

Kenneth Thompson and **Dennis Ritchie** developed the highly successful **UNIX** operating system. In addition, Dennis Ritchie developed the **C** programming language evolved from the B programming language developed by Thompson.

12.8.4 The Development of Graphical User Interfaces (early 1960s)

Another major advance in operating systems design was the move from **text-based user interfaces** (in which all commands were typed) to **graphical user interfaces** (GUIs). A GUI interface provides the familiar computer user interaction by use of a mouse. It might surprise you that the **computer mouse** was invented and first demonstrated back in the early 1960s, by a professor at Stanford University named **Doug Engelbart** (Figure 12-53). He is shown here holding a prototype of the mouse, constructed out of a wood shell and two metal wheels. Soon afterward he demonstrated the first word processor and the first hypertext, the kind of links that now exist in web pages. He has received numerous awards, including the Lemelson-MIT prize—the world's largest single prize for innovation—and the National Medal of Technology, the nation's highest technology award.

Doug Engelbart invented the computer mouse in the early 1960s.

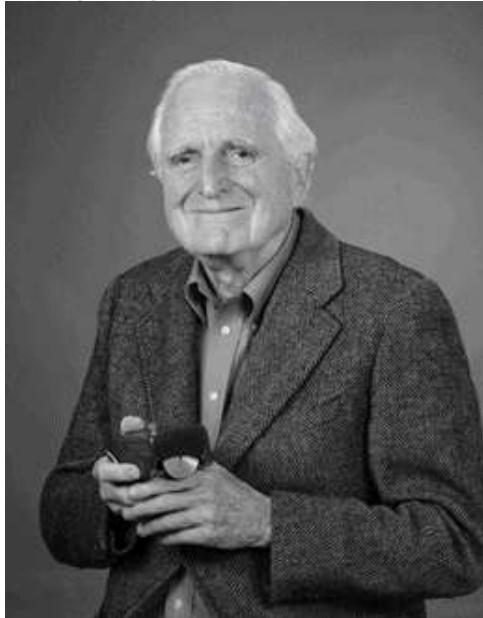


FIGURE 12-53 Doug Engelbart— Inventor of the Computer Mouse

12.9 The Rise of the Microprocessor 515

12.8.5 The Development of the Supercomputer (1972)



FIGURE 12-54 Seymour Cray— “The Father of the Supercomputer”

A supercomputer, as the name implies, is a computer system designed for maximum computational power. Whereas mainframe computers generally timeshare computations over a number of users, supercomputers focus their power on executing a few programs as quickly as possible. They are used in solving problems that require massive numbers of mathematical calculations, such as for weather forecasting.

In 1972, Seymour Cray (Figure 12-54), the “Father of Supercomputing,” started a company called Cray Research. The CRAY-1, released in 1976, became one of the most successful supercomputers in history, setting a new standard for high-speed computing (Figure 12-55).

What is classified as a supercomputer is relative to current technological capabilities. A typical personal computer today would have been considered a “supercomputer” in the not so distant past. There is a website that each

year lists the top 500 supercomputers worldwide (see www.top500.org). These supercomputers have demonstrated computational speeds measured in “teraflops,” i.e., trillions of floating-point operations per second. The fastest supercomputer as of this writing is the Cray-XT5 HE, dubbed the “jaguar,” with a speed of 1.75 petaflops (almost 2 quadrillion floating-point operations per second), with 224,000 individual processors. It is installed at the National Center

for Computational Sciences at Oak

Ridge National Laboratories in Tennessee.



FIGURE 12-55 The CRAY-1

Seymour Cray is called the “**Father of the Supercomputer.**”

FOURTH-GENERATION COMPUTERS

(early 1970s to the Present)

12.9 The Rise of the Microprocessor

12.9.1 The First Commercially Available Microprocessor (1971)

The development of the personal computer (originally “microcomputer”) has been one of the biggest technical revolutions in history. It didn’t emerge from the research labs of major companies and universities, but rather from a group of hobbyists interested in building and owning their own computers.

Before the development of the microprocessor in the early 1970s, it was not possible for individuals to build their own computer system. A central processing unit (CPU) was too technically sophisticated for hobbyists to build. The first microprocessor, the 4-bit 4004 released by Intel in 1971, was only powerful enough for handheld calculators. But in 1974, Intel came out with the 8080 processor (Figure 12-56). The 8080 was an 8-bit processor with a clock speed of 2 MHz (thousands of times slower than a typical CPU in personal computers today) and contained about 6,000 transistors (compared to billions of transistors today). It was powerful enough, however, to build a computer system around. Thus, the age of the personal computer was about to begin. We look at the development of the personal computer next.

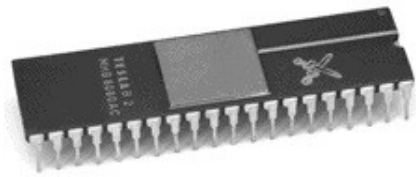


FIGURE 12-56 The Intel 8080 Microprocessor

The **first commercially available microprocessor** was the **Intel 4004**, a four-bit processor released in 1971.

12.9.2 The First Commercially Available Microcomputer Kit (1975)

The first commercially available microcomputer was a kit (also offered fully assembled) called the **Altair 8800** (Figure 12-57), first advertised in January 1975 on the cover of *Popular Electronics* magazine. It consisted of a box with levers and lights, used to input and display information in binary notation (i.e., light on for “1,” light off for “0”). This was, especially by today’s standards, a very crude and simple computer. However, given that no one until then could say that they had their own “personal computer,” it was a

thrill for many hobbyists.



