in each phase. In a nutshell, here's what you should do:

Analyze the Problem Figure out exactly what the problem to be solved is. Try to understand as much as possible about it. Until you really know what the problem is, you cannot begin to solve it.

Determine Specifications Describe exactly what your program will do. At this point, you should not worry about *how* your program will work, but rather about deciding exactly *what* it will accomplish. For simple programs this involves carefully describing what the inputs and outputs of the program will be and how they relate to each other.

Create a Design Formulate the overall structure of the program. This is where the *how* of the program gets worked out. The main task is to design the algorithm(s) that will meet the specifications.

Implement the Design Translate the design into a computer language and put it into the computer. In this book, we will be implementing our algorithms as Python programs.

Test/Debug the Program Try out your program and see if it works as expected. If there are any errors (often called *bugs*), then you should go back and fix them. The process of locating and fixing errors is called *debugging* a program. During the debugging phase, your goal is to find errors, so you should try everything you can think of that might "break" the program. It's good to keep in mind the old maxim: "Nothing is foolproof because fools are too ingenious."

Maintain the Program Continue developing the program in response to the needs of your users. Most programs are never really finished; they keep evolving over years of use.

2.2 Example Program: Temperature Converter

Let's go through the steps of the software development process with a simple real-world example involving a fictional computer science student, Susan Computewell.

Susan is spending a year studying in Germany. She has no problems with language, as she is fluent in many languages (including Python). Her problem is that she has a hard time figuring out the temperature in the morning so that she knows how to dress for the day. Susan listens to the weather report each

morning, but the temperatures are given in degrees Celsius, and she is used to Fahrenheit.

Fortunately, Susan has an idea to solve the problem. Being a computer science major, she never goes anywhere without her laptop computer. She thinks it might be possible that a computer program could help her out.

Susan begins with an analysis of her problem. In this case, the problem is pretty clear: the radio announcer gives temperatures in degrees Celsius, but Susan only comprehends temperatures that are in degrees Fahrenheit.

Next, Susan considers the specifications of a program that might help her out. What should the input be? She decides that her program will allow her to type in the temperature in degrees Celsius. And the output? The program will display the temperature converted into degrees Fahrenheit. Now she needs to specify the exact relationship of the output to the input.

Susan does some quick figuring. She knows that 0 degrees Celsius (freezing) is equal to 32 degrees Fahrenheit, and 100 Celsius (boiling) is equal to 212 Fahrenheit. With this information, she computes the ratio of Fahrenheit to Celsius degrees as $\frac{212-32}{100-0} = \frac{180}{100} = \frac{9}{5}$. Using F to represent the Fahrenheit temperature and C for Celsius, the conversion formula will have the form $F = \frac{9}{5}C + k$ for some constant k. Plugging in 0 and 32 for C and F, respectively, Susan immediately sees that $k = 32$. So, the final formula for the relationship is $F = \frac{9}{5}C + 32$. That seems an adequate specification.

Notice that this describes one of many possible programs that could solve this problem. If Susan had background in the field of Artificial Intelligence (AI), she might consider writing a program that would actually listen to the radio announcer to get the current temperature using speech recognition algorithms. For output, she might have the computer control a robot that goes to her closet and picks an appropriate outfit based on the converted temperature. This would be a much more ambitious project, to say the least!

Certainly, the robot program would also solve the problem identified in the problem analysis. The purpose of specification is to decide exactly what this particular program will do to solve a problem. Susan knows better than to just dive in and start writing a program without first having a clear idea of what she is trying to build.

Susan is now ready to design an algorithm for her problem. She immediately realizes that this is a simple algorithm that follows a standard pattern: *Input, Process, Output* (IPO). Her program will prompt the user for some input information (the Celsius temperature), process it to convert to a Fahrenheit temperature, and then output the result by displaying it on the computer screen.

Susan could write her algorithm down in a computer language. However, the precision required to write it out formally tends to stifle the creative process of developing the algorithm. Instead, she writes her algorithm using *pseudocode*. Pseudocode is just precise English that describes what a program does. It is meant to communicate algorithms without all the extra mental overhead of getting the details right in any particular programming language.

Here is Susan's completed algorithm:

```
Input the temperature in degrees Celsius (call it celsius)
Calculate fahrenheit as (9/5)celsius + 32
Output fahrenheit
```

The next step is to translate this design into a Python program. This is straightforward, as each line of the algorithm turns into a corresponding line of Python code.

```
# convert.py
#    A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = eval(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")

main()
```

See if you can figure out what each line of this program does. Don't worry if some parts are a bit confusing. They will be discussed in detail in the next section.

