Unit 2

Programming Basics

Learning Outcomes

- Use the Python software to get programs running.
- Create programs that perform simple calculations.
- Use variables to store information in a program.
- Create programs that take input from the user.
- Explain how computers store information in binary.
- Take a simple problem and create an algorithm that solves it.
- Implement that algorithm in Python.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapter 2 in *How to Think Like a Computer Scientist.*

Topic 2.1 Starting with Python

In this course, you will be using the *Python* programming language. You can download Python for free or use it in the lab. See Appendix A for more instructions on how to use the software.

One nice feature of Python is its *interactive interpreter*. You can start up Python and start typing in Python code. It will be executed immediately, and you will see the results.

You can also type Python code into a file and save it. Then, you can run it all at once. The interactive interpreter is generally used for exploring the language or testing ideas. Python code in a file can be run as an application and even double-clicked to run your program.

You will start by working with the Python interpreter. See Appendix A for instructions on getting Python running. When you start the Python interpreter, you'll see something like this:

```
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

The `>>>` is the *prompt*. Whenever you see it in the interpreter, you can type Python commands. When you press return, the command will be executed and you will be shown the result. Whenever you see the `>>>` prompt in examples, it's an example of what you'd see in the interpreter if you typed the code after the `>>>`.

For some reason, when people are taught to program, the first program they see is one that prints the words "Hello world" on the screen. Not wanting to rock the boat, you will do that too. Here's what it looks like in the Python interpreter:

```
>>> print "Hello world"
Hello world
```

The stuff after the prompt is the first line of Python code you have seen. You could have also typed it into a text editor, named the file hello.py and run it.

The `print` command in Python is used to put text on the screen. Whatever comes after it will be printed on the screen.

Any text in quotes, like `"Hello world"` in the example, is called a *string*. Strings are just a bunch of *characters*. Characters are letters, numbers, spaces, and punctuation. Strings have to be placed in quotes to be distinguished from Python commands. If we had left out the quotes, Python would have complained that it didn't know what "`Hello`" meant, since there is no built-in command called `Hello`.

The Interpreter vs. the Editor

When you use Python's *IDLE* (Integrated DeveLopment Environment), the first window that you see is the interactive interpreter. That's the window with the >>> prompt where you can type Python code, press return, and see its result. You can use this to test small sections of Python code to make sure they do what you expect.

If you create a "New Window", the window you create is a file editing window. It doesn't have a prompt or anything else. You can type Python code here and *save* it as a .py file. You can run a Python .py file by double clicking it. You can also press F5 while editing it in IDLE to run it. You should use an editor window to write whole programs.

Check-Up Questions

- ▶ Type `print "Hello world!"` into an editor window in IDLE and save it as hello.py file. Run it with Python.

  If you're using Windows and you run the program by double-clicking the file, the output window might disappear before you can see the results. You can stop this from happening by running the program in IDLE or by waiting for the user to press return before ending the program. We'll talk about how to do that in the next topic.

- ▶ Add a few more `print` statements to your hello.py program (one per line). Run it and see what happens.

Statements

If you did the "Check-Up Questions" above, you would have created a file containing one line:

```
print "Hello world!"
```

This line is a Python *statement*.

Statements are the basic building blocks of Python programs. Each statement expresses a part of the overall algorithm that you're implementing. The `print` statement is the first one you have seen so far, but there are many others. Each one lets you express a different idea in such a way that the computer can complete it.

When you run a Python program (i.e., code you typed in a `.py` file and saved), the statements are executed in the order they appear in the file. So, the Python program

```
print "Hello world!"
print "I'm a Python program that prints stuff."
```

. . . will produce this output:

```
Hello world!
I'm a Python program that prints stuff.
```

Topic 2.2 Doing Calculations

In order to implement the algorithm in Figure 1.3, you will need to be able to calculate $guess + 1$ and $\lfloor (smallest + largest)/2 \rfloor$.

Python can do calculations like this. An *expression* is any kind of calculation that produces a result. Here are some examples of using expressions in `print` statements in the Python interpreter:

```
>>> print 10 - 2
8
>>> print 15/3
5
>>> print 25+19*5
120
>>> print 10.2 / 2 / 2
2.55
```

The Python *operators* `+`, `-`, `*`, and `/` perform addition, subtraction, multiplication, and division, as you might expect. Note that they do order-of-operations they way you'd expect, too:

```
>>> print 25+19*5
120
>>> print 25+(19*5)
120
>>> print (25+19)*5
220
```

Parentheses do the same thing they do when you're writing math: they wrap up part of a calculation so it's done first. Note that a number by itself is an expression too.