FIGURE 12-57 The Altair 8800—The First

Microcomputer Kit

The **Altair** was the **first available microcomputer** (personal computer), appearing in 1975.

12.10 The Dawn of Personal Computing

12.10.1 The Beginnings of Microsoft (1975)

In 1975, the Homebrew Computer Club began in the San Francisco area. Among the soon to be infamous members were **Paul Allen** and **Bill Gates** (Figure 12-58), who created a version of the BASIC programming language for the new Altair system, and comprised the beginnings of Microsoft. Also present at the club was **Gary Kildall**, who wrote the first programming language and disk

operating system for microcomputers, among many other contributions.

In the mid-1980s, Microsoft produced the MS-DOS operating system for personal computers, which dominated the market.

MS-DOS was a text-based operating system.



FIGURE 12-58 Bill Gates and

Paul Allen

In the 1990s, Microsoft came out with the GUI-based Windows operating system, which remains the most widely used operating system today.

In 1975, **Bill Gates** and **Paul Allen** started the Microsoft Corporation. **Gary Kildall** wrote the first programming language and disk operating system specifically for microcomputers.

12.10.2 The Apple II (1977)



FIGURE 12-59 Steve Jobs and Steve

Wozniak

Two other members of the Homebrew Computer Club were **Steve Jobs** and **Steve Wozniak** (Figure 12-59). In 1977 they went on to create the most successful personal computer at the time, the **Apple II** (Figure 12-60) and form the Apple Computer company. One of the Apple II's main features was the ability to display color graphics, displaying output on a standard television

monitor. Various models of the Apple II were produced until 1993.



FIGURE 12-60 The Apple II

In 1977, **Steve Jobs** and **Steve Wozniak** created the most successful personal computer at the time, the **Apple II**.

12.10.3 IBM's Entry into the Microcomputer Market (1981)

IBM, which first had doubts about the future of microcomputers, saw the success of the Apple II computer and decided to develop their own microcomputer. In 1981, they came out with the **IBM-PC** (for "Personal Computer"), and before long, had sold 1 million machines (Figure 12-61). By the end of the 1980s, 65 million PCs were being used in offices, homes, and schools. Because of their size, reputation, and sales force, IBM dominated the personal computer market. Eventually, the generic name for microcomputers became "personal computer."



FIGURE 12-61 IBM-PC in 1981

In 1981, IBM came out with the IBM-PC ("personal computer"). By the end of the 1980s, 65 million PCs were in use.

12.10.4 Society Embraces the Personal Computer (1983)

In a very unconventional selection, *Time Magazine* selected the personal

computer for its January 3, 1983 “Man (Machine) of the Year” cover article. In the article, it said that by the millions, computers were finding their way into offices, schools, and homes. In addition, it discussed a survey showing that the average American believed that “in the near future,” computers would be as commonplace as TVs and refrigerators. The public image of what computers were had made a big change. The article stated (even in 1983) that the 30-ton ENIAC computer “could now fit on a chip the size of a pea.” (In fact, as part of the fiftieth anniversary of the ENIAC at the place of its birth, the University of Pennsylvania, a group of students did just that. They made an exact logical replication of the original ENIAC on a chip about 1/3” by 1/5” in size.)

In 1983, given the widespread use and impact of the personal computer, Time Magazine selects the **personal computer as the “Man” of the Year**.

12.10.5 The Development of Graphical User Interfaces (GUIs) An Early Machine Using a Mouse Driven Graphical User Interface (1975)

Because computers at the time were not powerful enough to support the computationally intensive graphical interactive method of computing (as demonstrated by Doug Engelbart), the idea did not initially catch on. However, in 1975, a computer was developed at Xerox Palo Alto Research Center (Xerox-PARC) that incorporated the use of a mouse device into a rather novel computer called the **Alto**. Its screen was the dimension of a normal sheet of paper, and had a graphical user interface. Although this was a notable advancement by those working at Xerox research center, it was never marketed by Xerox, leaving many of the developers feeling frustrated. Some years later, however, in 1979, Steve Jobs visited Xerox-PARC, and after seeing the Alto and its interactive mouse, immediately knew that the future of computing was about to change.

In 1975, a computer was developed at Xerox Palo Alto Research Center (Xerox-PARC) that incorporated the use of a mouse device into a rather novel computer called the **Alto**.

The First Commercially Successful Computer with GUI/Mouse (1984)

In January 1984, Apple came out with the first commercially successful computer with a graphical user interface, the **Macintosh** (Figure 12-62). In 1983, Apple had come out with another mouse/GUI system named Lisa aimed at the business market, its high selling price kept it from being commercially successful. The Macintosh, on the other hand, was priced for individuals, and

highly marketed. In one of the most famous commercials in the history of television during the 1984 Super Bowl game, Apple created an image of a new wave of computing, awaking large groups of robotlike, hypnotized individuals, meant to represent the awakening of the “blind” followers of IBM (which had not yet come out with a GUI-based machine).



