Introduction 1

The intent of this text is to introduce you to computer programming using the Python programming language. Learning to program is a bit like learning to play piano, although quite a bit easier since we won't have to program while keeping time according to a time signature. Programming is a creative process so we'll be working on developing some creative skills. At the same time, there are certain patterns that can be used over and over again in this creative process. The goal of this text and the course you are taking is to get you familiar with these patterns and show you how they can be used in programs. After working through this text and studying and practicing you will be able to identify which of these patterns are needed to implement a program for a particular task and you will be able to apply these patterns to solve new and interesting problems.

As human beings our intelligent behavior hinges on our ability to match patterns. We are pattern-matchers from the moment we are born. We watch and listen to our parents and siblings to learn how to react to situations. Babies watch us to learn to talk, walk, eat, and even to smile. All these behaviors are learned through pattern matching. Computer Science is no different. Many of the programs we create in Computer Science are based on just a few patterns that we learn early in our education as programmers. Once we've learned the patterns we become effective programmers by learning to apply the patterns to new situations. As babies we are wired to learn quickly with a little practice. As we grow older we can learn to use patterns that are more abstract. That is what Computer Science is all about: the application of abstract patterns to solve new and interesting problems.

PRACTICE is important. There is a huge difference between reading something in this text or understanding what is said during a lecture and being able to do it yourself. At times this may be frustrating, but with practice you will get better at it. As you read the text make sure you take time to do the practice exercises. Practice exercises are clearly labeled with a gray background color. These exercises are your chance to use a concept that you have just learned. Answers to practice exercises are included at the end of each chapter so you can check your answers.

K.D. Lee, *Python Programming Fundamentals,* Undergraduate Topics in Computer Science, 1
DOI 10.1007/978-1-84996-537-8_1, © Springer-Verlag London Limited 2011

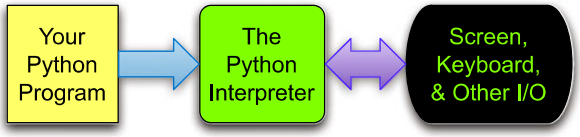


Fig. 1.1 The Python interpreter

1.1 The Python Programming Language

Python is the programming language this text uses to introduce computer programming. To run a Python program you need an interpreter. The Python interpreter is a program that reads a Python program and then executes the statements found in it as depicted in Fig. 1.1. While studying this text you will write many Python programs. Once your program is written and you are ready to try it you will tell the Python interpreter to execute your Python program so you can see what it does.

For this process to work you must first have Python installed on your computer. Python is free and available for download from the internet. The next section of this chapter will take you through downloading and installing Python. Within the last few years there were some changes to the Python programming language between Python 2 and Python 3. The text will describe differences between the two versions of Python as they come up. In terms of learning to program, the differences between the two versions of Python are pretty minor.

To write Python programs you need an editor to type in the program. It is convenient to have an editor that is designed for writing Python programs. An editor that is specifically designed for writing programs is called an IDE or Integrated Development Environment. An IDE is more than just an editor. It provides highlighting and indentation that can help as you write a program. It also provides a way to run your program straight from the editor. Since you will typically run your program many times as you write it, having a way to run it quickly is handy. This text uses the Wing IDE 101 in many of its examples. This IDE is simple to install and is free for educational use. Wing IDE 101 is available for Mac OS X, Microsoft Windows, and Linux.

When learning to program and even as a seasoned professional, it can be advantageous to run your program using a tool called a debugger. A debugger allows you to run your program, stop it at any point, and inspect the state of the program to help you better understand what is happening as your program executes. The Wing IDE includes an integrated debugger for that purpose. There are certainly other IDEs that might be used and nothing presented in this text precludes you from using something else. Some examples of IDEs for Python development include Netbeans, Eclipse, Eric, and IDLE. Eric's debugger is really quite nice and could serve as an alternative to Wing should Wing IDE 101 not be an option for some reason.

Fig. 1.2 Installing Python on Windows

1.2 Installing Python and Wing IDE 101

To begin writing Python programs on your own computer, you need to have Python installed. There were some significant changes between Python 2.6 and Python 3 which included a few changes that make programs written for version 3 incompatible with programs written for version 2.6 and vice versa. If you are using this book as part of an introductory course, your instructor may prefer you install one version or the other. Example programs in this text are written using Python 3 syntax but the differences between Python 2 and 3 are few enough that it is possible to use either Python 2 or 3 when writing programs for the exercises in this text. Inset boxes titled Python 2 ⇝ 3 will highlight the differences when they are first encountered in the text.

If you are running Windows you will likely have to install Python yourself. You can get the installation package from http://python.org. Click the *DOWNLOAD* link on the page. Then pick the appropriate installer package. Most will want to download the *Python 3.1 (or newer) Windows x86 MSI Installer* package. Once you have downloaded it, double-click the package and take all the defaults to install it as pictured in Fig. 1.2.

If you have a Mac, then Python is already installed and may be the version you want to use, depending on how new your Mac is. You can find out which version of Python you have by opening a terminal window. Go to the Applications folder and look in the Utilities sub-folder for the Terminal application. Start a terminal and in the window type *python*.

1

You should see something like this:

```
Kent's Mac> python
Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more info.
>>>
```

You can press and hold the control key (i.e. the ctrl key) and press 'd' to exit Python or just close the terminal window. If you do not have version 3.1 or newer installed on your Mac you may wish to download the *Python 3.1 (or newer) MacOS Installer Disk Image* from the http://python.org web site. Once the file is downloaded you can double-click the disk image file and then look for the *Python.mpkg* file and double-click it as pictured in Fig. 1.3. You will need an administrator password to install it which in most cases is just your own password.

If you have a Mac you will also need the X11 Server installed. The X11 Server, called the XQuartz project, is needed to run Wing IDE 101 on a Mac. Directions for installing the X11 Server can be found at http://wingware.com/doc/howtos/osx. The server is provided by Apple and the XQuartz developers and is freely distributed. The X11 server has shipped with Mac OS X since version 10.5.

While you don't need an IDE like Wing to write and run Python programs, the debugger support that an IDE like Wing provides will help you understand how Python programs work. It is also convenient to write your programs in an IDE so you can run them quickly and easily. To install Wing IDE 101 you need to go to the http://wingware.com web site.

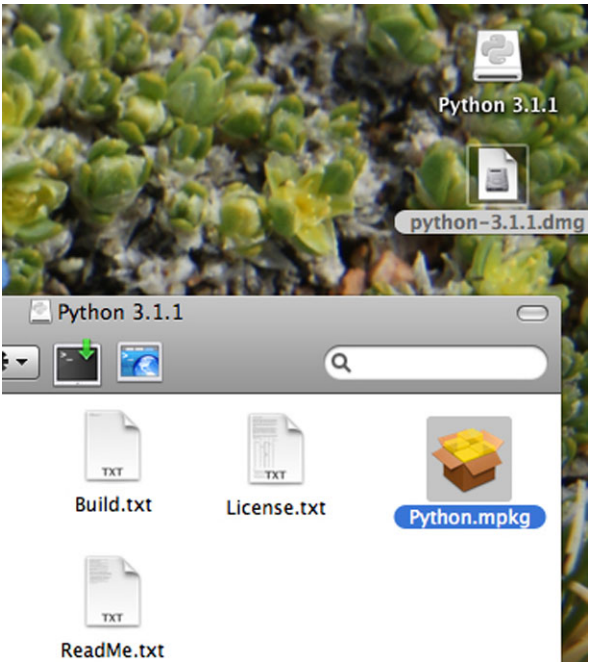


Fig. 1.3 Installing Python on Mac OS X

Fig. 1.4 Installing Wing IDE 101 on Windows

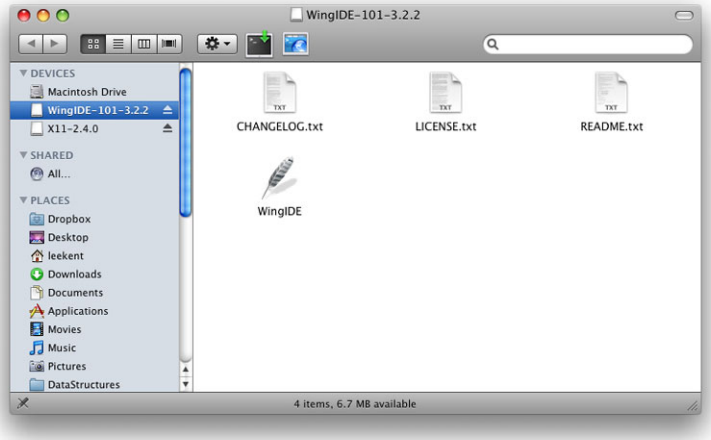


Fig. 1.5 Installing Wing IDE 101 on a Mac

Find the *Download* link at the top of the web page and select *Wing IDE 101* to download the installation package. If you are installing on a Mac, pick the Mac version. If you are installing on Windows, pick the Windows version. Download and run the installation package if you are using Windows. Running the Windows installer should display an installer window like that pictured in Fig. 1.4. Take all the defaults to install it.