```
>>> print 18
18
```

Now, in Figure 1.3, suppose that the current value of *smallest* is 76 and *largest* is 100. Then, we can at least do the right calculation:

```
>>> print (76+100)/2
88
```

Python can do calculations on strings too.

```
>>> print "An" + "Expression"
AnExpression
>>> print "An " + 'Expression'
An Expression
>>> print 'ABC' * 4
ABCABCABCABC
```

Note that when you enter a string, it has to be wrapped up in quotes. This is the only way Python can distinguish between characters that are part of a string or part of the expression itself. In Python, single quotes (`'`) and double quotes (`"`) can be used interchangeably.

If you forget the quotes around a string, you'll probably get an error message:

```
>>> print An + 'Expression'
NameError: name 'An' is not defined
```

Here, Python is telling us that it doesn't know the word "`An`". It does know words like `print` and a few others. If you want to talk about a bunch of characters as part of a string, they have to be surrounded by quotes. So, even when a number, or anything else is in quotes, it is treated like a string (which makes sense, since strings go in quotes):

```
>>> print 120 * 3
360
>>> print "120" * 3
120120120
>>> print "120 * 3"
120 * 3
```

Functions

Python can also use *functions* as part of expressions. These work like functions in mathematics: you give the function some *arguments*, and something is done to calculate the result. The result that the function gives back is called its *return value*.

For example, in Python, there is a built-in function `round` that is used to round off a number to the nearest integer:

```
>>> print round(13.89)
14.0
>>> print round(-4.3)
-4.0
>>> print round(1000.5)
1001.0
```

Functions can take more than one argument. The `round` function can take a second argument (an *optional argument*) that indicates the number of decimal places it should round to. For example,

```
>>> print round(12.3456, 1)
12.3
>>> print round(12.3456, 2)
12.35
>>> print round(12.3456, 5)
12.3456
```

In these examples, the value is rounded to the nearest 0.1, 0.01, and 0.00001. For this function, if you don't indicate the optional argument, its default is 0. The *default value* for optional arguments depends on the function.

Functions can take any type of information as their argument and can return any type. For example, Python's `len` function will return the length of a string, i.e. how many characters it has:

```
>>> print len("hello")
5
>>> print len("-<()>-")
6
>>> print len("")
0
```

There are many other ways to do calculations on numbers and strings than we have seen here. You will see more as you learn more about programming. You will see some more functions as you need them.

Check-Up Questions

▶ Try `print`ing the results of some other expressions. Check the calculations by hand and make sure the result is what you expect.

▶ Try some of the above string expressions, swapping the single quotes for double quotes and vice-versa. Convince yourself that they really do the same thing.

▶ Some of the examples above "multiply" a string by a number (like `"cow"*3`). The result is repetition of the string. What happens if you multiply a number by a string (`3*"cow"`)? What about a string by a string (`"abc"*"def"`)?

Topic 2.3 Storing Information

You aren't going to want to always print out the result of a calculation like we did in Topic 2.2. Sometimes, you need to perform a calculation to be used later, without needing to display the results right away. You might also want to ask the user a question and remember their answer until you need it.

For example, in the algorithm in Figure 1.2, you want to calculate values for *smallest*, *largest*, and *guess* and store those results. You also need to ask the user for their answer and store the result. You need to keep all of those in the computer's memory.

Whenever we need the computer to temporarily remember some information in a program, we will use a *variable.* A variable is a way for you to reserve a little bit of the computer's memory to *store* the information you need.

You will give variables names that you will use to refer to them later. For example, if you ask the user for their age and want to store their input, you might use a variable named "`age`". The name of the variable should describe and somehow indicate what it represents.

To put a value in a variable, a *variable assignment statement* is used. For example, to put the result of the calculation `14/2` into a variable named `quotient`,

```
quotient = 14/2
```

In a variable assignment statement, put the name of the variable you want to change on the left, an equals sign, and the new value on the right.

You can use any expression to calculate the value that will be stored in the variable. Variables can store any kind of information that Python can manipulate. So far we have seen numbers and strings.

Be careful: Only the *result* of the calculation is stored, not the whole calculation.

To use the value that's stored in a variable, you just have to use its name. If a variable name is used in an expression, it is replaced with the stored value.

```
>>> num = 7
>>> word = "yes"
>>> print num - 3
4
>>> print word + word
yesyes
>>> num = 4
>>> print num - 3
1
```

Note that you can change the value in a variable. In the above example, `num` was first set to 7 and then changed to 4. Notice that the variable `num` was holding a number and `word` was holding a string. You can change the kind of information a variable holds by doing a variable assignment as well.

Topic 2.4 Types

As noted above and in Topic 2.2, Python treats numbers (like `2`, `-10`, and `3.14`) differently than strings (like `"abc"`, `"-10"`, and `""`). For example, you can divide two numbers, but it doesn't make sense to divide strings.

```
>>> print 10/2
5
>>> print "abc" / 2
TypeError: unsupported operand type(s) for /: 'str' and
'int'
```

Numbers and strings are two different *types* of information that Python can manipulate. *String* variables are used to hold text or collections of characters.

In Python, a `TypeError` indicates that you've used values whose types can't be used with the given operation. The type of the values given to an operator can change the way it works. In Topic 2.2, you saw that the `+` operator does different things on numbers (addition) and strings (joining).

In fact, the numeric values that Python stores aren't as simple as just "numbers". Have a look at this example from the Python interpreter:

```
>>> print 10/2
5
>>> print 10/3
3
>>> print 10.0/3
3.33333333333
```

Why does Python give a different answer for `10/3` than it does for `10.0/3`? The division operation does different things with *integers* than with *floating point* values.

Integers are numbers without any fraction part. So, `10`, `0`, and `-100` are all integers. Numbers with fractional parts, like `3.14`, `-0.201`, and `10.0`, are stored as floating point values. These two types are represented differently in the computer's memory, as we will discuss in Topic 2.6.

That's why Python comes up with different answers for `10/3` and `10.0/3`: there are different types of values given. In the case of integer division (`10/3`), the rule is that the result must be an integer. The floating point result has its fractional part rounded down to give the integer `3`. For floating point division, the result can have a fractional part, so the result is what you'd probably expect.

There is a built-in function called `type` that will tell you the type of an object in Python. Try `type(10/3)` and `type(10.0/3)`.

When implementing the pseudocode in Figure 1.3, you can actually use this to make sure the calculation *guess* rounds down to the next integer.

Note that you can trick Python into treating a whole number like a floating point number by giving it a fractional part with you type it. So `10` is an integer (or "*int*" for short), but `10.0` is a floating point value (or "*float*").

Type Conversion

Sometimes, you'll find you have information of one type, but you need to convert it to another.

For example, suppose you want to calculate the average of several integers. You would do the same thing you would do by hand: add up the numbers and divide by the number of numbers. Suppose you had found the sum of 10 numbers to be 46, leaving the values 46 in `sum` and 10 in `num`. If you try to divide these numbers in Python, you'll get the result 4, since you're dividing two integers. Really, you want the result 4.6, which you would get if at least one of the values being divided was a float.

There are Python functions that can be used to change a value from one type to another. You can use these in an expression to get the type you want. The function `int()` converts to an integer, `float()` converts to a floating point value, and `str()` converts to a string. For example,

```
>>> float(10)
10.0
>>> str(10)
'10'
>>> int('10')
10
>>> int(83.7)
83
>>> str(123.321)
'123.321'
>>> int("uhoh")
ValueError: invalid literal for int(): uhoh
```

As you can see, these functions will do their best to convert whatever you give them to the appropriate type. Sometimes, that's just not possible: there's no way to turn `"uhoh"` into an integer, so it causes an error.

In the example of calculating the average, we can do a type conversion to get the real average:

```
>>> total = 46
>>> num = 10
>>> print total/num
4
>>> print float(total)/num
4.6
>>> print float(total/num)
4.0
```

Have a closer look at the last example. Since the conversion is wrapped around the whole calculation, only the result is converted. So, Python divides the integers 46 and 10 to get 4. This is converted to the floating point value 4.0. In order for the floating point division to work, at least one of the numbers going *into* the division must be a floating point value.

Converting numbers to strings is often handy when printing. Again, suppose you have 46 in the variable `total` and you want to print out a line like

```
The sum was 46.
```

You can print out multiple values with the comma, but they are separated by spaces:

```
>>> print "The sum was", total, "."
The sum was 46 .
```

Note that there's a space between the `46` and the period. You can remove this by combining strings to get the result we want:

```
>>> print "The sum was " + str(total) + "."
The sum was 46.
```

When Python joins strings, it doesn't add any extra spaces. You have to convert `total` to a string here since Python doesn't know how to add a string and a number:

```
>>> print "The sum was " + total + "."
TypeError: cannot concatenate 'str' and 'int' objects
```

The word *concatenate* means "join together". When you use the `+` on strings, it's not really adding them, it's joining them. That's called concatenation.

Topic 2.5 User Input

Something else you will need to do to implement the algorithm from Figure 1.3 is to get input from the user. You need to ask them if the number they're thinking of is larger, smaller or equal.

To do this in Python, use the `raw_input` function. This function will give the user whatever message you tell it to, wait for them to type a response and press enter, and return their response to your expression.

For example, this program will ask the user for their name and then say hello:

```
name = raw_input("What is your name? ")
print "Hello, " + name + "."
```

If you run this program, it will display "`What is your name? `" on the screen and wait for the user to respond. Their response will be stored in the variable `name`. For example,

```
What is your name? Julius
Hello, Julius.
```

If the user enters something else, that's what will go in the `name` variable,

```
What is your name? Joey Jo-Jo
Hello, Joey Jo-Jo.
```

In this guide, any input that the user types will be **set in bold, like this**.

Whenever you use the `raw_input` function, it will return a string. That's because as far as the interpreter is concerned, the user just typed a bunch of characters and that's exactly what a string is.

If you want to treat the user's input as an integer or floating point number, you have to use one of the type conversion functions described above. For example, if you ask the user for their height, you really want a floating point value, but we get a string. So, it must be converted:

```
m = float(raw_input("Enter your height (in metres): "))
inches = 39.37 * m
print "You are " + str(inches) + " inches tall."
```

When you run this program,

```
Enter your height (in metres): 1.8
You are 70.866 inches tall.
```

In this example, the user enters the string `"1.8"`, which is returned by the `raw_input` function. That is converted to the floating point number `1.8` by the `float` function. This is stored in the variable `m` (for "metres"). Once there is a floating point value in `m`, your program can do numeric calculations with it. The number of inches is calculated and the corresponding floating point number is stored in `inches`. To print this out, it is converted back to a string with the `str` function. Sometimes `print` will do the conversion for you, but it was done explicitly in this program.

Topic 2.6 How Computers Represent Information

You may be wondering why you have to care about all of the different types of values that Python can handle. Why should `25` be different from `25.0`? For that matter, how is the number `25` different from the string `"25"`?

The real difference here is in the way the computer stores these different kinds of information. To understand that, you need to know a little about how computers store information.

Binary

All information that is stored and manipulated with a computer is represented in *binary*, i.e. with zeros and ones. So, no matter what kind of information you work with, it has to be turned into a string of zeros and ones if you want to manipulate it with a computer.

Why just zeros and ones?

A computer's memory is basically a whole bunch of tiny rechargeable batteries (*capacitors*). These can either be discharged (0) or charged (1). It's fairly easy for the computer to look at one of these capacitors and decide if it's charged or not.

Prefix	Symbol	Factor
(no prefix)		$2^0 = 1$
kilo-	k	$2^{10} = 1024 \approx 10^3$
mega-	M	$2^{20} = 1048576 \approx 10^6$
giga-	G	$2^{30} = 1073741824 \approx 10^9$
tera-	T	$2^{40} = 1099511627776 \approx 10^{12}$

Figure 2.1: Prefixes for storage units.

It's possible to use the same technology to represent digits from 0 to 9, but it's very difficult to distinguish ten different levels of charge in a capacitor. It's also very hard to make sure a capacitor doesn't discharge a little to drop from a 7 to a 6 without noticing. So, modern computers don't do this. They just use a simpler system with two levels of charge and end up with zeros and ones.

Hard disks and other storage devices also use binary for similar reasons. Computer networks do as well.

A single piece of storage that can store a zero or one is called a *bit*. Since a bit is a very small piece of information to worry about, bits are often grouped. It's common to divide a computer's memory into eight-bit groups called *bytes*. So, 00100111 and 11110110 are examples of bytes.

When measuring storage capacity, the number of bits or bytes quickly becomes large. Figure 2.1 show the prefixes that are used for storage units and what they mean.

For example, "12 megabytes" is

$$12 \times 2^{20} \text{ bytes} = 12582912 \text{ bytes} = 12582912 \times 8 \text{ bits} = 100663296 \text{ bits}.$$

Note that the values in Figure 2.1 are slightly different than the usual meaning of the metric prefixes. One kilometre is exactly 1000 metres, not 1024 metres. When measuring storage capacities in computers, the 1024 version of the metric prefixes is usually used.

That statement isn't entirely true. Hard drive makers, for instance, generally use units of 1000 because people would generally prefer a "60 gigabyte" drive to a "55.88 gigabyte" drive ($60 \times 10^{12} = 55.88 \times 2^{30}$).

Unsigned Integers

Once you have a bunch of bits, you can use them to represent numbers.

First, think about the way you count with regular numbers: 1, 2, 3, 4, 5. . . . Consider the number 157. What does each of the digits in that number mean? The "1" is one hundred, "5" is five tens, and "7" is seven ones: $157 = (1 \times 10^2) + (5 \times 10) + (7 \times 1)$.

As you go left from one place to the next, the value it represents is multiplied by 10. Each digit represents the number of 1s, 10s, 100s, 1000s. . . . The reason the values increase by a factor of 10 is that there are ten possible digits in each place: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This is called *decimal* or *base 10 arithmetic*. (The "dec-" prefix in latin means 10.)

Applying the same logic, there is a counting system with bits, *binary* or *base 2 arithmetic* ("bi-" means 2). The rightmost bit will be the number of 1s, the next will be the number of 2s, then 4s, 8s, 16s, and so on. Binary values are often written with a little 2 (a subscript), to indicate that they are base 2 values: 101_2. If there's any possibility for confusion, base 10 values are written with a subscript 10: 34_{10}.

To convert binary values to decimal, do the same thing you did above, substituting 2s for the 10s:

$$\begin{aligned} 1001_2 &= (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 8 + 1 \\ &= 9_{10} \,. \end{aligned}$$

The base 2 value 1001_2 is equal to 9_{10}. Another example with a larger number:

$$\begin{aligned} 10011101_2 &= (1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + \\ &\qquad (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 128 + 16 + 8 + 4 + 1 \\ &= 157_{10} \,. \end{aligned}$$

So, 10011101 is the base 2 representation of the number 157. Any positive whole number can be represented this way, given enough bits. All of the values that can be represented with four bits are listed in Figure 2.2.

You should be able to convince yourself that for any group of n bits, there are 2^n different possible values that can be stored in those bits. So, n bits

binary	**decimal**	**binary**	**decimal**
1111	15	0111	7
1110	14	0110	6
1101	13	0101	5
1100	12	0100	4
1011	11	0011	3
1010	10	0010	2
1001	9	0001	1
1000	8	0000	0

Figure 2.2: The four-bit unsigned integer values.

$$\begin{array}{r} 1010 \\ +\,0100 \\ \hline 1110 \end{array} \qquad \begin{array}{r} {\scriptstyle 1} \\ 1011 \\ +\,0010 \\ \hline 1101 \end{array} \qquad \begin{array}{r} {\scriptstyle 1}{\scriptstyle 1} \\ 1101 \\ +\;\;0101 \\ \hline 10010 \end{array}$$

Figure 2.3: Some examples of binary addition

can represent any number from 0 to $2^n - 1$. Other common groupings are of 16 bits (which can represent numbers 0 to $2^{16} - 1 = 65535$) and 32 bits (which can represent numbers 0 to $2^{32} - 1 = 4294967295$).

The computer can do operations like addition and subtraction on binary integers the same way you do with decimal numbers. You just have to keep in mind that $1 + 1 = 2_{10} = 10_2$, so if you add two 1's together, there is a carry.

There are a few examples of binary addition in Figure 2.3. These correspond to the decimal operations $10 + 4 = 14$, $11 + 2 = 13$, and $13 + 5 = 18$. You can use the familiar algorithms you know for subtraction, multiplication, and division as well.

Positive and Negative Integers

The method described above will let us represent any *positive* integer in the computer's memory. What about negative numbers?

The bits that make up the computer's memory must be used to represent both positive and negative numbers. The typical method is called *two's*

binary	**decimal**	**binary**	**decimal**
1111	−1	0111	7
1110	−2	0110	6
1101	−3	0101	5
1100	−4	0100	4
1011	−5	0011	3
1010	−6	0010	2
1001	−7	0001	1
1000	−8	0000	0

Figure 2.4: The four-bit two's complement values

complement notation. (The previous method, which can't represent negative values, is generally called *unsigned integer* representation.)

To convert a positive value to a negative value in two's complement, you first flip all of the bits (convert 0s to 1s and 1s to 0s) and then add one. So, the four-bit two's complement representation for −5 is:

start with the positive version:	0101
flip all of the bits:	1010
add one:	1011 .

All of the four-bit two's complement values are shown in Figure 2.4. If we use four bits, we can represent values from −8 to 7.

Here are a few other reasons computers use two's complement notation:

- It's easy to tell if the value is negative: if the first bit is 1, it's negative.
- For positive numbers (values with the first bit 0), the unsigned and two's complement representations are identical. The values 0–7 have the same representations in Figures 2.2 and 2.4.
- Addition and subtraction work the same way as for unsigned numbers. Look back at Figure 2.3. If you instead interpret at the numbers as two's complement values, the corresponding decimal calculations are $-6 + 4 = -2$, $-5 + 2 = -3$, and $-3 + 5 = 2$. (You have to ignore the last 1 that was carried in the last example—the computer will.) They are still correct. That means that the parts of the computer that do calculations don't have to know whether they have unsigned or two's complement values to work with.

- No number has more than one two's complement representation. If instead the first bit was used for the sign (0 = positive, 1 = negative), then there would be two versions of zero: 0000 and 1000. This is a waste of one representation, which wastes storage space, not to mention that the computer has to deal with the special case that 0000 and 1000 are actually the same value. That makes it difficult to compare two values.

Most modern computers and programming languages use 32 bits to store integers. With this many bits, it is possible to store integers from -2^{31} to $2^{31} - 1$ or -2147483648 to 2147483647.

So, in many programming languages, you will get an error if you try to add one to 2147483647. In other languages, you will get -2147483648. The analogous calculation with four bits is $7 + 1$:

$$\begin{array}{r} {}^{1\,1\,1} \\ 0\,1\,1\,1 \\ +\,0\,0\,0\,1 \\ \hline 1\,0\,0\,0 \end{array}$$

If these were unsigned values, this is the right answer. But, if you look in Figure 2.4, you'll see that 1000 represents -8. If this *overflow* isn't caught when doing two's complement, there's a "wraparound" that means you can suddenly go from a large positive number to a large negative one, or vice-versa.

In Python, you don't generally see any of this. Python will automatically adjust how it represents the numbers internally and can represent any integer. But, if you go on to other languages, you will eventually run into an integer overflow.

Another type of numbers is the floating point value. They have to be stored differently because there's no way to store fractional parts with two's complement. Floating point representation is more complicated; it is beyond the scope of this course.

Characters and Strings

The other types of information that you have seen in your Python experience are characters and strings. A *character* is a single letter, digit or punctuation symbol. A *string* is a collection of several characters. So, some characters are `T`, `$`, and `4`. Some strings are `"Jasper"`, `"742"`, and `"bhay-gn-flay-vn"`.

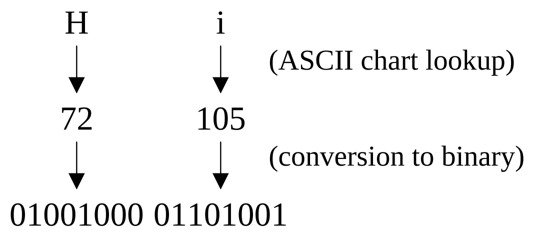

Figure 2.5: Conversion of the string “Hi” to binary.

Storing characters is as easy as storing unsigned integers. For a byte (8 bits) in the computer’s memory, there are $2^8 = 256$ different unsigned numbers: 0–255. So, just assign each possible character a number and translate the numbers to characters.

For example, the character `T` is represented by the number 84, the character `$` by 36, and `4` by 52. This set of translations from numbers to characters and back again is called a *character set.* The particular character set that is used by almost all modern computers, when dealing with English and other western languages, is called *ASCII.* The course web site contains links to a full list of ASCII characters, if you’d like to see it.

So, in order to store the character `T` in the computer’s memory, first look up its number in the character set and get 84. Then, use the method described for unsigned integers to convert the number 84 to an 8-bit value: 01010100. This can then be stored in the computer’s memory.

With only one byte per character, we can only store 256 different characters in our strings. This is enough to represent English text, but it starts to get pretty hard to represent languages with accents (like á or ü). It’s just not enough characters to represent languages like Chinese or Japanese.

The *Unicode* character set was created to overcome this limitation. Unicode can represent up to 2^{32} characters. This is enough to represent all of the written languages that people use. Because of the number of possible characters, Unicode requires more than one byte to store each character.

In ASCII, storing strings with several characters, can be done by using a sequence of several bytes and storing one character in each one. For example, in Figure 2.5, the string “Hi” is converted to binary.

In Figure 2.5, the binary string 0100100001101001 represents “Hi” in ASCII. But, if you look at this chunk of binary as representing an integer, it's the same as 18537. How does the computer know whether these two bytes in memory are representing the string "Hi" or the number 18537?

There actually isn't any difference as far as the computer itself is concerned. Its only job is to store the bits its given and do whatever calculations it's asked to do. The programming language must keep track of what kind of information the different parts of the memory are holding. This is why the concept of types is so important in Python. If Python didn't keep track of the type of each variable, there would be no way to tell.

In some programming languages, C in particular, you can work around the type information that the programming language is storing. For example, you could store the string "Hi" and then later convince the computer that you wanted to treat that piece of memory like a number and get 18537. This is almost always a bad idea.

How computers represent various types of information is sometimes quite important when programming. It is also discussed in CMPT 150 (Computer Design) and courses that cover how programming languages work like CMPT 379 (Compiler Design).

Topic 2.7 Example Problem Solving: Feet and Inches

Back in Topic 2.5, there was a program that converted someone's height in metres to inches:

```
Enter your height (in metres): 1.6
You are 62.992 inches tall.
```

But, people don't usually think of their height in terms of the number of inches. It's much more common to think of feet and inches. It would be better if the program worked like this:

```
Enter your height (in metres): 1.6
You are 5' 3" tall.
```

```
**write** "Enter your height (in metres):"
**read** $metres$
**set** $totalinches$ to $39.37 \times metres$
**set** $feet$ to $\lfloor totalinches/12 \rfloor$
**set** $inches$ to $totalinches - feet \times 12$
round $inches$ to the nearest integer
**write** "You are $feet'$ $inches''$ tall."
```

Figure 2.6: Meters to feet-inches conversion pseudocode.

```
metres = float(raw_input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = total_inches/12
print "You are " + str(feet) + " feet tall."
```

Figure 2.7: Converting to feet and inches: number of feet.

The notation $5'\ 3''$ is used to indicate "5 feet and 3 inches", which is $5 \times 12 + 3 = 63$ inches.

To do this conversion, convert the number of metres to inches, as done in Topic 2.5, by multiplying by 39.37. Then, determine how many feet and inches there are in the total number of inches. The pseudocode is shown in Figure 2.6.

When you're converting an idea for an algorithm to code, you shouldn't try to do it all at once, especially when you're first learning to program. Implement part of the algorithm first, then test the program to make sure it does what you expect before you move on. Trying to find problems in a large chunk of code is very hard: start small.

Start writing a Python program to implement the pseudocode in Figure 2.6. You can grab the first few lines from the program in Topic 2.5. Then, try to calculate the number of feet. This has been done in Figure 2.7.

Note that when you run this program, it calculates the number of feet as a floating point number:

```
Enter your height (in metres): 1.6
You are 5.24933333333 feet tall.
```

```
metres = float(raw_input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = int(total_inches/12)
inches = total_inches - feet*12
print "You are " + str(feet) + " feet and " \
        + str(inches) + " inches tall."
```

Figure 2.8: Converting to feet and inches: feet and inches.

This makes sense, given what we know about types: when Python divides a floating point value (`metres`), it returns a floating point value. But in the algorithm, you need an integer and it needs to be rounded *down* to the next integer. This is what the `int` function does when it converts floating point numbers to integers, so you can use that to get the correct value in `feet`.

If you have a statement in Python that you want to split across multiple lines, so it's easier to read, you can end the line with a backslash, "\". This was done in Figures 2.7 and 2.8, so the code would fit on the page.

Once you have the correct number of feet as an integer, you can calculate the number of inches too. This is done in Figure 2.8.

This program does the right calculation, but leaves the number of inches as a floating point number:

```
Enter your height (in metres): 1.6
You are 5 feet and 2.992 inches tall.
```

To convert the number of inches to an integer, you can't use the `int` function, which would always round down. You shouldn't get 5′ 2″ in the above example; you should round to the *nearest* integer and get 5′ 3″.

You can use the `round` function for this. Note that `round` does the rounding, but leaves the result as a floating point value. You will have to use the `int` function to change the type, but the value will already be correct.

See Figure 2.9 for the details. When you run this program, the output is almost correct:

```
Enter your height (in metres): 1.6
You are 5 feet and 3 inches tall.
```

```
metres = float(raw_input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = int(total_inches/12)
inches = int(round(total_inches - feet*12))
print "You are " + str(feet) + " feet and " \
        + str(inches) + " inches tall."
```

Figure 2.9: Converting to feet and inches: rounding inches.

The last thing you have to do to get the program working exactly as specified at the start of the topic is to print out the feet and inches in the proper format: 5′ 3″. This presents one last problem. You can't just print double quotes, since they are used to indicate where the string literal begins and ends. Code like this will generate an error:

```
print str(feet) + "' " + str(inches) + "" tall"
```

The interpreter will see the `""` and think it's an empty string (a string with no characters in it). Then, it will be very confused by the word "`tall`". The solution is to somehow indicate that the quote is something that it should print, not something that's ending the string. There are several ways to do this in Python:

- Put a backslash before the quote. This is called *escaping a character*. It's used in a lot of languages to indicate that you mean the character itself, not its special use.

  ```
  print str(feet) + "' " + str(inches) + "\" tall"
  ```

- Use a single quote to wrap up the string. In Python, you can use either single quotes (`'`) or double quotes (`"`) to indicate a string literal. There's no confusion if you have a double quote inside a single-quoted string.

  ```
  print str(feet) + "' " + str(inches) + '" tall'
  ```

  Of course, you have to use double quotes for the string that contains a single quote.

```
metres = float(raw_input( \
        "Enter your height (in metres): "))
total_inches = 39.37 * metres
feet = int(total_inches/12)
inches = int(round(total_inches - feet*12))
print "You are " + str(feet) + "' " \
        + str(inches) + '" tall.'
```

Figure 2.10: Converting to feet and inches: printing quotes

- A final trick that can be used is Python's *triple-quoted string*. If you wrap a string in *three* sets of double quotes, you can put anything inside (even line breaks). This can be a handy trick if you have a lot of stuff to print and don't want to have to worry about escaping characters.

  ```
  print str(feet) + """' """ + str(inches) \
          + """" tall"""
  ```

  This can be very cumbersome and hard to read for short strings like this. (As you can see, it made the whole thing long enough it wouldn't fit on one line.) It's more useful for long strings.

So, finally, the quotes can be printed to produce the desired output. See Figure 2.10. When the program runs, it produces output like this:

```
Enter your height (in metres): 1.6
You are 5' 3" tall.
```

But, there is still one problem with this program that is a little hard to notice. What happens when somebody comes along who is 182 cm tall?

```
Enter your height (in metres): 1.82
You are 5' 12" tall.
```

That's not right: five feet and twelve inches should be displayed as six feet and zero inches. The problem is with the rounding-off in the calculation. For this input, `total_inches` becomes 71.6534, which is *just under* six feet (72 inches). Then the division to calculate `feet` gives a result of 5, which we should think of as an error.

The problem isn't hard to fix: we are just doing the rounding-off too late. If instead of `total_inches` being the floating-point value 71.6534, we could

```
metres = float(raw_input( \
        "Enter your height (in metres): "))
total_inches = int(round(39.37 * metres))
feet = total_inches/12
inches = total_inches - feet*12
print "You are " + str(feet) + "' " \
        + str(inches) + '" tall.'
```

Figure 2.11: Converting to feet and inches: fixed rounding error

round it off immediately to 72. That would correct this problem and it has been done in Figure 2.11.

Now we get the right output:

```
Enter your height (in metres): 1.82
You are 6' 0" tall.
```

This is a good lesson for you to see at this point: it's important to test your program carefully, since bugs can hide in unexpected places.

Check-Up Questions

- ▶ Download the code from this topic from the course web site and test it with some other inputs. Do the conversion by hand and make sure the program is working correctly.
- ▶ Try some "bad" inputs and see what the program does. For example, what if the user types in a negative height? What if they type something that isn't a number?

Summary

There's a lot in this unit. You should be writing your first programs and figuring out how computers work. The example developed in Topic 2.7 is intended to give you some idea of how the process of creating a program might look.

When you're learning to program, you should be writing programs. Reading this Guide over and over won't help. You should actually spend some time at a computer, experimenting with the ideas presented here, learning how to decipher error messages, and dealing with all of the other problems that come with writing your first programs.

Key Terms

- interactive interpreter
- statement
- expression
- operator
- function
- argument
- variable
- variable assignment
- type
- conversion
- integer
- unsigned integer
- string
- ASCII
- floating point
- binary
- bit
- byte
- two's complement
- character set
- escaping a character