FIGURE 12-62 Apple Macintosh in 1984

In January 1984, Apple came out with the first commercially successful computer with a graphical user interface, the **Macintosh**.
Mouse-Driven GUI Operating Systems Predominate (1995)

Microsoft began releasing its GUI interface operating systems, Windows, in 1985 with the release of Windows 1.0. However, its Windows operating system for the IBM-PC (and clones) did not begin to catch on until the release of Windows 3.0 in 1990, some waiting until the release of Window 3.1 in 1993. However, when the much promoted and completely redesigned Windows 95 was released in August 1995, within six weeks 7 million copies had been sold. The GUI-based computing paradigm of computing had become mainstream.

With the release of the **Microsoft Windows** operating system the GUI-based computing paradigm of computing had become mainstream.

12.10.6 The Development of the C 11 Programming Language

We take the opportunity here to mention one of the most widely used programming languages today, C 11 (pronounced “cee plus plus”). The other widely used programming language is Java (discussed in section 12.13.4).

C 11 was developed by Bjarne Stroustrup in the early 1980s at Bell Labs (Figure 12-63). It was designed essentially as an extension of the C programming language, also developed at Bell Labs over a decade earlier. The most significant feature added to C 11 is the ability to use classes, therefore facilitating object-oriented programming. However, as opposed to most object-oriented

programming languages, such as Java, the use of classes is not required. Therefore, C 11 program can be written strictly in the procedural style. In this way C 11 is a hybrid language. There is some playfulness in the name of the language. The operator symbol “ 11” that exists in C (and C 11)

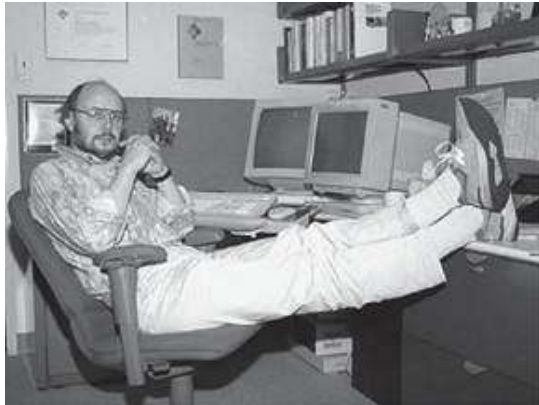


FIGURE 12-63 Bjarne Stroustrup— Inventor of the C11 Programming Language

increments a value by 1. Since Stroustrup’s new language was “an increment” of C, the name C 11 was chosen.

Bjarne Stroustrup invented C11 at Bell Labs in the early 1980s as an extension of the C programming language. C11 has become one of the most widely used programming languages today.

THE DEVELOPMENT OF COMPUTER NETWORKS

12.11 The Development of Wide Area Networks

12.11.1 The Idea of Packet-Switched Networks (early 1960s)

During the early 1960s, **Leonard Kleinrock** (Figure 12-64) at MIT published work on the notion of a **packet-switched network**, in which communications to be sent on a network are divided into equal-sized packets and transmitted individually. When all packets are received at the destination, they are then reassembled into the original complete communication. This is how all communication on the Internet is done. (The first experiment of having two computers—one in Massachusetts, the other in California—communicate over standard (non-packet-switched) telephone lines failed, proving Kleinrock’s claim of the need for a packetswitched approach correct.) Because of the distance

covered by these networks, they became known as **wide-area networks (WANs)**.



FIGURE 12-64 Leonard Kleinrock—Inventor of Internet Technology

Leonard Kleinrock invented the notion of a packet-switched network, which is how all communication on the Internet is performed.

12.11.2 The First Packet-Switched Network: ARPANET (1969)

What was needed next for wide-area computer networks was a standard “language” or communications protocol that all computers on a network could understand. By the end of 1968, a number of individuals working on packet-switched communications received funding from a military defense department, resulting in a network called **ARPANET**, becoming the first packet-switched network.

In September 1969, the first Arpanet node was installed at the University of California at Los Angeles (UCLA). Another node was installed at Stanford, and in October 1969, the first “host to host” message was sent directly from one computer to another—the beginnings of what we now know as the Internet. Over the years, more universities and companies were added to the network. In October 1972, the first public demonstration of **email** communication occurred. For the next ten years, email became the most used aspect of the new ARPANET.

ARPANET was the first packet-switched network, becoming operational in

1969. The first public demonstration of **email** was in 1972.

12.12 The Development of Local Area Networks (LANs) 521

12.12 The Development of Local Area Networks (LANs)

12.12.1 The Need for Local Area Networks

The term “wide-area network” is meant to convey communicating computers over a wide area (e.g., globally). This concept first emerged during the era of mainframe computers (in the 1960s). However, computer networking had new emerging needs with the rise in the use of minicomputers during the 1970s, and personal computers during the 1980s.

Before the use of personal computers, for example, employees in a company could communicate and share files through remote terminal screens to the company’s mainframe computer. However, during the 1980s, when the personal computer began appearing on everyone’s desk, something was lost. Now each employee had all the computing power that they needed right at hand, but their machine was isolated from all others in the company, each in their own “computing island.”

Computer networking had new emerging needs with the rise in the use of minicomputers during the 1970s, and personal computers during the 1980s.

12.12.2 The Development of Ethernet (1980)

Robert Metcalf (Figure 12-65) was working as a researcher for the Xerox Corporation in the early 1970s when he was asked to work on a particular problem. Xerox was in the process of developing the world’s first laser printer, and wanted all the computers (hundreds of them) at the Xerox-PARC research facility where he worked to be connected to it. This problem called for a means of communication that was *fast* and allowed *many communications at once* from the hundreds of computers in the building.

His solution was the development of **local area networks (LANs)**. Local area networks are defined as a network of computers and computing devices (such as printers) within about a one-mile radius. As opposed to wide-area networks, in which the topology (i.e., the connections between computers) allows messages to be routed many different ways, Metcalf developed a network based on a simple topology. His topology consisted of a single, high-speed connection that all computing devices could share, called **Ethernet**. Ethernet was made

commercially available in 1980, and is now the most widely used local area network standard. Robert Metcalfe was awarded the National Medal of Technology from President George W. Bush for his work in 2003.



FIGURE 12-65 Robert Metcalfe—Inventor of Local Area Networks

Robert Metcalfe invented the concept of a local area network, which led to the development of Ethernet, the most widely used local networking standard today.

12.13 The Development of the Internet and World Wide Web

12.13.1 The Realization of the Need for “Internetworking”

The development of ARPANET proved the viability of packet-switched computer networking. However, it had its own specific protocol (i.e., specific means of communication between computers). The future was viewed, however, as developing into a number of different kinds of packet-switched networks, each with their own unique protocols. The idea of “internetworking” was the ability of individual networks, normally unable to communicate with each other, to be able to communicate using an “internetworking protocol” (IP), resulting in one big computer network.

Internetworking is the ability of individual networks, normally unable to communicate, to communicate using an “internetworking protocol” (IP).

12.13.2 The Development of the TCP/IP Internetworking Protocol (1973)

A research program was initiated in 1973 to investigate ways of internetworking different packet-switched networks. The standard internetworking protocol that evolved was called TCP/IP. Since packets of a given communication travel independently in a packet-switched network, TCP/IP had the responsibility of ensuring that each packet arrived at its destination intact and reassembled into the complete original communication. The development of the TCP/IP protocol was developed by **Vinton G. Cerf** and **Robert E. Kahn** (Figure 12-66) referred to as the “Fathers of the Internet.” In 2005 they each received the Medal of Freedom from President George W. Bush for their work.

On January 1 1983, TCP/IP became the standard protocol for the internetworking of networks. On that date, all computers currently connected to the Internet (using older protocols), smoothly switched to the new protocol simultaneously, and thus was the birth of *the Internet* as we know it today. The Internet is now comprised of a collection of over 50,000 independent networks, on all seven continents, and even planned for use in outer space.

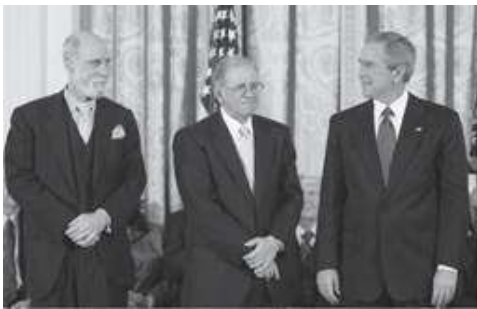


FIGURE 12-66 Vinton G. Cerf and Robert E. Kahn—“The Fathers of the Internet”

Vinton G. Cerf and **Robert E. Kahn** developed a common network protocol called **TCP/IP** that allowed for the internetworking of computer networks. Thus, they are referred to as the “Fathers of the Internet.”

12.13.3 The Development of the World Wide Web (1990)

The idea of having text that can be clicked on to lead one to more text has goes all the way back to a system developed in 1945, providing links between documents on microfiche. Doug Engelbart in the early 1960s demonstrated the notion of “mouse-clickable” text. Later, the term “hypertext” was coined.

Hypertext documents are essentially “three-dimensional text,” in that there are

words “behind” other words when clicked on.



FIGURE 12-67 Tim Berners-Lee—“Father of the World Wide Web”

12.13 The Development of the Internet and World Wide Web 523

Tim Berners-Lee (Figure 12-67), while working at CERN in 1980 (a European center for nuclear research outside of Geneva, Switzerland), developed a simple program with hypertext properties. He later revived his idea and in 1990 developed a program he named “WorldWideWeb,” the first web browser and hypertext editor. He is thus known as the “Father of the World Wide Web.”

A **web browser** is an application program to properly display hypertext (web) pages, performing the necessary actions of mouse events (e.g., mouse click on a link). In May 1991, the WideWorldWeb program was released for use at CERN for the creation, searching, and cross-referencing of research

papers in theoretical physics.

In 1993, a web browser was developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois named **Mosaic**, developed by **Marc Andreessen** (Figure 12-68). This was the first browser readily available to the general public, credited with popularizing the World Wide Web. Andreessen later went on to form Netscape Corporation, developing the Netscape family of browsers.



FIGURE 12-68 Marc Andreessen—Developed the First Readily Available Web Browser

Tim Berners-Lee was the inventor of the World Wide Web. **Marc Andreessen** developed Mosaic, the first web browser readily available to the general public. He went on to form Netscape Corporation.

12.13.4 The Development of the Java Programming Language (1995)

The development of the World Wide Web (and Internet browsers such as Netscape) was a very significant advance for the sharing of information. However, the information was static. One could interact with the browser by clicking on a link of one page of (static) information to retrieve another page. However, the web pages themselves did not have much capability of interacting with the user. For example, a web page offering a mortgage calculation would get the input from the user, then send it back to the server for calculations to occur there, with the results downloaded in another static page. Thus, there was not a way to download within a web page programs capability of performing significant computation on the user's computer (the "client").

As a matter of coincidence, a language perfectly suited for embedding programs in web pages emerged at the same time as the web. This was the **Java programming language**, was developed by James Gosling (Figure 12-69) at Sun Microsystems during the early 1990s.



FIGURE 12-69 James Gosling—Inventor of the Java Programming Language

A critical feature needed for the development of dynamic web pages was that the language used had to be executable (“understood”) by all the different kinds of computers on the Internet if the capability were to be available to all. The historical coincidence that occurred was that in 1990, Sun Microsystems began work on a new programming language written with the same capability of being able to be “compiled once, run anywhere.” The need was related to software development for embedded systems, that is, systems containing an embedded processor, such as microwave ovens. With the new World Wide Web, it was realized the features of Java, developed for another purpose, were fundamentally suited for dynamic web pages. Java was expanded to include features especially suited for network applications, and was first released to the public in 1995, just a couple of years after the “web” became publically and freely available.

James Gosling invented the Java programming language at Sun Microsystems in the early 1990s. It has become one of the most widely used programming languages today.

Python 3 Programmers **APPENDIX** Reference

The author gratefully acknowledges the contributions of Leela Sedaghat to the style and content of this reference.

This reference provides the features of Python 3 that are most relevant for the text. Therefore, it is not intended to be an exhaustive resource. For complete coverage of the Python programming language, Standard Library and other Python-related information, we refer readers to the official web site:

<http://www.python.org>

A. Getting Started with Python 528

A1. About Python 528

A2. Downloading and Installing Python 528

A3. Program Development Using IDLE 529

A4. Common Python Programming Errors 539

B. Python Quick Reference 540

B1. Python Coding Style 541

B2. Python Naming Conventions 542

B3. Comment Statements in Python 543

B4. Literal Values in Python 544

B5. Arithmetic, Relational, and Boolean Operators in Python 545 B6. Built-in Types and Functions in Python 546

B7. Standard Input and Output in Python 549

B8. General Sequence Operations in Python 550

B9. String Operations in Python 551

B10. String Formatting in Python 552

B11. Lists in Python 553

B12. Dictionaries in Python 554

B13. Sets in Python 555

525

B14. if Statements in Python 556 B15. for Statements in Python 557 B16. while Statements in Python 558 B17. Functions in Python 559

B18. Classes in Python 560

B19. Objects in Python 561

B20. Exception Handling in Python 562 B21. Text Files in Python 563

B22. Modules in Python 564

C. Python Standard Library Modules 565 C1. The math Module 566

C2. The random Module 567

C3. The turtle Module 568

C4. The webbrowser Module 570

A. GETTING STARTED WITH PYTHON

A1. About Python

The Python programming language was created by Guido van Rossum (www.python.org/~guido) at the Centrum Wiskunde & Informatica (National Research Institute for Mathematics and Computer Science) in the Netherlands in the late 1980s. It is designed for *code readability*. It therefore has a clear and simple syntax. At the same time, Python also has a powerful set of programming features.

Python is free, open source software (<http://www.python.org>). The reference (standard) implementation of the Python programming language (called CPython) is managed by the non-profit Python Software Foundation. The language is bundled with a Python development environment called IDLE. The bundle is available for download at the official Python web site (see below). There are two, incompatible versions of Python currently supported: Python 2 (2.7.3) and Python 3 (3.2.3 at the time of this writing). Python 2.7.3 will be the last release version of Python 2. This text uses Python 3.

Python is growing in popularity. Many companies and organizations use Python including Google, Yahoo and YouTube. Python is also widely used in the scientific community, including the National Weather Service, Los Alamos National Laboratory, and NASA. Python also continues to gain popularity for use in introductory computer science courses.

A2. Downloading and Installing Python

To download and install the Python interpreter (and bundled IDLE program) go to <http://www.python.org/download/> which displays the page shown in Figure A-1.

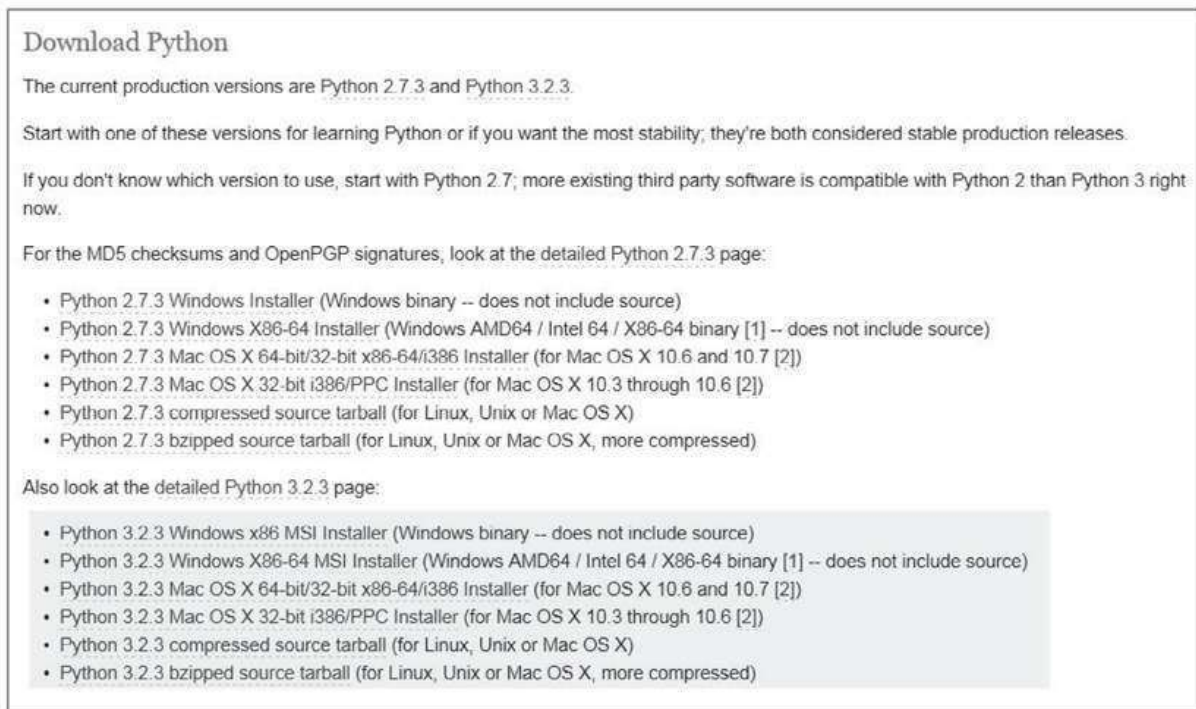


FIGURE A-1 Official Download Site of Python

Select the appropriate Python 3 download for your system: Windows x86 MSI installer (for typical Windows machines), Windows x86-64 (for 64-bit Windows machines, not typical), Mac OS X 64-bit/32-bit x86-64/i386 (for newer Macs) and Mac OS X 32-bit i386/PPC (for older Macs). There are also versions for Linux and Unix, as shown. Follow the installation directions.

A3. Program Development Using IDLE

What is IDLE? IDLE is an *integrated development environment* (IDE) for developing Python programs. An IDE consists of three major components: an **editor** for creating and modifying programs, an **interpreter** or **compiler** for executing programs, and a **debugger** for “debugging” (fixing errors in) a program.

When you execute IDLE on your system, a window such as shown in Figure A-2 will be displayed.

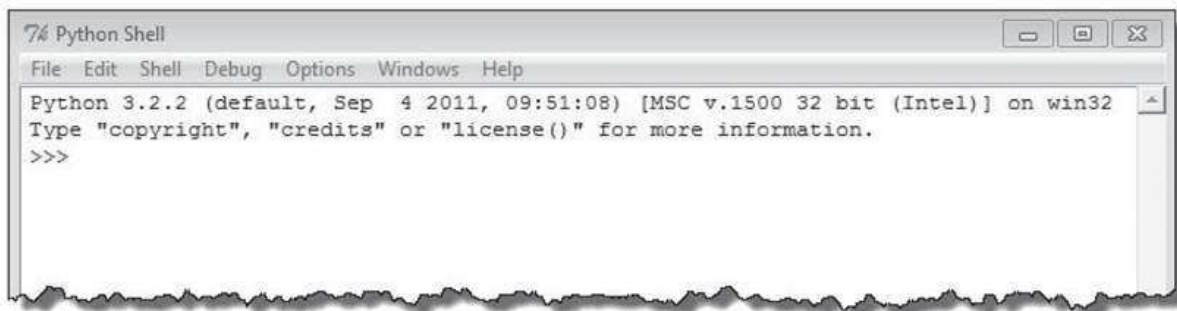


FIGURE A-2 The Python Shell

Note that the version of Python is displayed on the lines right before the prompt (...).

Interacting with the Python Shell The Python Shell provides a means of directly interacting with the Python interpreter.

Thus, whatever Python code that is typed in will be immediately executed, as shown in Figure A-3.

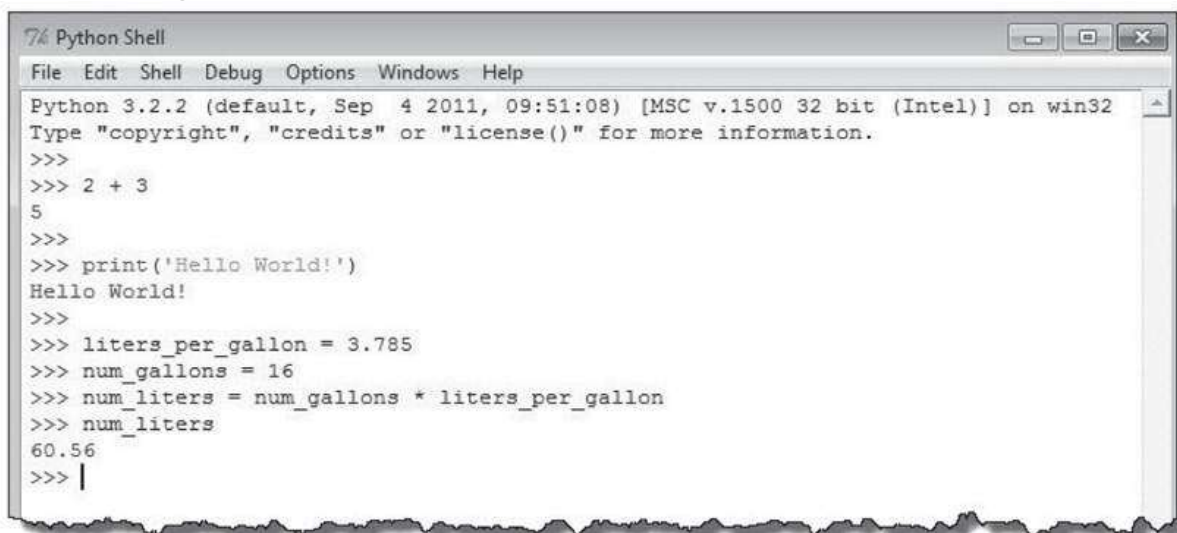


FIGURE A-3 Interacting with the Python Shell

This is referred to in programming as *interactive mode*. All variables will remain defined until the shell is either closed or restarted. To restart the shell, select Restart Shell from the Shell dropdown list, as shown in Figure A-4.