1

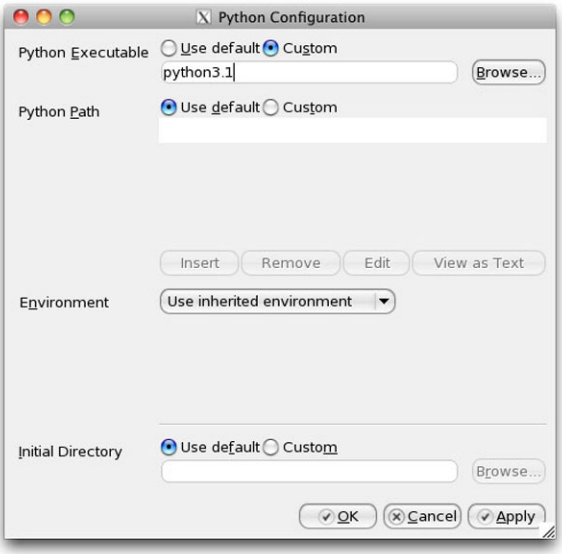


Fig. 1.6 Configuring Wing's Python interpreter

If you are installing Wing IDE 101 on a Mac then you need to mount the disk image. To do this you must double-click a file that looks like *wingide-101-3.2.2-1-i386.dmg*. After double-clicking that file you will have a mounted disk image of the same name, minus the .dmg extension). If you open a Finder window for that disk image you will see a window that looks like Fig. 1.5. Drag the Wing IDE icon to your Applications folder and you can add it to your dock if you like.

Configuring Wing

If you look at Fig. 1.8 you will see that the Python interpreter shows up as *Python 3.1.1*. When you install Wing, you should open it and take a look at your *Python Shell* tab. If you see the wrong version of Python then you need to configure Wing to use the correct Python Shell. To do this you must open Wing and go to the Edit menu. Under the Edit menu, select *Configure Python...* and type in the appropriate interpreter. If you are using a Mac and wish to use version 3.1 then you would type *python3.1*. Figure 1.6 shows you what this dialog box looks like and what you would type in on a Mac. In Windows, you should click the browse button and find python.exe. This will be in a directory like *C:\Python31* if you chose the defaults when installing.

There is one more configuration change that should be made. The logical flow of a Python program depends on the program's indentation. Since indentation is so important, Wing can provide a visual cue to the indentation in your program called an *indent guide*. These indent guides will not show up in this chapter, but they will in subsequent chapters. Go to the Edit menu again and select *Preferences*. Then click on the *Indentation* selection in the dialog box as shown in Fig. 1.7. Select the checkbox that says *Show Indent Guides*.

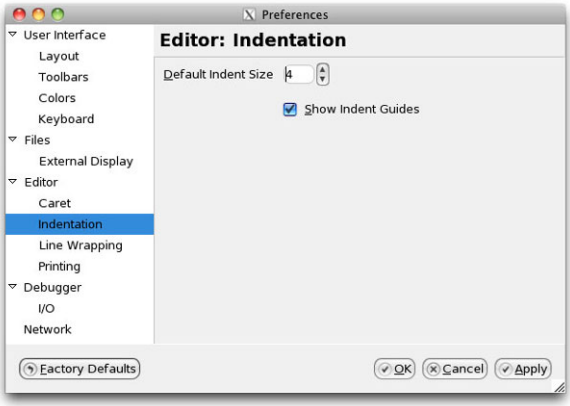


Fig. 1.7 Configuring indent guides

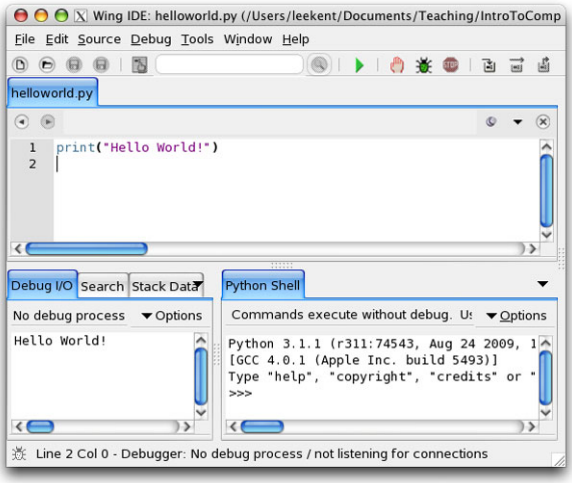


Fig. 1.8 The Wing IDE

That's it! Whether you are a Mac or Windows user if you've followed the directions in this section you should have Python and Wing IDE 101 installed and ready to use. The next section shows you how to write your first program so you can test your installation of Wing IDE 101 and Python.

1.3 Writing Your First Program

To try out the installation of your IDE and Python you should write a program and run it. The traditional first program is the *Hello World* program. This program simply prints "*Hello World*!" to the screen when it is run. This can be done with one statement in Python.

1

Open your IDE if you have not already done so. If you are using Windows you can select it by going to the Start menu in the bottom left hand corner and selecting *All Programs*. Look for *Wing IDE 101* under the Start menu and select it. If you are using a Mac, go to the Applications folder and double-click the Wing IDE icon or click on it in your dock if you installed the icon on your dock. Once you've done this you will have a window that looks like Fig. 1.8.

In the IDE window you go to the *File* menu and select *New* to get a new edit tab within the IDE. You then enter one statement, the print statement shown in Fig. 1.8 to print *Hello World!* to the screen. After entering the one line program you can run it by clicking the green debug button (i.e. that button that looks like a bug) at the top of the window. You will be prompted to save the file. Click the *Save Selected Files* button and save it as *helloworld.py*. You should then see *Hello World!* printed at the bottom of the IDE window in the *Debug I/O* tab.

The print statement that you see in this program prints the string "Hello World!" to standard output. Text printed to standard output appears in the *Debug I/O* tab in the Wing IDE. That should do it. If it doesn't you'll need to re-read the installation instructions either here or on the websites you downloaded Python and Wing IDE from or you can find someone to help you install them properly. An IDE is used in examples and practice exercises throughout this text so you'll need a working installation of an IDE and Python to make full use of this text.

Python 2 ⇝ 3

Prior to Python version 3 print statements were different than many other statements in Python because they lacked parentheses[8]. Parentheses were added to print statement in Python 3. So,

```
print "Hello World!"
```

became

```
print("Hello World!")
```

in Python 3 and later. A print statement prints its data and then moves to a new line unless the newline character is suppressed. Before Python 3 the newline was suppressed by adding a comma to the end of the print statement.

```
print "Hello",
print " World!"
```

In Python 3 the same can be done by specifying an empty line end.

```
print("Hello",end="")
print(" World!")
```

1.4 What is a Computer?

So you've written your first program and you've been *using* a computer all your life. But, what is a computer, really? A computer is composed of a Central Processing Unit (abbreviated CPU), memory, and Input/Output (abbreviated I/O) devices. A screen is an output device. A mouse is an input device. A hard drive is an I/O device.

The CPU is the brain of the computer. It is able to store values in memory, retrieve values from memory, add/subtract two numbers, compare two numbers and do one of two

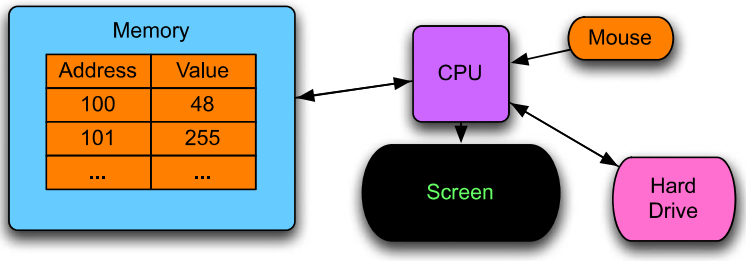


Fig. 1.9 Conceptual view of a computer

things depending on the outcome of that comparison. The CPU can also control which instruction it will execute next. Normally there are a list of instructions, one after another, that the CPU executes. Sometimes the CPU may jump to a different location within that list of instructions depending on the outcome of some comparison.

That's it. A CPU can't do much more than what was described in the previous paragraph. CPU's aren't intelligent by any leap of the imagination. In fact, given such limited power, it's amazing how much we are able to do with a computer. Everything we use a computer for is built on the work of many, many people who have built layers and layers of programs that make our life easier.

The memory of a computer is a place where values can be stored and retrieved. It is a relatively fast storage device, but it loses its contents as soon as the computer is turned off. It is called volatile store. The memory of a computer is divided into different locations. Each location within memory has an address and can hold a value. Figure 1.9 shows the contents of memory location 100 containing the number 48.

The hard drive is non-volatile storage or sometimes called persistent storage. Values can be stored and retrieved from the hard drive, but it is relatively slow compared to the memory and CPU. However, it retains its contents even when the power is off.

In a computer, everything is stored as a sequence of 0's and 1's. For instance, the string 01010011 can be interpreted as the decimal number 83. It can also represent the capital letter 'S'. How we interpret these strings of 0's and 1's is up to us. We can tell the CPU how to interpret a location in memory by which instruction we tell the CPU to execute. Some instructions treat 01010011 as the number 83. Other instructions treat it as the letter 'S'.

One digit in a binary number is called a *bit*. Eight bits grouped together are called a *byte*. Four bytes grouped together are called a *word*. 2^{10} bytes are called a *kilobyte* (i.e. KB). 2^{10} kilobytes are called a *megabyte* (i.e. MB). 2^{10} megabytes are called a *gigabyte* (i.e. GB). 2^{10} gigabytes are called a *terabyte* (i.e. TB). Currently memories on computers are usually in the one to eight GB range. Hard Drives on computers are usually in the 500 GB to two TB range.

Binary	Dec	Char	Binary	Dec	Char	Binary	Dec	Char
0100000	32	␣	1000000	64	@	1100000	96	`
0100001	33	!	1000001	65	A	1100001	97	a
0100010	34	"	1000010	66	B	1100010	98	b
0100011	35	#	1000011	67	C	1100011	99	c
0100100	36	$	1000100	68	D	1100100	100	d
0100101	37	%	1000101	69	E	1100101	101	e
0100110	38	&	1000110	70	F	1100110	102	f
0100111	39	'	1000111	71	G	1100111	103	g
0101000	40	(	1001000	72	H	1101000	104	h
0101001	41	)	1001001	73	I	1101001	105	i
0101010	42	*	1001010	74	J	1101010	106	j
0101011	43	+	1001011	75	K	1101011	107	k
0101100	44	,	1001100	76	L	1101100	108	l
0101101	45	-	1001101	77	M	1101101	109	m
0101110	46	.	1001110	78	N	1101110	110	n
0101111	47	/	1001111	79	O	1101111	111	o
0110000	48	0	1010000	80	P	1110000	112	p
0110001	49	1	1010001	81	Q	1110001	113	q
0110010	50	2	1010010	82	R	1110010	114	r
0110011	51	3	1010011	83	S	1110011	115	s
0110100	52	4	1010100	84	T	1110100	116	t
0110101	53	5	1010101	85	U	1110101	117	u
0110110	54	6	1010110	86	V	1110110	118	v
0110111	55	7	1010111	87	W	1110111	119	w
0111000	56	8	1011000	88	X	1111000	120	x
0111001	57	9	1011001	89	Y	1111001	121	y
0111010	58	:	1011010	90	Z	1111010	122	z
0111011	59	;	1011011	91	[	1111011	123	{
0111100	60	<	1011100	92	\	1111100	124	|
0111101	61	=	1011101	93	]	1111101	125	}
0111110	62	>	1011110	94	^	1111110	126	~
0111111	63	?	1011111	95	_	1111111	127	DEL

Fig. 1.10 The ASCII table

1.5 Binary Number Representation

Each digit in a decimal number represents a power of 10. The right-most digit is the number of ones, the next digit is the number of 10's, and so on. To interpret integers as binary numbers we use powers of 2 just as we use powers of 10 when interpreting integers as

decimal numbers. The right-most digit of a binary number represents the number of times $2^0 = 1$ is needed in the representation of the integer. Our choices are only 0 or 1 (i.e. we can use one 2^0 if the number is odd), because 0 and 1 are the only choices for digits in a binary number. The next right-most is $2^1 = 2$ and so on. So 01010011 is $0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 83$. Any binary number can be converted to its decimal representation by following the steps given above. Any decimal number can be converted to its binary representation by subtracting the largest power of two that is less than the number, marking that digit as a 1 in the binary number and then repeating the process with the remainder after subtracting that power of two from the number.

Practice 1.1 What is the decimal equivalent of the binary number 01010101_2?

Example 1.1 There is an elegant algorithm for converting a decimal number to a binary number. You need to carry out long division by 2 to use this algorithm. If we want to convert 83_{10} to binary then we can repeatedly perform long division by 2 on the quotient of each result until the quotient is zero. Then, the string of the remainders that were accumulated while dividing make up the binary number. For example,

$83/2 = 41$ *remainder* 1
$41/2 = 20$ *remainder* 1
$20/2 = 10$ *remainder* 0
$10/2 = 5$ *remainder* 0
$5/2 = 2$ *remainder* 1
$2/2 = 1$ *remainder* 0
$1/2 = 0$ *remainder* 1

The remainders from last to first are 1010011_2 which is 83_{10}. This set of steps is called an algorithm. An algorithm is like a recipe for doing a computation. We can use this algorithm any time we want to convert a number from decimal to binary.

Practice 1.2 Use the conversion algorithm to find the binary representation of 58_{10}.

To add two numbers in binary we perform addition just the way we would in base 10 format. So, for instance, $0011_2 + 0101_2 = 1000_2$. In decimal format this is $3 + 5 = 8$. In binary format, any time we add two 1's, the result is 0 and 1 is carried.

To represent negative numbers in a computer we would like to pick a format so that when a binary number and its opposite are added together we get zero as the result. For this to work we must have a specific number of bits that we are willing to work with.

1

Typically thirty-two or sixty-four bit addition is used. To keep things simple we'll do some eight bit addition in this text. Consider $00000011_2 = 3_{10}$.

It turns out that the 2's complement of a number is the negative of that number in binary. For example, the numbers $3_{10} = 00000011_2$ and $-3_{10} = 11111101_2$. 11111101_2 is the 2's complement of 00000011. It can be found by reversing all the 1's and 0's (which is called the 1's complement) and then adding 1 to the result.

Example 1.2 Adding 00000011 and 11111101 together gives us

$$\begin{array}{r} 00000011 \\ +11111101 \\ = 100000000 \end{array}$$

This only works if we limit ourselves to 8 bit addition. The carried 1 is in the ninth digit and is thrown away. The result is 0.

Practice 1.3 If $01010011_2 = 83_{10}$, then what does -83_{10} look like in binary? HINT: Take the 2's complement of 83 or figure out what to add to 01010011_2 to get 0.

If binary $11111101_2 = -3_{10}$ does that mean that 253 can't be represented? The answer is yes and no. It turns out that 11111101_2 can represent -3_{10} or it can represent 253_{10} depending on whether we want to represent both negative and positive values or just positive values. The CPU instructions we choose to operate on these values determine what types of values they are. We can choose to use signed integers in our programs or unsigned integers. The type of value is determined by us when we write the program.

Typically, 4 bytes, or one word, are used to represent an integer. This means that 2^{32} different signed integers can be represented from -2^{31} to $2^{31} - 1$. In fact, Python can handle more integers than this but it switches to a different representation to handle integers outside this range. If we chose to use unsigned integers we could represent numbers from 0 to $2^{32} - 1$ using one word of memory.

Not only can 01010011_2 represent 83_{10}, it can also represent a character in the alphabet. If 01010011_2 is to be interpreted as a character almost all computers use a convention called ASCII which stands for the American Standard Code for Information Interchange [12]. This standard equates numbers from 0 to 127 to characters. In fact, numbers from 128 to 255 also define extended ASCII codes which are used for some character graphics. Each ASCII character is contained in one byte. Figure 1.10 shows the characters and their equivalent integer representations.

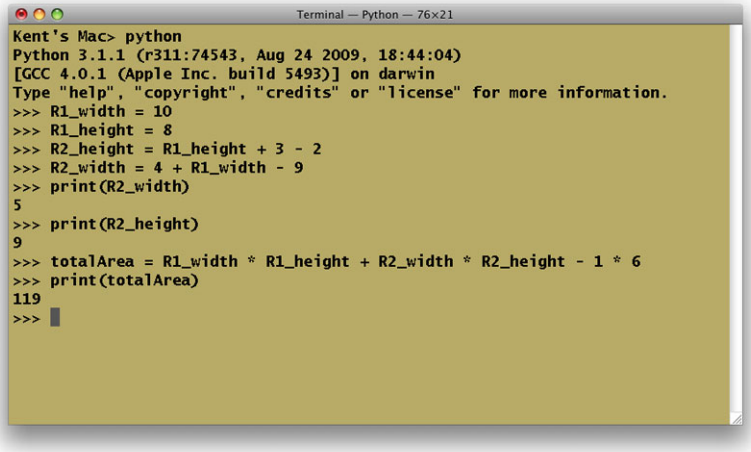


Fig. 1.11 The Python shell

Practice 1.4 What is the binary and decimal equivalent of the space character?

Practice 1.5 What determines how the bytes in memory are interpreted? In other words, what makes 4 bytes an integer as opposed to four ASCII characters?

1.6 What is a Programming Language?

If we were to have to write programs as sequences of numbers we wouldn't get very far. It would be so tedious to program that no one would want to be a programmer. In the spring of 2006 Money Magazine ranked Software Engineer [4] as the number one job in America in terms of overall satisfaction which included things like compensation, growth, and stress-levels. So it must not be all that tedious.

A programming language is really a set of tools that allow us to program at a much higher level than the 0's and 1's that exist at the lowest levels of the computer. Python and the Wing IDE provides us with a couple of tools. The lower right corner of the Wing

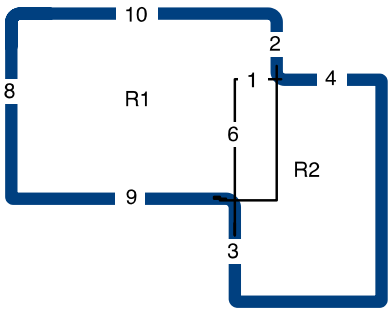


Fig. 1.12 Overlapping rectangles

IDE has a tab labeled *Python Shell*. The shell allows programmers to interact with the Python interpreter. The interpreter is a program that interprets the programs we write. If you have a Mac or Linux computer you can also start the Python interpreter by opening up a terminal window. If you use Windows you can start a Command Prompt by looking under the Accessories program group. Typing *python* at a command prompt starts a Python interpreter as shown in Fig. 1.11.

Consider computing the area of a shape constructed of overlapping regular polygons. In Fig. 1.12 all angles are right angles and all distances are in meters. Our job is to figure out the area in square meters. The lighter lines in the middle help us figure out how to compute the area. We can compute the area of the two rectangles and then subtract one of the overlapping parts since otherwise the overlapping part would be counted twice.

This can be computed on your calculator of course. The Python Shell is like a calculator and Fig. 1.11 shows how it can be used to compute the area of the shape. The first line sets a variable called *R1_width* to the value of 10. Then *R1_height* is set to 8. We can store a value in memory and give it a name. This is called an *assignment statement*. Your calculator can store values. So can Python. In Python these values can be given names that mean something in our program. *R1_height* is the name we gave to the height of the R1 rectangle. Anytime we want to retrieve that value we can just write *R1_height* and Python will retrieve its value for us.

Practice 1.6 Open up the Wing IDE or a command prompt and try out the assignment and print statements shown in Fig. 1.11. Make sure to type the statements into the python shell. You DO NOT type the >>>. That is the Python shell prompt and is printed by Python. Notice that you can't fix a line once you have pressed enter. This will be remedied soon.

Practice 1.7 Take a moment and answer these questions from the material you just read.

1. What is an assignment statement?
2. How do we retrieve a value from memory?
3. Can we retrieve a value before it has been stored? What happens when we try to do that?

Interacting directly with the Python shell is a good way to quickly see how something works. However, it is also painful because mistakes can't be undone. In the next section we'll go back to writing programs in an editor so they can be changed and run as many times as we like. In fact, this is how most Python programming is done. Write a little, then test it by running it. Then write a little more and run it again. This is called prototyping and is an effective way to write programs. You should write all your programs using prototyping while reading this text. Write a little, then try it. That's an effective way to program and takes less time than writing a lot and then trying to figure out what went wrong.

1.7 Hexadecimal and Octal Representation

Most programmers do not have to work with binary number representations. Programming languages let programmers write numbers in base 10 and they do the conversion for us. However, once in a while a programmer must be concerned about the binary representation of a number. As we've seen, converting between binary and decimal isn't hard, but it is somewhat tedious. The difficulty arises because 10 is not a power of 2. Converting between base 10 and base 2 would be a lot easier if 10 were a power of 2. When computer programmers have to work with binary numbers they don't want to have to write out all the zeroes and ones. This would obviously be tedious as well. Instead of converting numbers to base 10 or writing all numbers in binary, computer programmers have adopted two other representations for binary numbers, base 16 (called hexadecimal) and base 8 (called octal).

In hexadecimal each digit of a number can represent sixteen different binary numbers. The sixteen hexadecimal digits are 0–9, and A–F. Since 16 is a power of 2, there are exactly 4 binary digits that make up each hexadecimal digit. So, 0000_2 is 0_{16} and 1111_2 is F_{16}. So, the binary number 10101110 is AE in hexadecimal notation and 256 in octal notation. If we wish to convert either of these two numbers to binary format the conversion is just as easy. 1010_2 is A_{16} for instance. Again, these conversions can be done quickly because there are four binary digits in each hexadecimal digit and three binary digits in each octal digit.

1

Example 1.3 To convert the binary number 01010011_2 to hexadecimal we have only to break the number into two four digit binary numbers 0101_2 and 0011_2. $0101_2 = 5_{16}$ and $0011_2 = 3_{16}$. So the hexadecimal representation of 01010011_2 is 53_{16}.

Python has built-in support of hexadecimal numbers. If you want to express a number in hexadecimal form you preface it with a $0x$ to signify that it is a hexadecimal number. For instance, here is how Python responds to $0x53$ being entered into the Python shell.

```
Kent's Mac> python
Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more info.
>>> 0x53
83
>>> 0o123
83
>>>
```

Since $8 = 2^3$, each digit of an octal number represents three binary digits. The octal digits are 0–7. The number $01010011_2 = 123_8$. When converting a binary number to octal or hexadecimal we must be sure to start with the right-most bits. Since there are only 8 bits in 01010011 the left-most octal digit corresponds to the left-most two binary digits. The other two octal digits each have three binary digits. Again, Python has built-in support for representing octal digits. Writing a number with a leading zero and the letter o means that it is in octal format. So $0o123$ is the Python representation of 123_8 and it is equal to 83_{10}.

Python 2 $\rightsquigarrow$ 3

Originally, octal numbers were written with a leading zero (i.e. 0123). In Python 3, octal numbers must be preceded with a zero and the letter o (i.e. 0o123).[8]

Practice 1.8 Convert the number 58_{10} to binary and then to hexadecimal and octal.

1.8 Writing Your Second Program

Writing programs is an error-prone activity. Programmer's almost never write a non-trivial program perfectly the first time. As programmers we need a tool like an Integrated Development Environment (i.e. IDE) that helps us find and fix our mistakes. Going to the *File* menu of the Wing IDE window and selecting *New* opens a new edit pane. An edit pane can be used to write a program but it won't execute each line as you press enter. When writing

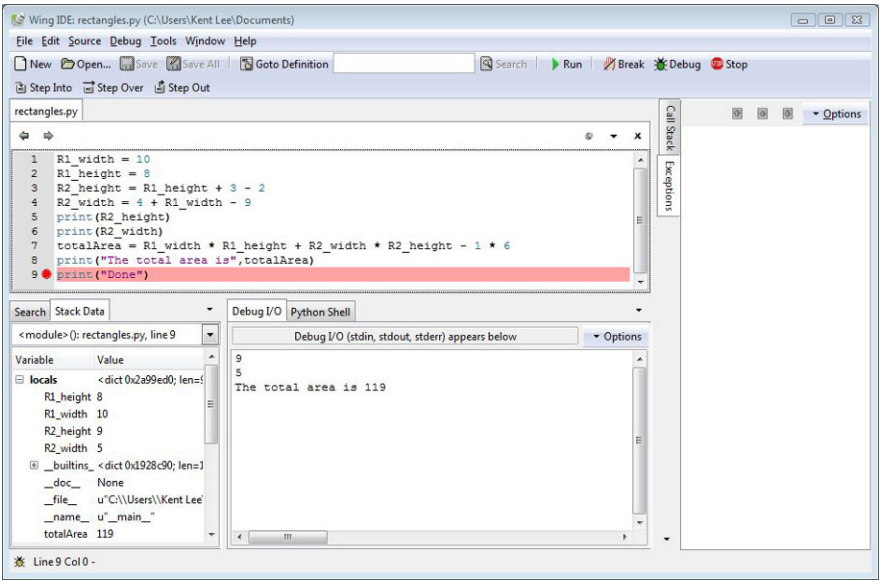


Fig. 1.13 The Wing IDE

a program we can write a little bit and then execute it in the Python interpreter by pressing *F5* on the keyboard or by clicking the debug button.

When we write a program we will almost certainly have to debug it. *Debugging* is the word we use when we have to find errors in our program. Errors are very common and typically you will find a lot of them before the program works perfectly. Debugging refers to removing bugs from a program. Bugs are another name for errors. The use of the words *bug* and *debugging* in Computer Science dates back to at least 1952 and probably much earlier. Wikipedia has an interesting discussion of the word *debugging* if you want to know more. While you can use the Python Shell for some limited debugging, a *debugger* is a program that assists you in debugging your program. Figure 1.13 has a picture of the Wing IDE with the program we've been working on typed into the editor part of the IDE. To use the debugger we can click the mouse in the area where the red circle appears next to the numbers. This is called setting a breakpoint. A breakpoint tells Python to stop running when Python reaches that statement in the program. The program is not finished when it reaches that step, but it stops so you can inspect the state of the program.

The state of the program is contained in the bottom left corner of the IDE. This shows you the *Stack Data* which is just another name for the program's state. You can see that the variables that were defined in the program are all located here along with their values at the present time.

1

Practice 1.9 Create an edit pane within the Wing IDE and write the program as it appears in Fig. 1.13. Write a few lines, then run it by pressing *F5* on the keyboard or clicking on the *Debug* button. The first time you press *F5* you will be prompted to save the program. Make sure you save your program where you can find it later.

Try setting a break point by clicking where the circle appears next to the numbers in Fig. 1.13. You should see a red circle appear if you did it right. Then run the program again to see that it stops at the breakpoint as it appears in Fig. 1.13. You can stop a program at any point by setting a breakpoint on that line. When the debugger stops at a breakpoint it stops before the statement is executed. You must click the *Debug* button, not the *Run* button to get it to stop at breakpoints.

Look at the Stack Data to inspect the state of the program just before the word *Done* is printed. Make sure it matches what you see here. Then continue the execution by clicking the Debug button or pressing *F5* again to see that *Done* is printed.

1.9 Syntax Errors

Not every error is found using a debugger. Sometimes errors are syntax errors. A syntax error occurs when we write something that is not part of the Python language. Many times a syntax error can occur if we forget to write something. For instance, if we forget a parenthesis or a double quote is left out it will not be a correct Python program. Syntax errors are typically easier to find than bugs in our program because Python can flag them right away for us. These errors are usually highlighted right away by the IDE or interpreter. Syntax errors are those errors that are reported before the program starts executing. You can tell its a syntax error in Wing because there will not be any *Stack Data*. Since a syntax error shows up before the program runs, the program is not currently executing and therefore there is not state information in the stack data. When a syntax error is reported the editor or Python will typically indicate the location of the error *after* it actually occurs so the best way to find syntax errors is to look backwards from where the error is first reported.

Example 1.4 Forgetting a parenthesis is a common syntax error.

```
print(R2_height
```

This is not valid syntax in Python since the right parenthesis is missing. If we were to try to run a Python program that contains this line, the Python interpreter complains that this is not valid syntax. Figure 1.14 shows how the Wing IDE tells us about this syntax error. Notice that the Wing IDE announces that the syntax error occurs on the line after where it actually occurred.

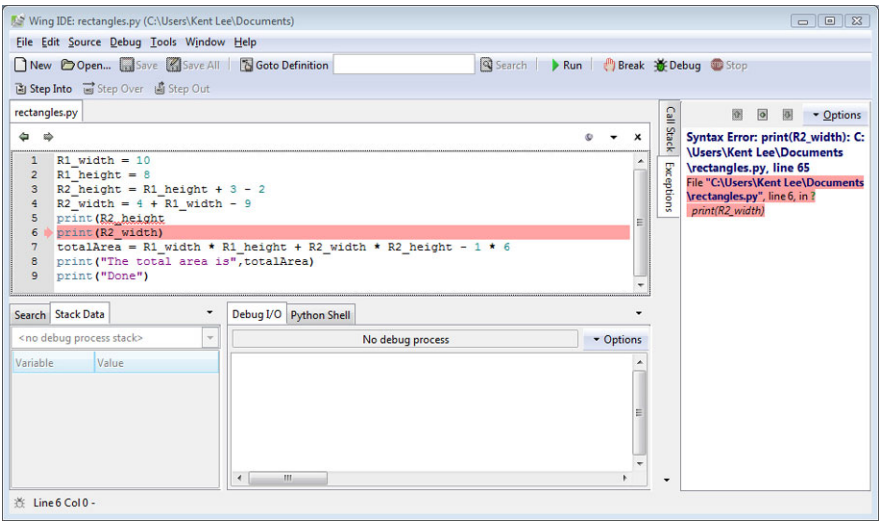


Fig. 1.14 A syntax error

There are other types of errors we can have in our programs. Syntax errors are perhaps the easiest errors to find. All other errors can be grouped into the category of *run-time errors*. Syntax errors are detected before the program runs. Run-time errors are detected while the program is running. Unfortunately, run-time errors are sometimes much harder to find than syntax errors. Many run-time errors are caused by the use of invalid operations being applied to values in our programs. It is important to understand what types of values we can use in our programs and what operations are valid for each of these types. That's the topic of the next section.

1.10 Types of Values

Earlier in this chapter we found that bytes in memory can be interpreted in different ways. The way bytes in memory are interpreted is determined by the *type* of the value or object and the operations we apply to these values. Each value in Python is called an object. Each object is of a particular type. There are several data types in Python. These include integer (called *int* in Python), *float*, boolean (called *bool* in Python), string (called *str* in Python), *list*, *tuple*, *set*, dictionary (called *dict* in Python), and *None*.

In the next chapters we'll cover each of these types and discuss the operations that apply to them. Each type of data and the operations it supports is covered when it is needed to learn a new programming skill. The sections on each of these types can also serve as a reference for you as you continue working through the text. You may find yourself coming back to the sections describing these types and their operations over and over

1

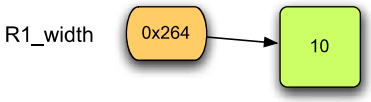


Fig. 1.15 A reference

again. Reviewing types and their operations is a common practice among programmers as they design and write new programs.

1.11 The Reference Type and Assignment Statements

There is one type in Python that is typically not seen, but nevertheless is important to understand. It is called the reference type. A reference is a pointer that points to an object. A pointer is the address of an object. Each object in memory is stored at a unique address and a reference is a pointer that points to an object.

An assignment statement makes a reference point to an object. The general form of an assignment statement is:

```
<identifier> = <expression>
```

An *identifier* is any letters, digits, or underscores written without spaces between them. The identifier must begin with a letter or underscore. It cannot start with a digit. The *expression* is any expression that when evaluated results in one of the *types* described in Sect. 1.10. The left hand side of the equals sign must be an identifier and only one identifier. The right hand side of the equals sign can contain any expression that may be evaluated.

In Fig. 1.15, the variable *R1_width* (orange in the figure) is a reference that points at the integer object 10 colored green in the figure. This is what happens in memory in response to the assignment statement:

```
R1_width = 10
```

The *0x264* is the reference value, written in hexadecimal, which is a pointer (i.e. the address) that points at the integer object 10. However, typically you don't see reference values in Python. Instead, you see what a reference points to. So if you type *R1_width* in the Python shell after executing the statement above, you won't see *0x264* printed to the screen, you'll see 10, the value that *R1_width* refers to. When you set a breakpoint and look at the stack data in the debugger you will also see what the reference refers to, not the reference itself (see Fig. 1.13).

It is possible, and common, in Python to write statements like this:

```
x = 1
# do something with x
x = x + 1
```

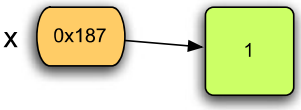


Fig. 1.16 Before incrementing x

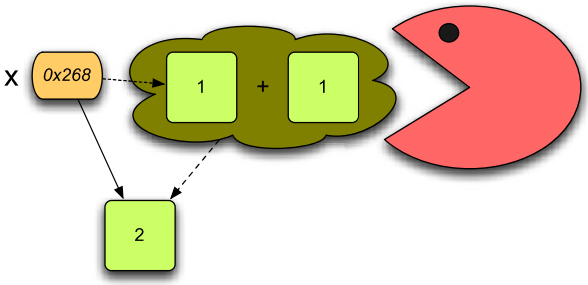


Fig. 1.17 After incrementing x

According to what we have just seen, Fig. 1.16 depicts the state of memory after executing the first line of code and before executing the second line of code. In the second line of code, writing $x = x + 1$ is not an algebraic statement. It is an assignment statement where one is added to the value that x refers to. The correct way to read an assignment statement is from right to left. The expression on the right hand side of the equals sign is evaluated to produce an object. The equals sign takes the reference to the new value and stores it in the reference named by the identifier on the left hand side of the equals sign. So, to properly understand how an assignment statement works, it must be read from right to left. After executing the second statement (the line beginning with a pound sign is a comment and is not executed), the state of memory looks like Fig. 1.17. The reference called x is updated to point to the new value that results from adding the old value referred to by x and the 1 together.

The space for the two left over objects containing the integers 1 in Fig. 1.17 is reclaimed by the *garbage collector*. You can think of the garbage collector as your favorite arcade game character running around memory looking for unattached objects (objects with no references pointing to them—the stuff in the cloud in Fig. 1.17). When such an object is found the garbage collector reclaims that memory for use later much like the video game character eats dots and fruit as it runs around.

The garbage collector reclaims the space in memory occupied by unreferenced objects so the space can be used later. Not all programming languages include garbage collection but many languages developed recently include it and Python is one of these languages. This is a nice feature of a language because otherwise we would have to be responsible for freeing all of our own memory ourselves.

1.12 Integers and Real Numbers

In most programming languages, including Python, there is a distinction between integers and real numbers. Integers, given the type name *int* in Python, are written as a sequence of

Operation	Operator	Comments
Addition	x + y	x and y may be floats or ints.
Subtraction	x - y	x and y may be floats or ints.
Multiplication	x * y	x and y may be floats or ints.
Division	x / y	x and y may be floats or ints. The result is always a float.
Floor Division	x // y	x and y may be floats or ints. The result is the first integer less than or equal to the quotient.
Remainder or Modulo	x % y	x and y must be ints. This is the remainder of dividing x by y.
Exponentiation	x ** y	x and y may be floats or ints. This is the result of raising x to the y^{th} power.
Float Conversion	float(x)	Converts the numeric value of x to a float.
Integer Conversion	int(x)	Converts the numeric value of x to an int. The decimal portion is truncated, not rounded.
Absolute Value	abs(x)	Gives the absolute value of x.
Round	round(x)	Rounds the float, x, to the nearest whole number. The result type is always an int.

Fig. 1.18 Numeric operations

digits, like 83 for instance. Real numbers, called *float* in Python, are written with a decimal point as in 83.0. This distinction affects how the numbers are stored in memory and what type of value you will get as a result of some operations.

In Fig. 1.18 the type of the result is a float if either operand is a float unless noted otherwise in the table.

Dividing the integer 83 by 2 yields 41.5 if it is written 81/2. However, if it is written 83//2 then the result is 41. This goes back to long division as we first learned in elementary school. 83//2 is 41 with a remainder of 1. The result of floor division isn't always an *int*. 83//2.0 yields 41.0 so be careful. While floor division returns an integer, it doesn't necessarily return an *int*.

We can insure a number is a float or an integer by writing *float* or *int* in front of the number. So, *float(83)//2* also yields 41.0. Likewise, *int(83.0)//2* yields 41.

There are infinitely many real numbers but only a finite number of floats that can be represented by a computer. For instance, the number *PI* is approximately 3.14159. However,

Python 2 ⇝ 3

In Python 2 the floor division operator was specifically for *floats*. If both operands were *ints* then integer (i.e. floor) division was automatically used by writing the / operator. If you are using Python 2 and want to use integer division then you must insure that both operands are ints. Likewise, if you want to use floating point division you must insure that at least one operand is a float. When using Python 2, to force floating point division of x/y you can write:

```
z = float(x) / y
```

It should also be noted that in Python 2 the *round* function returned the same type as its operand. In Python 3 the *round* function returns an *int*.[8]

that number can't be represented in some implementations of Python. Instead, that number is approximated as 3.1415899999999999 in at least one Python implementation. Writing 3.14159 in a Python program is valid, but it is still stored internally as the approximated value. This is not a limitation of Python. It is a limitation of computers in general. Computers can only approximate values when there are infinitely many possibilities because computers are finite machines.

You can use what is called integer conversion to transform a floating point number to its integer portion. In effect, integer conversion truncates the digits after the decimal point in a floating point number to get just the whole number part. To do this you write *int* in front of the floating point number you wish to convert. This does not convert the existing number. It creates a new number using only the integer portion of the floating point number.

Example 1.5 Assume that you work for the waste water treatment plant. Part of your job dictates that you report the gallons of water treated at the plant. However, your meter reports lbs of water treated. You have been told to to report the amount of treated waste water in gallons and ounces. There are 128 ounces in a gallon and 16 ounces in a pound. Here is a short program that performs the conversion.

```
1 lbs = float(input("Please enter the lbs of water treated: "))
2 ounces = lbs * 16
3 gallons = int(ounces / 128)
4 ounces = ounces - gallons * 128
5 print("That's",gallons,"gallons and", \
6       ounces,"ounces of treated waste water.")
```

In Example 1.5 the lbs were first converted to ounces. Then the whole gallons were computed from the ounces by converting to an integer the result of dividing the ounces float by 128. On line 4 the remaining ounces were computed after taking out the number of ounces contained in the computed gallons.

Several of the operations between ints and floats are given in Fig. 1.18. If you need to round a float to the nearest integer (instead of truncating the fractional portion) you can use the *round* function. Absolute value is taken using *abs*. There are other operations between floats and ints that are not discussed in this chapter. A complete list of all operations supported by integers and floats are given in Appendices and B. If you need to read some documentation about an operator you can use the appendices or you can search for Python documentation on the internet or you can start a Python shell and type *help*(*float*) or *help*(*int*). This help facility is built into the Python programming language. There is extensive documentation for every type within Python. Typing *help*(*type*) in the Python shell where *type* is any type within Python will provide you with all the operations that are available on that type of value.

Operation	Operator	Comments
Indexing	s[x]	Yields the x^{th} character of the string s. The index is zero based, so s[0] is the first character.
Concatenation	s + t	Yields the juxtaposition of the strings s and t.
Length	len(s)	Yields the number of characters in s.
Ordinal Value	ord(c)	Yields the ordinal value of a character c. The ordinal value is the ASCII code of the character.
Character Value	chr(x)	Yields the character that corresponds to the ASCII value of x.
String Conversion	str(x)	Yields the string representation of the value of x. The value of x may be an int, float, or other type of value.
Integer Conversion	int(s)	Yields the integer value contained in the string s. If s does not contain an integer an error will occur.
Float Conversion	float(s)	Yields the float value contained in the string s. If s does not contain a float an error will occur.

Fig. 1.19 String operations

Practice 1.10 Write a short program that computes the length of the hypotenuse of a right triangle given the two legs as pictured in Fig. 1.23 on page 32. The program should use three variables, *sideA*, *sideB*, and *sideC*. The Pythagorean theorem states that the sum of the squares of the two legs of the triangle equals the square of the hypotenuse. Be sure to assign all three variables their correct values and print the length of *sideC* at the end of the program. HINT: Raising a value to the $1/2$ power is the same thing as finding the square root. Try values 6 and 8 for *sideA* and *sideB*.

1.13 Strings

Strings are another type of data in Python. A string is a sequence of characters.

```
name = 'Sophus Lie'
print("A famous Norwegian Mathematician is", name)
```

This is a short program that initializes a variable called *name* to the string '*Sophus Lie*'. A string literal is an actual string value written in your program. String literals are *delimited* by either double or single quotes. Delimited means that they start and end with quotes. In the code above the string literal *Sophus Lie* is delimited by single quotes. The string *A famous Norwegian Mathematician is* is delimited by double quotes. If you use a

single quote at the beginning of a string literal, you must use a single quote at the end of the string literal. Delimiters must come in matching pairs.

Strings are one type of sequence in Python. There are other kinds of sequences in Python as well, such as lists which we'll look at in a couple of chapters. Python supports operations on sequences. For instance, you can get an individual item from a sequence. Writing,

```
print(name[0])
```

will print the first character of the string that *name* references. The 0 is called an index. Each subsequent character is assigned a subsequent position in the string. Notice the first position in the string is assigned 0 as its index. The second character is assigned index 1, and so on. Strings and their operations are discussed in more detail in Chap. 3.

Practice 1.11 Write the three line program given in the two listings on page 24. Then, without writing the string literal "house", modify it to print the string "house" to the screen using string indexing. HINT: You can add strings together to build a new string. So,

```
name = "Sophus" + " Lie"
```

will result in *name* referring to the string "*Sophus Lie*".

1.14 Integer to String Conversion and Back Again

It is possible in Python to convert an integer to a string. For instance,

```
x = str(83)
print(x[0])
print(x[1])
y = int(x)
print(y)
```

This program converts 83 to '83' and back again. Integers and floats can be converted to a string by using the *str* conversion operator. Likewise, an integer or a float contained in a string can be converted to its numeric equivalent by using the *int* or *float* conversion operator. Conversion between numeric types and string types is frequently used in programs especially when producing output and getting input.

Conversion of numeric values to strings should not be confused with ASCII conversion. Integers may represent ASCII codes for characters. If you want to convert an integer to its

1

ASCII character equivalent you use the *chr* conversion operator. For instance, *chr*(83) is '*S*'. Likewise, if you want to convert a character to its ASCII code equivalent you use the *ord* conversion operator. So *ord*('*S*') is equal to *83*.

Practice 1.12 Change the program above to convert 83 to its ASCII character equivalent. Save the value in a variable and print the following to the screen in the exact format you see here.

```
The ASCII character equivalent of 83 is S.
```

You might have noticed in Fig. 1.19 there is an operator called *int* and another called *float*. Both of these operators are also numeric operators and appear in Fig. 1.18. This is called an overloaded operator because *int* and *float* are operators that work for both numeric and string operands. Python supports overloaded operators like this. This is a nice feature of the language since both versions of *int* and *float* do similar things.

1.15 Getting Input

To get input from the user you can use the *input* function. When the *input* function is called the program stops running the program, prompts the user to enter something at the keyboard by printing a string called the *prompt* to the screen, and then waits for the user to press the *Enter* key. The user types a string of characters and presses enter. Then the *input* function returns that string and Python continues running the program by executing the next statement after the input statement.

Example 1.6 Consider this short program.

```
name = input("Please enter your name:")
print("The name you entered was", name)
```

The *input* function prints the prompt "Please enter your name:" to the screen and waits for the user to enter input in the Python Shell window. The program does not continue executing until you have provided the input requested. When the user enters some characters and presses enter, Python takes what they typed before pressing enter and stores it in the variable called *name* in this case. The type of value read by Python is always a string. If we want to convert it to an integer or some other type of value, then we need to use a

conversion operator. For instance, if we want to get an int from the user, we must use the *int* conversion operator.

Python 2 ⇝ 3

Python 2 included a way to get input also called *input*. This old version of the *input* function was very confusing to use and was eliminated in Python 3. In Python 2 the equivalent of Python 3's *input* was called *raw_input*. If you are writing a Python 2 program you may replace any call to *input* with a call to *raw_input*. If you are using Python 2, DO NOT use the *input* function. In Python 2 the value that the input function returned partially depended on the variable names in your program! This was a bad idea that led to much confusion in Python programs so almost no one used it and hence it was eliminated in Python 3.[8]

Practice 1.13 Assume that we want to pause our program to display some output and we want to let the user press some key to continue. We want to print "press any key to continue…" to the screen. Can we use the input function to implement this? If so, how would you write the input statement? If not, why can't you use input?

Example 1.7 This code prompts the user to enter their age. The string that was returned by *input* is first converted to an integer and then stored in the variable called *age*. Then the age variable can be added to another integer. It is important to remember that *input* always returns a string. If some other type of data is desired, then the appropriate type conversion must be applied to the string.

```
age = int(input("Please enter your age:"))
olderAge = age + 1
print("Next year you will be", olderAge)
```

1.16 Formatting Output

In this chapter just about every fragment of code prints something. When a value is printed, it appears on the console. The location of the console can vary depending on how you run a program. If a program is run from within the Wing IDE, the console is the *Python Shell* window in the IDE. If the program is debugged from within Wing IDE 101, the output appears in the *Debug I/O* window.

1

When printing, we may print as many items as we like on one line by separating each item by a comma. Each time a comma appears between items in a print statement, a space appears in the output.

Example 1.8 Here is some code that prints a few values to the screen.

```
name = "Sophus"
print(name,"how are you doing?")
print("I hope that,", name, "is feeling well today.")
```

The output from this is:

```
Sophus how are your doing?
I hope that Sophus is feeling well today.
```

To print the contents of variables without spaces appearing between the items, the variables must be converted to strings and string concatenation can be used. The + operator adds numbers together, but it also concatenates strings. For the correct + operator to be called, each item must first be converted to a string before concatenation can be performed.

Example 1.9 Assume that we ask the user to enter two floating point numbers, x and y, and we wish to print the result of raising x to the yth power. We would like the output to look like this.

```
Please enter a number: 4.5
Please enter an exponent: 3.2
4.5^3.2 = 123.10623351
```

Here is a program that will produce that output, with no spaces in the exponentiation expression. NOTE: The caret symbol (i.e. ^) is not the Python symbol for exponentiation.

```
1 base = float(input("Please enter a number: "))
2 exp = float(input("Please enter an exponent: "))
3 answer = base ** exp
4 print(str(base) + "^" + str(exp), "=", answer)
```

In Example 1.9, line 4 of the program prints three items to the console. The last two items are the = and the value that the answer variable references. The first item in the print statement is the result of concatenating *str*(*base*), the caret, and *str*(*exp*). Both *base* and *exp* must be converted to strings first, then string concatenation will be performed by the + operator because the operands on either side of the + are both strings.

Type	Specifier	Comments
int	%*w***d**	Places an integer in a field of width *w* if specified. %2d would place an integer in a field of width 2. *w* may be omitted.
int	%*w***x**	Format the integer in hexadecimal. Put it in a field of width *w* if specified. *w* may be omitted.
int	%*w***o**	Format the integer in octal. Put it in a field of width *w* if specified. *w* may be omitted.
float	%*w*.*d***f**	Format a floating point number with total width *w* (including the decimal point) and with *d* digits after the decimal point. Displaying the entire number include the *d* digits takes precedence over displaying in a field of *w* characters should *w* not be big enough. *w* and *d* may be omitted.
float	%*w*.*d***e**	Format a floating point number using scientific notation with *d* digits of precision and *w* field width. Scientific notation uses an exponent of 10 to move the decimal point so only one digit appears to the left of the decimal point. *w* and *d* may be omitted.
str	%*w***s**	Place a string in a field of width *w*. *w* may be omitted.
	%%	Include a % sign in the formatted string.

Fig. 1.20 Format specifiers

Practice 1.14 The sum of the first n positive integers can be computed by the formula

$$sum(1..n) = 1 + 2 + 3 + 4 + \cdots + n = n(n+1)/2$$

Write a short Python program that computes the sum of the first 100 positive integers and prints it to the screen in the format shown below. Use variables to represent the 1, the 100, and the result of the computation. Your program must compute the 5050 value. You cannot just print the result to the screen. You must compute it first from the 100.

```
sum(1..100)=5050
```

For advanced control of the format of printing we can use string formatting. String formatting was first used in the C language *printf* function back in the 1970's. It's an idea that has been around a long time, but is still useful. The idea is that we place formatting instructions in a string and then tell Python to replace the formatting instructions with the actual values. This is best described with an example.

Example 1.10 Assume we wish to re-implement the program in Example 1.9. However, in this version of the program, if the user enters more than 2 decimal places for either

1

number we wish to round the numbers to two digits of precision when they are printed to the console. Assume we wish to round the answer to 4 decimal places when displayed. The following code will do this.

```
1 base = float(input("Please enter a number: "))
2 exp = float(input("Please enter an exponent: "))
3 answer = base ** exp
4 print("%1.2f^%1.2f = %1.4f"%(base,exp,answer))
```

Running this program produces the following output.

```
Please enter a number: 4.666666667
Please enter an exponent: 3.3333333333
4.67^3.33 = 169.8332
```

Line 4 in Example 1.10 prints the result of formatting a string. To use Python formatting, a format string must be written first, followed by a percent sign, followed by the replacement values. If there is more than one replacement value, they must be written in parentheses. Each time a `%` appears inside the format string it is replaced by one of the values that appear after the format string. How a value is formatted when it is placed in the format string is controlled by the format specifier. Figure 1.20 contains some specifiers for common types of data in Python. Every format specifier may include an optional width field. If specified, the width field specifies the actual width of the replaced data. If the width of the data being inserted into the format string exceeds the alloted width, the entire field is included anyway, stretching the width of the formatted string. String formatting can be very useful when generating a printed report of some data.

Practice 1.15 Re-do Practice 1.14 using format specifiers when printing instead of converting each item to a string. The goal is for the output to look exactly the same.

```
sum(1..100)=5050
```

1.17 When Things Go Wrong

As a programmer, you will soon discover that things can go wrong when writing a program. No programmer writes every program correctly the first time. We are all human and make mistakes. What makes a programmer a really good programmer is when they can find their

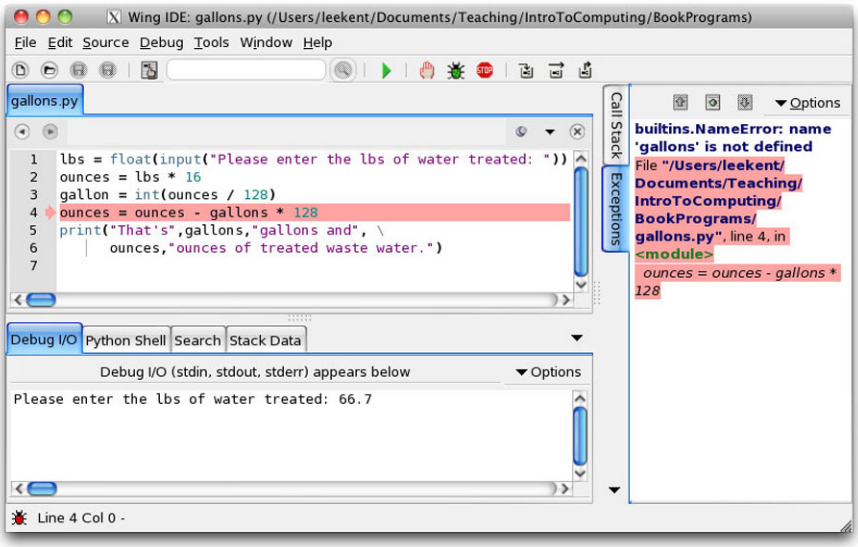


Fig. 1.21 A run-time error

mistakes and correct them. Debugging programs is a skill that can be learned and therefore can be taught as well. But, it takes lots of practice and patience. Fortunately, you will have many chances to practice as you work your way through this book.

Sometimes, especially when you are first learning to debug your programs, it can help to have someone to talk to. Just the act of reading your code to someone else may cause you to find your mistake. Of course, if you are using this text as part of a course you may not want to read your code to another class member as that may violate the guidelines your instructor has set forth. But, nevertheless, you might find that reading your code to someone else may help you discover problems. This is called a code walk-through by programming professionals. It is a common practice and is frequently required when writing commercially available programs.

There is no substitute for thorough testing. You should run your program using varied values for input. Try to think of values that might cause your program to break. For instance, what if 0 is entered for an integer? What if a non-integer value is entered when an integer was required? What happens if the user enters a string of characters when a number was required?

Sometimes the problems in our code are not due to user input. They are just plain old mistakes in programming caused either by temporarily forgetting something, or by our misunderstanding how something works. For instance, in this chapter we learned about assignment statements. You can store a value in the memory of a computer and point a named reference at the value so you can retrieve it later. But, you must assign a name to a value before you can retrieve it. If you don't understand that concept, or if you forgot where you assigned a value a name in your program, you might accidentally write some code that tries to use that value before it is assigned a name. For instance, consider the

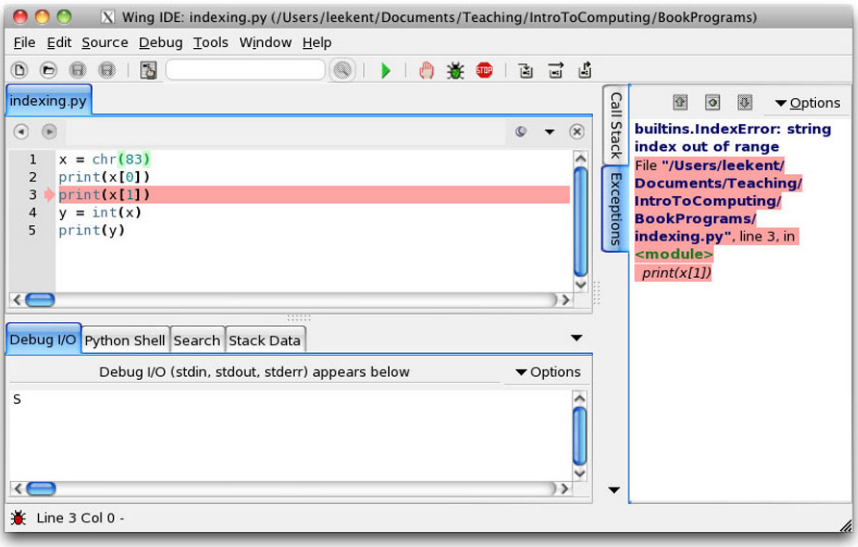
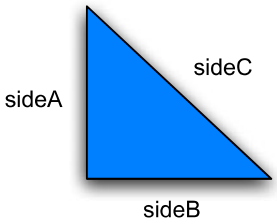


Fig. 1.22 An index out of range error

Fig. 1.23 A right triangle



program in Fig. 1.21. The program is trying to use the gallons variable which has not been assigned a value. The error message is on the right side of the window. The line where the error was first detected by Python is highlighted.

In the example in Fig. 1.21 the actual error is not on the line that is highlighted. The highlighted line is the line where Python first *detected* the error. This is a very common occurrence when debugging. Detection of an error frequently occurs after the location of the actual error. To become a good programmer you must learn to look backwards through your code from the point where an error is detected to find the location where it occurred. In this case, the *gallon* variable should have been written as *gallons* on line 3 but was incorrectly typed.

Another common error is the *index out of range* error. This can occur when trying to access a value in a sequence by indexing into the sequence. If the index is for an item that is outside the range of the sequence, an index out of range error will occur. For instance, if you have a string called x that is one character long and you try to access the second element of the string, your program will abort with an index out of range error. Figure 1.22 shows this happening in a snippet of code.

Once again, in the example in Fig. 1.22 the error did not occur on the line that is highlighted. The error occurred because the programmer meant to take the *str*(*83*) which would result in "83" as a string instead of the *chr*(*83*) which results in the string "S". If the string had been "83" then line 3 would have worked and would have printed 3 to the screen.

When an error occurs it is called an uncaught exception. Uncaught exceptions result in the program terminating. They cannot be recovered from. Because uncaught exceptions result in the program terminating it is vital to test your code so that all variations of running the program are tested before the program is released to users. Even so, there are times when a user may encounter an error. Perhaps it has happened to you? In any case, thorough testing is critical to your success as a programmer and learning to debug and test your code is as important as learning to program in the first place. As new topics are introduced in this text, debugging techniques will also be introduced to provide you with the information you need to become a better debugger.

1.18 Review Questions

1. What does the acronym IDE stand for? What does it do?
2. What does the acronym CPU stand for? What does it do?
3. How many bytes are in a GB? What does GB stand for?
4. What is the decimal equivalent of the binary number 01101100?
5. What is the hexadecimal equivalent of the binary number 01101100?
6. What is the binary equivalent of the number -62?
7. What is the ASCII equivalent of the decimal number 62?
8. What is a type in Python? Give an example. Why are there types in Python programs?
9. How can you tell what type of value is stored in 4 contiguous bytes of memory?
10. How can you interactively work with the Python interpreter?
11. What is prototyping as it applies to computer programming?
12. Name two different types of errors that you can get when writing a computer program? What is unique about each type of error?
13. What is a reference in a Python program?
14. Why shouldn't you compare floats for equality?
15. What would you have to write to ask the user to enter an integer and then read it into a variable in your program? Write some sample code to do this.
16. Assume that you have a constant defined for $pi = 3.14159$. You wish to print just 3.14 to the screen using the *pi* variable. How would you print the *pi* variable so it only display 3.14?

1

1.19 Exercises

1. Write a program that asks the user to enter their name. Then it should print out the ASCII equivalent of each of the first four characters of your name. For instance, here is a sample run of the program below.

```
Please enter your name: Kent
K ASCII value is 75
e ASCII value is 101
n ASCII value is 110
t ASCII value is 116
```

2. Write a program that capitalizes the first four characters of a string by converting the characters to their ASCII equivalent, then adding the necessary amount to capitalize them, and converting the integers back to characters. Print the capitalized string. Here is a sample of running this program.

```
Please enter a four character string: kent
The string capitalized is KENT
```

3. You can keep track of your car's miles per gallon if you keep track of how many miles you drive your car on a tank of gas and you always fill up your tank when getting gas. Write a program that asks the user to enter the number of miles you drove your car and the number of gallons of gas you put in your car and then prints the miles per gallon you got on that tank of gas. Here is a sample run of the program.

```
Please enter the miles you drove: 256
Please enter the gallons of gas you put in the tank: 10.1
You got 25.346534653465348 mpg on that tank of gas.
```

4. Write a program that converts US Dollars to a Foreign Currency. You can do this by finding the exchange rate on the internet and then prompting for the exchange rate in your program. When you run the program it should look exactly like this:

```
What is the amount of US Dollars you wish to convert? 31.67
What is the current exchange rate
(1~US Dollar equals what in the Foreign Currency)? 0.9825
The amount in the Foreign Currency is $31.12
```

5. Write a program that converts centimeters to yards, feet, and inches. There are 2.54 centimeters in an inch. You can solve this problem by doing division, multiplication, addition, and subtraction. Converting a float to an int at the appropriate time will help in solving this problem. When you run the program it should look exactly like this (except possibly for decimal places in the inches):

```
How many centimeters do you want to convert? 127.25
This is 1 yards, 1 feet, 2.098425 inches.
```

6. Write a program that computes the minimum number of bills and coins needed to make change for a person. For instance, if you need to give $34.36 in change you would need one twenty, one ten, four ones, a quarter, a dime, and a penny. You don't have to compute change for bills greater than $20 dollar bills or for fifty cent pieces. You can solve this problem by doing division, multiplication, subtraction, and converting floats to ints when appropriate. So, when you run the program it should look exactly like this:

```
How much did the item cost: 65.64
How much did the person give you: 100.00
The person's change is $34.36
The bills or the change should be:
1 twenties
1 tens
0 fives
4 ones
1 quarters
1 dimes
0 nickels
1 pennies
```

7. Write a program that converts a binary number to its decimal equivalent. The binary number will be entered as a string. Use the powers of 2 to convert each of the digits in the binary number to its appropriate power of 2 and then add up the powers of two to get the decimal equivalent. When the program is run, it should have output identical to this:

```
Please enter an eight digit binary number: 01010011
The decimal equivalent of 01010011 is 83.
```

8. Write a program that converts a decimal number to its binary equivalent. The decimal number should be read from the user and converted to an *int*. Then you should follow the algorithm presented in Example 1.1 to convert the decimal number to its binary equivalent. The binary equivalent must be a string to get the correct output. The output from the program must be identical to this:

```
Please enter a number: 83
The binary equivalent of 83 is 01010011.
```

   You may assume that the number that is entered is in the range 0-255. If you want to check your work, you can use the *bin* function. The *bin* function will take a decimal number and return a string representation of that binary number. However, you should not use the *bin* function in your solution.

1

9. Complete the program started in practice problem 1.10. Write a program that asks the user to enter the two legs of a right triangle. The program should print the length of the hypotenuse. If *sideA* and *sideB* are the lengths of the two legs and *sideC* is the length of the third leg of a right triangle, then the Pythagorean theorem says that $sideA^2 + sideB^2 = sideC^2$. Ask the user to enter *sideA* and *sideB*. Your program should print the value of *sideC*.

```
Please enter the length of the first leg: 3
Please enter the length of the second leg: 4
The length of the hypotenuse is 5.0
```

1.20 Solutions to Practice Problems

These are solutions to the practice problems in this chapter. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

Solution to Practice Problem 1.1

The decimal equivalent of the binary number 01010101_2 is 85.

Solution to Practice Problem 1.2

$58/2 = 29$ *remainder* 0
$29/2 = 14$ *remainder* 1
$14/2 = 7$ *remainder* 0
$7/2 = 3$ *remainder* 1
$3/2 = 1$ *remainder* 1
$1/2 = 0$ *remainder* 1

So the answer is 00111010_2.

Solution to Practice Problem 1.3

$-83_{10} = 10101101_2$

Solution to Practice Problem 1.4

The ASCII code for space is 32. $32_{10} = 00100000_2$

Solution to Practice Problem 1.5

We, as programmers, determine how bytes in memory are interpreted by the statements that we write. If we want to interpret the bits 01010011 as a character we write '*S*' in our program. If we want the same bits to represent an integer, we write *83* in our program.

Solution to Practice Problem 1.6

There is no solution needed for this exercise. Try it out and if you have problems, talk to your instructor or someone who can help to make sure you get this working before proceeding.

Solution to Practice Problem 1.7

1. An assignment statement is written as

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

   where a variable is assigned the value of an expression.
2. To retrieve a value from memory we write the name of the variable that refers to that value.
3. If we use a variable before it has been assigned a value Python will complain of a name error, meaning the variable has not been assigned a value yet.

Solution to Practice Problem 1.8

The binary representation of 58 is 00111010. The number is $3A_{16}$ and 72_8. In Python syntax that would be $0x3A$ and $0o72$.

Solution to Practice Problem 1.9

There is no solution needed for this since it is in the text. However, you should make sure you try this so you understand the mechanics of writing a program using the IDE. If you can't get it to work you should ask someone that did get it to work for help or ask your instructor.

1

Solution to Practice Problem 1.10

```
sideA = 6
sideB = 8
sideC = (sideA*sideA + sideB**2) ** 0.5
print(sideC)
```

Solution to Practice Problem 1.11

Here is one program that you might get as a result.

```
name = 'Sophus Lie'
print("The name is", name)
word = name[3] + name[1] + name[4] + name[5] + name[9]
print(word)
```

Solution to Practice Problem 1.12

Here is one version of the program. Do you understand why + was used at the end of the print statement?

```
x = chr(83)
print("The ASCII character equivalent of",ord(x),"is",x+".")
```

Solution to Practice Problem 1.13

You cannot use input to implement this because the input function waits for the enter key to be pressed, not just any key. You could prompt the user though with "Press Enter to continue…".

Solution to Practice Problem 1.14

Here is a version of the program. It must have variables to 1 and 100 to be correct according to the directions.

```
start = 1
end = 100
sumOfNums = end * (end + 1) // 2
print("sum("+str(start)+".."+str(end)+")="+str(sumOfNums))
```

Solution to Practice Problem 1.15

```
start = 1
end = 100
sumOfNums = end * (end + 1) // 2
print("sum(%d..%d)=%d"%(start,end,sumOfNums))
```