After completing her program, Susan tests it to see how well it works. She uses inputs for which she knows the correct answers. Here is the output from two of her tests:

```
What is the Celsius temperature? 0
The temperature is 32.0 degrees Fahrenheit.

What is the Celsius temperature? 100
The temperature is 212.0 degrees Fahrenheit.
```

You can see that Susan used the values of 0 and 100 to test her program. It looks pretty good, and she is satisfied with her solution. She is especially pleased that no debugging seems necessary (which is very unusual).

2.3 Elements of Programs

Now that you know something about the programming process, you are *almost* ready to start writing programs on your own. Before doing that, though, you need a more complete grounding in the fundamentals of Python. The next few sections will discuss technical details that are essential to writing correct programs. This material can seem a bit tedious, but you will have to master these basics before plunging into more interesting waters.

2.3.1 Names

You have already seen that names are an important part of programming. We give names to modules (e.g., `convert`) and to the functions within modules (e.g., `main`). Variables are used to give names to values (e.g., `celsius` and `fahrenheit`). Technically, all these names are called *identifiers*. Python has some rules about how identifiers are formed. Every identifier must begin with a letter or underscore (the "_" character) which may be followed by any sequence of letters, digits, or underscores. This implies that a single identifier cannot contain any spaces.

According to these rules, all of the following are legal names in Python:

```
x
celsius
spam
spam2
SpamAndEggs
Spam_and_Eggs
```

Identifiers are case-sensitive, so `spam`, `Spam`, `sPam`, and `SPAM` are all different names to Python. For the most part, programmers are free to choose any name that conforms to these rules. Good programmers always try to choose names that describe the thing being named.

One other important thing to be aware of is that some identifiers are part of Python itself. These names are called *reserved words* or *keywords* and cannot be

used as ordinary identifiers. The complete list of Python keywords is shown in Table 2.1.

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

Table 2.1: Python Keywords

2.3.2 Expressions

Programs manipulate data. So far, we have seen two different kinds of data in our example programs: numbers and text. We'll examine these different data types in great detail in later chapters. For now, you just need to keep in mind that all data has to be stored on the computer in some digital format, and different types of data are stored in different ways.

The fragments of program code that produce or calculate new data values are called *expressions*. The simplest kind of expression is a *literal*. A literal is used to indicate a specific value. In `chaos.py` you can find the numbers `3.9` and `1`. The `convert.py` program contains `9`, `5`, and `32`. These are all examples of numeric literals, and their meaning is obvious: `32` represents, well, 32 (the number 32).

Our programs also manipulated textual data in some simple ways. Computer scientists refer to textual data as *strings*. You can think of a string as just a sequence of printable characters. A string literal is indicated in Python by enclosing the characters in quotation marks (`""`). If you go back and look at our example programs, you will find a number of string literals such as: `"Hello"` and `"Enter a number between 0 and 1: "`. These literals produce strings containing the quoted characters. Note that the quotes themselves are not part of the string. They are just the mechanism to tell Python to create a string.

The process of turning an expression into an underlying data type is called *evaluation*. When you type an expression into a Python shell, the shell evaluates the expression and prints out a textual representation of the result. Consider

this small interaction:

```
>>> 32
32
>>> "Hello"
'Hello'
>>> "32"
'32'
```

Notice that when the shell shows the value of a string, it puts the sequence of characters in single quotes. This is a way of letting us know that the value is actually text, not a number (or other data type). In the last interaction, we see that the expression `"32"` produces a string, not a number. In this case, Python is actually storing the characters "3" and "2," not a representation of the number 32. If that's confusing right now, don't worry too much about it; it will become clearer when we discuss these data types in later chapters.

A simple identifier can also be an expression. We use identifiers as variables to give names to values. When an identifier appears as an expression, its value is retrieved to provide a result for the expression. Here is an interaction with the Python interpreter that illustrates the use of variables as expressions:

```
>>> x = 5
>>> x
5
>>> print(x)
5
>>> print(spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

First the variable `x` is assigned the value 5 (using the numeric literal `5`). In the second line of interaction, we are asking Python to evaluate the expression `x`. In response, the Python shell prints out `5`, which is the value that was just assigned to `x`. Of course, we get the same result when we explicitly ask Python to print `x` using a `print` statement. The last interaction shows what happens when we try to use a variable that has not been assigned a value. Python cannot find a value, so it reports a *NameError*. This says that there is no value with that name. The important lesson here is that a variable must always be assigned a value before it can be used in an expression.

More complex and interesting expressions can be constructed by combining simpler expressions with *operators*. For numbers, Python provides the normal set of mathematical operations: addition, subtraction, multiplication, division, and exponentiation. The corresponding Python operators are `+`, `-`, `*`, `/`, and `**`. Here are some examples of complex expressions from `chaos.py` and `convert.py`:

```
3.9 * x * (1 - x)
9/5 * celsius + 32
```

Spaces are irrelevant within an expression. The last expression could have been written `9/5*celsius+32` and the result would be exactly the same. Usually it's a good idea to place some spaces in expressions to make them easier to read.

Python's mathematical operators obey the same rules of precedence and associativity that you learned in your math classes, including using parentheses to modify the order of evaluation. You should have little trouble constructing complex expressions in your own programs. Do keep in mind that only the round parentheses are allowed in numeric expressions, but you can nest them if necessary to create expressions like this.

```
((x1 - x2) / 2*n) + (spam / k**3)
```

By the way, Python also provides operators for strings. For example, you can "add" strings.

```
>>> "Bat" + "man"
'Batman'
```

This is called *concatenation*. As you can see, the effect is to create a new string that is the result of "gluing" the strings together. You'll see a lot more string operations in Chapter 5.

2.4 Output Statements

Now that you have the basic building blocks, identifier and expression, you are ready for a more complete description of various Python statements. You already know that information can be displayed on screen using Python's built-in function `print`. So far, we have looked at a few examples, but I have not yet explained the `print` function in detail. Like all programming languages, Python has a precise set of rules for the syntax (form) and semantics (meaning) of each

statement. Computer scientists have developed sophisticated notations called *meta-languages* for describing programming languages. In this book we will rely on a simple template notation to illustrate the syntax of various statements.

Since `print` is a built-in function, a `print` statement has the same general form as any other function invocation. We type the function name `print` followed by parameters listed in parentheses. Here is how the `print` statement looks using our template notation:

```
print(<expr>, <expr>, ..., <expr>)
print()
```

These two templates show two forms of the `print` statement. The first indicates that a `print` statement can consist of the function name `print` followed by a parenthesized sequence of expressions, which are separated by commas. The angle bracket notation (<>) in the template is used to indicate "slots" that are filled in by other fragments of Python code. The name inside the brackets indicates what is missing; `expr` stands for an expression. The ellipses ("...") indicate an indefinite series (of expressions, in this case). You don't actually type the dots. The second version of the statement shows that it's also legal to have a `print` without any expressions to print.

As far as semantics are concerned, a `print` statement displays information in textual form. Any supplied expressions are evaluated left to right, and the resulting values are displayed on a line of output in a left-to-right fashion. By default, a single blank space character is placed between the displayed values. As an example, this sequence of `print` statements:

```
print(3+4)
print(3, 4, 3 + 4)
print()
print("The answer is", 3 + 4)
```

produces this output:

```
7
3 4 7

The answer is 7
```

The last statement illustrates how string literal expressions are often used in print statements as a convenient way of labeling output.

Notice that successive `print` statements normally display on separate lines of the screen. A bare `print`(no parameters) produces a blank line of output. Underneath, what's really happening is that the `print` function automatically appends some ending text after all of the supplied expressions are printed. By default, that ending text is a special marker character (denoted as `"\n"`) that signals the end of a line. We can modify that behavior by including an additional parameter that explicitly overrides this default. This is done using a special syntax for named or *keyword* parameters.

A template for the `print` statement including the keyword parameter to specify the ending-text looks like this:

```
print(<expr>, <expr>, ..., <expr>, end="\n")
```

The keyword for the named parameter is `end` and it is given a value using = notation, similar to variable assignment. Notice in the template I have shown its default value, the end-of-line character. This is a standard way of showing what value a keyword parameter will have when it is not explicitly given some other value.

One common use of the `end` parameter in `print` statements is to allow multiple `print`s to build up a single line of output. For example:

```
print("The answer is", end=" ")
print(3 + 4)
```

produces the single line of output:

```
The answer is 7
```

Notice how the output from the first print statement ends with a space (`" "`) rather than an end-of-line. The output from the second statement appears immediately following the space.

2.5 Assignment Statements

One of the most important kinds of statements in Python is the assignment statement. We've already seen a number of these in our previous examples.

2.5.1 Simple Assignment

The basic assignment statement has this form: