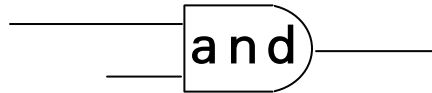


# Mathematics

for the Digital Age



# Programming

in Python

>>> Second Edition:  
with Python 3

Maria Litvin

Phillips Academy, Andover, Massachusetts

Gary Litvin

Skylight Software, Inc.

Skylight Publishing  
Andover, Massachusetts

Skylight Publishing  
9 Bartlet Street, Suite 70  
Andover, MA 01810

web: <http://www.skylit.com>  
e-mail: [sales@skylit.com](mailto:sales@skylit.com)  
[support@skylit.com](mailto:support@skylit.com)

**Copyright © 2010 by Maria Litvin, Gary Litvin, and  
Skylight Publishing**

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the authors and Skylight Publishing.

Library of Congress Control Number: 2009913596

ISBN 978-0-9824775-8-8 (soft cover)

ISBN 978-0-9824775-4-0 (hard cover)

The names of commercially available software and products mentioned in this book are used for identification purposes only and may be trademarks or registered trademarks owned by corporations and other commercial entities. Skylight Publishing and the authors have no affiliation with and disclaim any sponsorship or endorsement by any of these products' manufacturers or trademarks' owners.

1 2 3 4 5 6 7 8 9 10      15 14 13 12 11 10

Printed in the United States of America

## 2 An Introduction to Programming

### 2.1 Prologue

For a casual computer user, a computer program is something that comes on a CD or is downloaded from the Internet, and then runs on the computer screen — not unlike a TV show, except that the user has some control over what happens. For a programmer, a program is a set of instructions that the computer executes to perform precisely defined tasks. Actually, computer programming is much more than just “coding” these instructions — it involves many skills, including software design, devising algorithms, designing *user interfaces* (screens, commands, menus, toolbars, etc.), writing and testing code, and interacting with the users of software.

### 2.2 CPU and Memory

At the heart of a computer is the *Central Processing Unit (CPU)*. In a personal computer, the CPU is a microprocessor made from a tiny chip of silicon. The chip has millions of *transistors* etched on it. A transistor is a microscopic digital switch: it controls two states of a signal, “on” or “off,” “1” or “0.” The microprocessor is protected by a small ceramic case mounted on a *printed circuit board* called the *motherboard*. Also on the motherboard are memory chips.

The computer memory is a uniform pool of storage units called *bytes*.

#### ■ One byte holds eight bits.

A bit stores the smallest possible unit of information: “1” or “0”, “true” or “false.”

A CPU does not have to read or write memory bytes sequentially: bytes can be accessed in any order. This is why computer memory is called *random-access memory* or *RAM*). The same memory is used to store different types of information: numbers, letters, sounds, images, programs, and so on. All these things must be encoded, one way or another, as sequences of 0s and 1s.

A typical personal computer made in the year 2009 had 2 to 4 “gigs” (gigabytes) of RAM.

**1 kilobyte (KB) = 1024 bytes =  $2^{10}$  bytes, approximately one thousand bytes.**

**1 megabyte (MB) = 1024 kilobytes =  $2^{20}$  bytes = 1,048,576 bytes, approximately one million bytes.**

**1 gigabyte (GB) = 1024 megabytes =  $2^{30}$  bytes = 1,073,741,824 bytes, approximately one billion bytes.**

The CPU interprets and executes instructions stored in RAM. The CPU fetches the next instruction, interprets its operation code, and performs the appropriate operation. There are instructions for arithmetic and logical operations, for copying bytes from one location to another, and for changing the order of execution of instructions. The instructions are executed sequentially, unless a particular instruction tells the CPU to “jump” to another place in the program. *Conditional branching* instructions tell the CPU to continue with the next instruction or jump to another place depending on the result of the previous operation.

All this happens at amazing speeds. Each instruction takes one or several *clock cycles*, and a modern CPU runs at the speed of several GHz (gigahertz, that is, billion cycles per second).

To get a better feel for what CPU instructions are and how they are executed, let’s consider a couple of examples. This will involve a brief glimpse of *Assembly Language*, the primitive computer language that underlies the modern languages you have heard of, such as C++, Java, and Python.

## Example 1

The screen shot in Figure 2-1 shows a session with a program called *debug*. *debug* is an ancient program that was originally supplied with the MS-DOS operating system but still runs under *Windows*. *debug* allows a programmer to execute a program step by step, in a controlled manner, and examine the contents of memory at each step. This can help locate mistakes when the program is not working as expected. (Mistakes in computer code are called *bugs*, and the process of ridding a program of mistakes is called *debugging*.)

*debug* understands a few simple commands. The “a” command means “assemble”: it allows you to type in a few CPU instructions. Here we’ve used it to enter a small segment of a computer program. These instructions are written in Assembly Language for the Intel 8088 microprocessor (which was used in the original IBM PC in the early 1980s), but they are still compatible with modern Intel CPUs. Assembly

Language is very close to the actual machine language, but it allows you to use mnemonic names, rather than digits, for instruction codes.

```

C:\mywork>debug
-a 0100
0AF9:0100 mov bx,0
0AF9:0103 mov ax,1
0AF9:0106 cmp ax,6
0AF9:0109 jg 0110
0AF9:010B add bx,ax
0AF9:010D inc ax
0AF9:010E jmp 0106
0AF9:0110 nop
0AF9:0111
-u 0100 0110
0AF9:0100 BB0000      MOV     BX,0000
0AF9:0103 B80100      MOV     AX,0001
0AF9:0106 3D0600      CMP     AX,0006
0AF9:0109 7F05       JG     0110
0AF9:010B 01C3       ADD     BX,AX
0AF9:010D 40        INC     AX
0AF9:010E EBF6       JMP     0106
0AF9:0110 90        NOP
-g =0100 0110

AX=0007 BX=0015 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0AF9 ES=0AF9 SS=0AF9 CS=0AF9 IP=0110  NU UP EI PL NZ NA PO NC
0AF9:0110 90        NOP
-q

```

Figure 2-1. A session with the *debug* program

The CPU has several temporary storage locations, called *registers*. The above code works with two registers, called AX and BX. For example, the first instruction that we typed in — `mov bx, 0` — moves zero into the BX register.

After typing in the instructions, we verified our input by using *debug*'s “u” (unassemble) command. For each instruction, *debug* showed its address in memory, the instruction encoding, and the corresponding instruction in Assembly Language. The addresses and instruction codes are shown in “hex” (hexadecimal) system. We will explain the hexadecimal system in Chapter 5.

We then ran this code, using *debug*'s “g” (“go”) command. We told it to start at the address 0100 and stop when it gets to address 0110. When done, the “g” command displayed the contents of all the CPU registers and the instruction that will be executed next. Here it is a NOP (no operation) instruction — an instruction that does nothing.

Below is the explanation of what each instruction in this code segment does:

Hex address	Hex instruction code	Assembly language instruction	Our comment
0AF9:0100	BB0000	MOV BX,0000	; move 0 into the BX reg
0AF9:0103	B80100	MOV AX,0001	; move 1 into the AX reg
0AF9:0106	3D0600	CMP AX,0006	; compare AX to 6
0AF9:0109	7F05	JG 0110	; if greater, jump to 0110
0AF9:010B	01C3	ADD BX,AX	; add AX to BX
0AF9:010D	40	INC AX	; increment AX by 1
0AF9:010E	EBF6	JMP 0106	; jump back to 0106
0AF9:0110	90	NOP	; no operation -- skip

We leave it to you as an exercise (Question 7) to figure out what this code computes. The result is stored in the BX register.

### Example 2

The Intel 8088 instruction set includes the `call` instruction, which emulates a call to a function. `call` saves the return address (the address of the instruction that follows `call`) on the *system stack* and passes control to the instruction at the specified address (the beginning of the function code). The program continues until it encounters the `ret` (return) instruction. `ret` fetches the return address from the system stack and passes control to the instruction at that address.

In the code below, the function starts at the address 0100. The AX register serves as the input, and the BX register serves as the output. This is a very simple function: the output is equal to input plus one:

```
0AF9:0100 mov bx,ax ; move ax into bx
0AF9:0102 inc bx   ; increment bx
0AF9:0103 ret     ; return
```

We can call this function from another place in the program. For example:

```
0AF9:0106 mov ax,3
0AF9:0109 call 0100
```

When we execute these two instructions, we get

```
AX=0003 BX=0004 ...
```

If we execute

```
0AF9:010E mov ax,5
0AF9:0111 call 0100
```

we get

```
AX=0005 BX=0006 ...
```

## Exercises

1. Find a discarded desktop computer, make sure the power cord is unplugged, and remove the cover. Identify the motherboard, CPU, and memory chips. Identify other components of the computer: power supply, hard disk, CD-ROM drive, and other components.
2. Computer memory is called RAM because: ✓
  - A. It provides rapid access to data.
  - B. It is mounted on the motherboard.
  - C. It is measured in megabytes.
  - D. Its bytes can be addressed in random order.
  - E. Its chips are mounted in a rectangular array.
3. My old PC has 512 meg of RAM and a 120 gig hard drive. How many times more storage space does the hard disk have, as compared to RAM? ✓
4. How many different values can be encoded in 2 bits? 3 bits? 1 byte?
5. ASCII (read: 'as-kee) code represents upper- and lowercase letters of the English alphabet and other characters that you can find on a typical American keyboard. Each character is encoded in the same number of bits. Is one byte per character sufficient to represent all these characters? What is the smallest number of bits needed per character? ✓
6. In the program in Example 1, after we have issued the command

```
-g =0100 0110
```

how many times is the `cmp` instruction executed? ✓

7. ■ Explain the contents of the `AX` and `BX` registers after the program segment in Example 1 has been executed. What does this code compute? ≡ Hint: “hex” 15 is decimal 21. ≧
8. ♦ In 8088 Assembly Language, you can give names to memory locations and then refer to them by name. For example:

```
v1    dw    5 ; reserve 2 bytes to hold an integer, call it v1
      ;     ; set its initial value to 5
v2    dw    6

...
mov   ax,v1
```

You can also assign a label to a particular instruction and use it instead of the instruction address. For example:

```
L1:   cmp   ax,bx
      ...
      jmp  L1
```

Write a segment of code that moves the larger of the values stored in memory locations `v1` and `v2` into the `AX` register. Your code should consist of four instructions. ≡ Hint: in an 8088 CPU you can compare a memory location to a register, but you cannot compare two memory locations to each other in one instruction. ≧

9. ♦ Write a function in 8088 Assembly Language that takes the `AX` register as input and places its absolute value into `BX`, leaving `AX` unchanged. ≡ Hint: 8088 Assembly Language has the `neg` instruction, which negates the value in a register. ≧

## 2.3 Python Interpreter

It would be extremely tedious to write programs as sequences of digits (although in the very early days of the computer era, programmers did just that). Luckily, people quickly realized that they could define special languages for writing programs and use the computer itself to translate their programs from a high-level programming language into machine code. Some early programming languages were FORTRAN, COBOL, and BASIC. Some of the languages that are popular now are C++, C#, Java, Perl, and Ruby. Python is another popular programming language; it was

created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands.

Using a high-level programming language, you can write statements that translate into several CPU instructions. Figure 2-2 shows a function written in Python.

```
def sum1ToN(n):
    "Returns 1 + 2 + ... + n"
    s = 0
    k = 1
    while k <= n:
        s += k    # add k to s
        k += 1   # increment k by 1
    return s
```

**Figure 2-2. A function written in Python**

A program written in a machine language or an Assembly Language works only on a computer with a compatible CPU. A program written in a high-level language can be used with any CPU. For example, it can run on a PC or on a Mac.

There are two ways to convert a program written in a high-level programming language into machine code. The first approach is called *compiling*: a special program, called a *compiler*, examines the text of the program, generates appropriate machine language instructions, and saves them in an executable file. Once a program is compiled, the compiler is not needed to run it. The second method is called *interpreting*: a special program, called an *interpreter*, examines the text of the program, generates the appropriate instructions, and executes these instructions right away. An interpreter does not create an executable file.

Compiling is like making a written translation of a text from a foreign language; interpreting is like doing a simultaneous interpretation while a foreign speaker is talking. An interpreter can read a program from a file, or it can allow you to enter program statements line by line, interactively.

Modern languages, such as Java and Python, use a hybrid approach. First they compile a program into an intermediate low-level language, called *bytecode*. Then they interpret the bytecode, which is still independent of a particular CPU, but is much more compact, closer to the machine language, and easier to interpret.

The text of a program is governed by rather rigid *syntax rules*: you can't just type whatever you want and expect the computer to understand it.

**Every symbol in your program must be in just the right place.**

In English or another natural language, you can misspell a word or omit a few punctuation marks and still produce a readable text. This is because natural languages have *redundancy*: information is transmitted with less than optimal efficiency, but this lets the reader interpret a message correctly even if it has been somewhat garbled (Figure 2-3).



**Figure 2-3. A story by Lyla Fletcher Groom, age 5**  
Courtesy The Writing Workshop, [www.writingworkshop.com.au](http://www.writingworkshop.com.au)

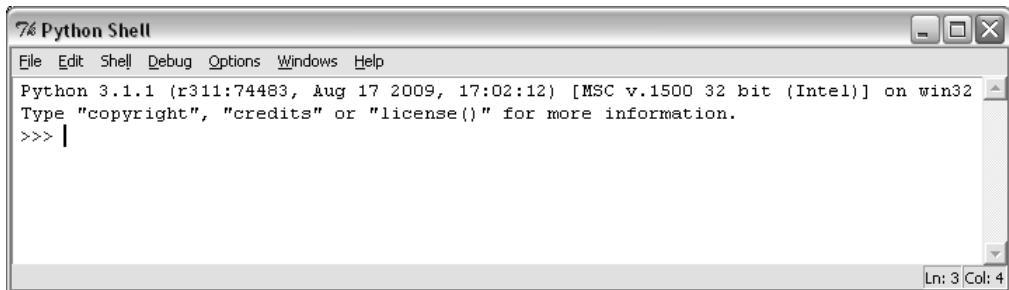
Programming languages have virtually no redundancy: almost every character is essential. There are many opportunities to make a mistake, so computer programmers have to learn patience and attention to detail.



We are now ready to experiment with Python. Python is available for free, even for commercial applications, under the *Open Source* license. The Python license is administered by the Python Software Foundation.

In a compiled language, you need to create the program text and save it in a file called *source code*, then run the source code file through a compiler to get an executable program. In Python, too, you can read a program from a source file. But you can also enter individual statements into the Python interpreter and see the result immediately.

Under *Windows*, it is more convenient to run the Python interpreter with a GUI (Graphical User Interface) front end. It is called *IDLE* (Figure 2-4).



**Figure 2-4. Python GUI shell under *Windows***

`>>>` is the Python interpreter's prompt. A *prompt* is a signal from a program that it is waiting for user input. The user can type in a statement; when the user presses `<Enter>`, the interpreter displays the result. For example, type

```
>>> 2+2 <Enter>
```

(user input is shown in bold). Python will respond:

```
4  
>>>
```

Looks reasonable! A number of things happen here. The interpreter reads the line of text (the statement) that you typed. It then analyzes the text and finds that the statement has two numbers (integers) separated by a + sign. The process of analyzing a text and extracting its components is called *parsing*.

A little experimentation will convince you that spaces do not matter (as long as the statement starts right after the prompt): you can type `2 + 2`, `2 +2`, or `2 + 2` — the result is the same. But if you type `2+*2`, you will get

```
>>> 2+*2
      ^
SyntaxError: invalid syntax
>>>
```

Now try:

```
>>> 2(3+4)
```

You would expect 14, right? No! What you get is

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    2(3+4)
TypeError: 'int' object is not callable
```

Clearly Python (we will call the interpreter “Python” for short, the same as the language) “thinks” there is something wrong with the statement you typed. But instead of reporting a syntax error, Python reports something else, which does not appear very helpful. (`TypeError` refers to the *type* of an object — an integer, a function, etc. — not to what you typed on the keyboard!) Apparently Python has decided that you were trying to call a function, named 2, with the input value 3+4, so Python tells you, in its own cryptic way, that 2 is not a function! You might think that Python can be really dumb sometimes. In fact, it is neither smart nor dumb — it’s just a program.

Meanwhile, what you really meant was

```
>>> 2*(3+4)
```

Perhaps you thought that the multiplication sign was optional, like in math. Not so. Just as we warned you: every character matters!

## Exercises

1. Define *redundancy*.
2. Type `2+-2` into the Python interpreter. Is this valid syntax? Explain the result. Now do the same for `2++2`.
- 3.▪ Try `2+++2`. Explain the result. ✓
4. Try `2**3` and `2**4`. What does Python's operator `**` do?
5. In Python, pieces of text in single or double quotes represent *literal strings*. Try `"abc" + "def"` and `'abc' + 'def'`. Explain what the `+` operator does when applied to strings. ✓
6. Can the `*` operator be applied to an integer and a string? Try `3 * '12'` and explain the result.
7. Type in `9-8*2+6` and explain the result. Type in `(5-1)*(1+2)**3` and explain the result. What is the *precedence of operators* in Python expressions (that is, which operators are applied first)?
8. Remove all redundant parentheses from the following Python expression:  
`(x - 2)**3 + (3*x)` ✓
9. Suppose we have defined a Python function

```
def reciprocal(x):
    return 1/x
```

What is the result of

```
>>> 1 + reciprocal(2*5)
```

How is a Python expression evaluated when one of the operands is a function call? How is a Python function evaluated when the argument is an expression?

- 10.▪ In Section 1.5, Question 1 you had a chance to experiment with Python's operator `%` applied to numbers. Determine its rank (precedence) as compared to `+`, `-`, `*`, `/`, and `**`.

## 2.4 Python Code Structure

Python wouldn't be much use if a programmer had to write every single statement separately. There must be a way to reuse a block of statements and execute this block multiple times. As we know, one way of doing this is to define a function. All programming languages let us define functions in one way or another.

### Example 1

```
# This function calculates 1 + 2 + ... + n
# using the formula sum = n(n+1)/2
def sum1ToN(n):
    'Returns 1 + 2 + ... + n'
    return n*(n+1)//2    # The // operator means
                        # integer division
```

The above function, familiar from Chapter 1, calculates the sum of all positive integers from 1 to  $n$ , using the formula derived in that chapter:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

Once a function is defined, you can call it with various arguments:

```
>>> sum1ToN(3)
6
>>> sum1ToN(6)
21
>>> sum1ToN(100)
5050
```

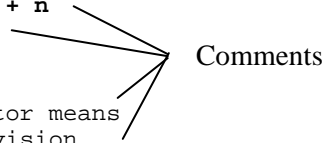


In Python, the # symbol, unless it is within a literal string, indicates a *comment*. The purpose of comments is to make programs more readable for humans. A comment can document what a function does or explain obscure code. The interpreter does not care: it simply skips all the text from # to the end of the line (Figure 2-5).

```

# This function calculates 1 + 2 + ... + n
# using the formula sum = n(n+1)/2
def sum1ToN(n):
    "Returns 1 + 2 + ... + n"
    return n*(n+1)//2 # The // operator means
                       # integer division

```



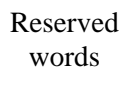
**Figure 2-5. Comments**

`def` is one of Python's *reserved words* (also known as *keywords*) — words that have special meanings in a programming language. Python uses about 30 reserved words. `def` indicates that we are defining a function. `return` is another reserved word (see Figure 2-6). When you enter Python code in *IDLE* (the GUI front end), different syntactic elements are displayed in different colors. The reserved words are orange by default.

```

# This function calculates 1 + 2 + ... + n
# using the formula sum = n(n+1)/2
def sum1ToN(n):
    "Returns 1 + 2 + ... + n"
    return n*(n+1)//2

```



**Figure 2-6. Reserved words**

**Python is case-sensitive. All reserved words, except `True`, `False`, and `None`, must be written in lowercase letters.**

**Note the colon at the end of the “`def`” line — it is required by the syntax rules.**

`sum1ToN` is the name we gave to our function, and `n` is the name we gave to its *argument* (input value).

**It is essential to give meaningful names to functions.**

A name in Python can consist of letters, digits, and underscore characters. A name cannot start with a digit. Python is case-sensitive, so `Sum` is different from `sum`. In our example, we could call our function `sumFrom1ToN` or `sum1_N`.

As you know, Python has several built-in functions with short names. These include `abs`, `id`, `input`, `max`, `min`, `pow`, `range`, `str`, `len`, and so on.

**Avoid using these names for your functions: if you do, your function will override the built-in function, and you won't be able to call that function.**

The lines that follow the “`def`” line are *indented* to the right. The indented lines form a block of related statements, in this case the definition of a function. Indentation must be consistent within a block — this is one of the Python syntax rules. It is customary to indent the next level by 4 spaces.

**When you write Python code in a text editor, use spaces rather than tabs for indenting lines.**

In addition to comments, it is customary to include a *documentation string* as the first line of a function, just below the `def` line (Figure 2-7). This string is optional. It is used to produce program documentation automatically.

```
# This function calculates 1 + 2 + ... + n
# using the formula sum = n(n+1)/2
def sum1ToN(n):
    "Returns 1 + 2 + ... + n" _____ Documentation
    return n*(n+1)/2 # The // operator means string
                   # integer division
```

**Figure 2-7. Documentation string**



**In Python, a simple statement is usually written on a separate line.**

One exception is literal strings enclosed within triple quotes `'''` or `"""`. Try:

```
>>> msg = '''And it always goes on
and on and on
and on and on'''
>>> msg
```

Python echoes the value of `msg`:

```
'And it always goes on\nand on and on\nand on and on'
```

`\n` in the message signifies the *newline* character. When encountered in the output text, `\n` shifts the next character position to the beginning of the next line. For example:

```
>>> print('And it always goes on\nand on and on\nand on and on')
And it always goes on
and on and on
and on and on
```

Or:

```
>>> print('Line 1\nLine 2')
Line 1
Line 2
```

Documentation strings often use the `""" ... """` or `'''...'''` if they exceed one line.

If a statement is too long to fit on one line, you can put a backslash `\` at the end of the line and continue on the next line. For example:

```
>>> 1 + 2 + 3 + 4 + 5 + 6 \
    + 7 + 8 + 9 + 10
55
```

However,

**a preferred style of writing an expression on multiple lines is to put it in parentheses.**

For example:

```
>>> (1 + 2 + 3 + 4 + 5 + 6
    + 7 + 8 + 9 + 10)
55
```

Python lets you place several statements on one line, although this is not common style. To do that, you have to separate the statements with semicolons. For example:

```
>>> x = 3; y = 2; print(x + y)
5
```

## Exercises

1. The following function returns the first character of a string (or the first element of a list):

```
def first(s):  
    return s[0]
```

Add a documentation string to this function and type all three lines at Python's prompt. Now try:

```
>>> first('Hello, world')
```

Then try

```
>>> first.__doc__
```

(Python uses two underscores on each side to mark special “system” names.)

2. Identify three reserved words in the function shown in Figure 2-2 on page 25. ✓
3. Identify two syntax errors in the following definition of a function:

```
def badCode(x)  
    Return x**2 - 1
```

4. What is the output from the following statement? ✓

```
print('One is better than \none' +\  
      '; two is better than one')
```

5. What is the output from the following statement?

```
print('Python is #1')
```

6. Find a syntax error in the following code: ✓

```
def mystery(n):  
    'Returns n cubed'  
    return n**3
```

7. ■ In Python, `s[-1]` refers to the last character of a string `s` (or the last element of a list `s`). Write a function that takes a string `s` and returns a new string that consists of two characters: the first and the last characters in `s`. Give your function a reasonable name. Include a documentation string in your code.
8. ■ Write a Python function `triangle(s)` that has only one `print` statement and prints

```
sssss
 sss
  s
```

where `s` is a one-character string. For example, `triangle('*')` should print

```
*****
 ***
  *
```

Can you omit the `return` statement in a function definition? If so, what will your function return?

## 2.5 Review

Terms introduced in this chapter:

<i>CPU</i>	<i>Programming language</i>
<i>RAM</i>	<i>Assembly Language</i>
<i>Bit</i>	<i>Syntax rules</i>
<i>Byte</i>	<i>Redundancy</i>
<i>Kilobyte</i>	<i>Parsing</i>
<i>Megabyte</i>	<i>Prompt</i>
<i>Gigabyte</i>	<i>Comment</i>
<i>Gigahertz</i>	<i>Indentation</i>
<i>Compiler</i>	<i>Reserved word (keyword)</i>
<i>Interpreter</i>	<i>Literal string</i>
<i>Source code</i>	<i>Function argument</i>
<i>Debugging</i>	

Some of the Python features introduced in this chapter:

```
def someFunction(x):  
    ...  
    ...  
    return ...
```

A block of related statements is indented.

+, \*, /, \*\*, % operators

Literal strings: 'abc', "abc", '''abc''', or """abc"""

\n stands for the newline character

\ at the end of the line means continue on the next line

# marks a comment that extends to the end of the line.