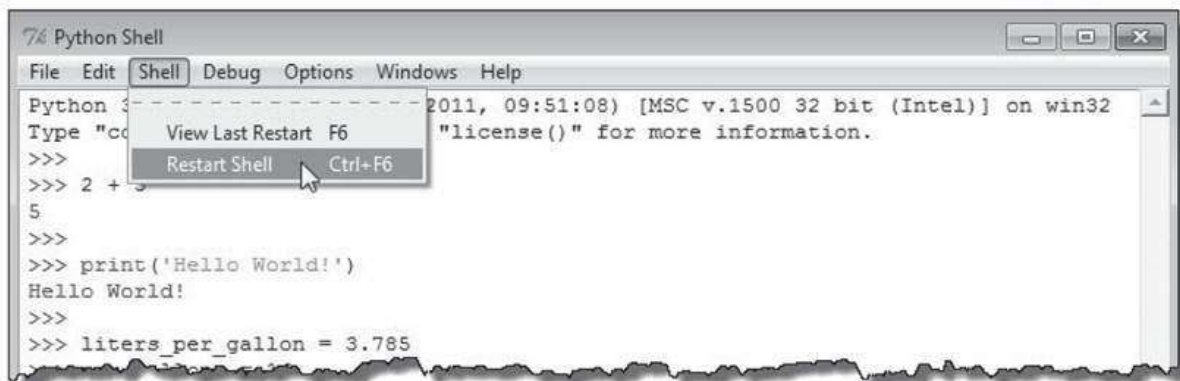


FIGURE A-4 Restarting the Python Shell

After the shell is restarted, all previously-defined variables become undefined and a “fresh” instance of the shell is executed, as shown in Figure A-5.

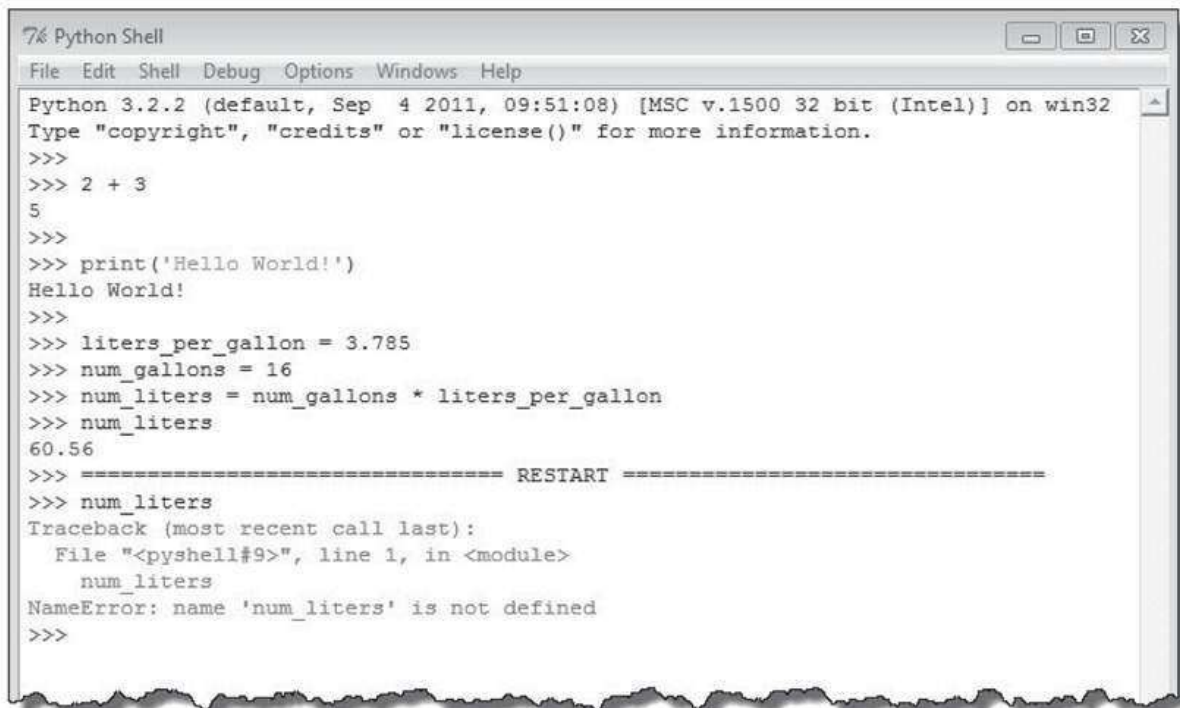
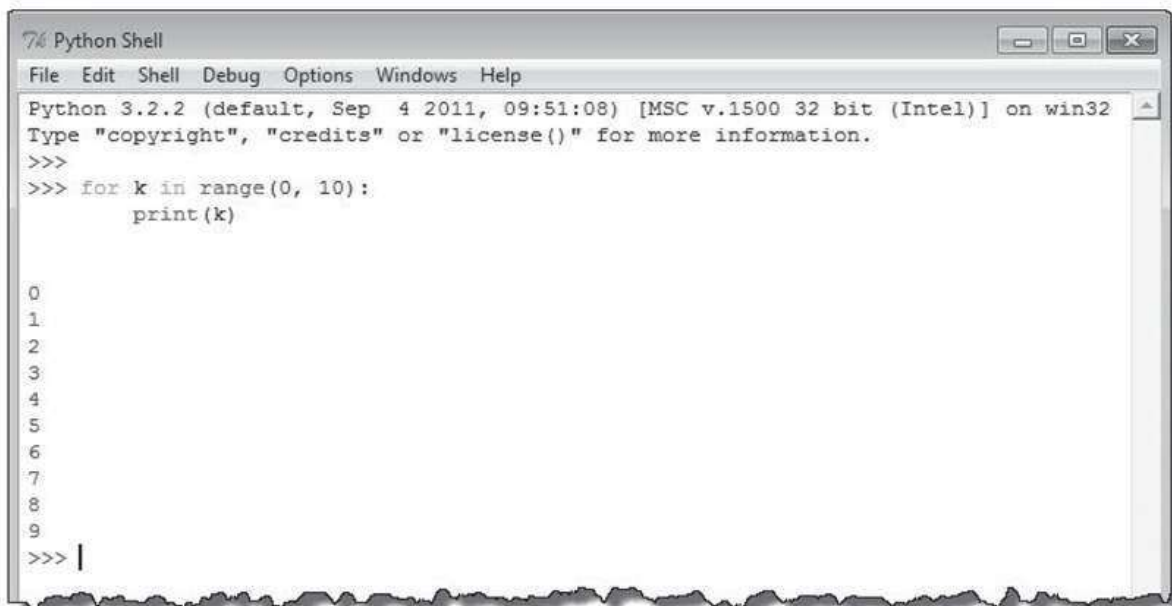


FIGURE A-5 A New Instance of the Python Shell

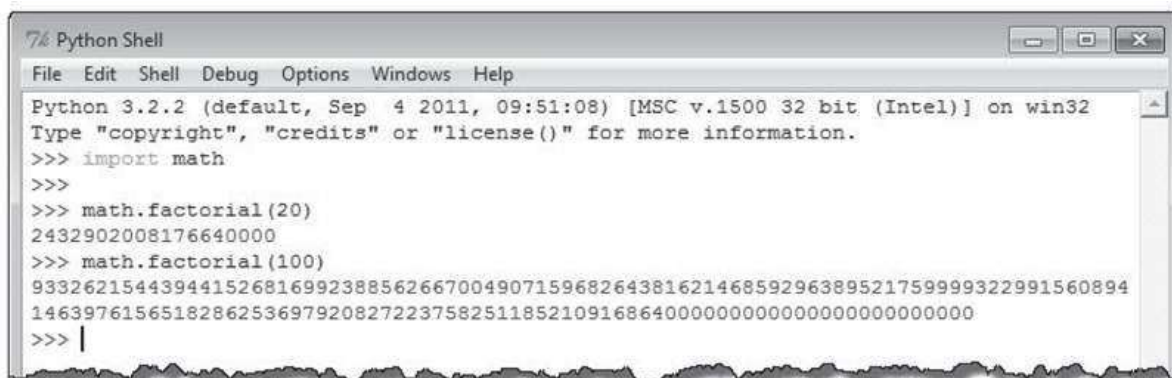
Being able to immediately execute instructions in the Python Shell provides a simple means of verifying the behavior of instructions in Python. For example, if one forgot the specific range of numbers that built-in function `range(start, end)` produces for a given start and end value, the single for statement in Figure A-6 can be easily executed to determine that.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep 4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> for k in range(0, 10):
    print(k)
0
1
2
3
4
5
6
7
8
9
>>> |
```

FIGURE A-6 Testing Small Program Segments in the Python Shell

We can also execute Python code in the shell that requires the use of modules. To utilize a particular module in the shell, the appropriate import statement is first entered.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.2 (default, Sep 4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>>
>>> math.factorial(20)
2432902008176640000
>>> math.factorial(100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991560894
1463976156518286253697920827223758251185210916864000000000000000000000
>>> |
```

FIGURE A-7 Importing a Module into the Python Shell

Finally, when working in the interactive Python Shell, there are helpful keyboard shortcuts that the user may need to use. Some of the most commonly used commands are listed in Figure A-8.

Windows Shortcut	Mac Shortcut	Action Performed
Ctrl+C	Ctrl+C	Terminates executing program
Alt+P	Ctrl+P	Retrieves previous command(s)
Alt+N	Ctrl+N	Retrieves next command(s)

FIGURE A-8 Python Shell Shortcuts

Creating and Editing Programs in IDLE Instructions typed into the shell are executed and then “discarded.” For program development, however, we need to save instructions into a file. This is referred to as a **script** in Python. A program in Python is a script. To begin creating a script in IDLE, select New Window from the File dropdown menu, as shown in Figure A-9.

A Python program (or other text files) can be typed in this window. This program is saved by going to the File menu option and selecting Save (or by using shortcut key Ctrl+S), depicted in Figure A-10.

When selecting the Save option on a currently unnamed file, a file window appears allowing a name to be entered for the new program (Figure A-11). If the program was already named, choosing Save would save the file under the current name, overwriting the currently saved file. Using Save As allows a file to be named under a different name, such as for creating a backup file.

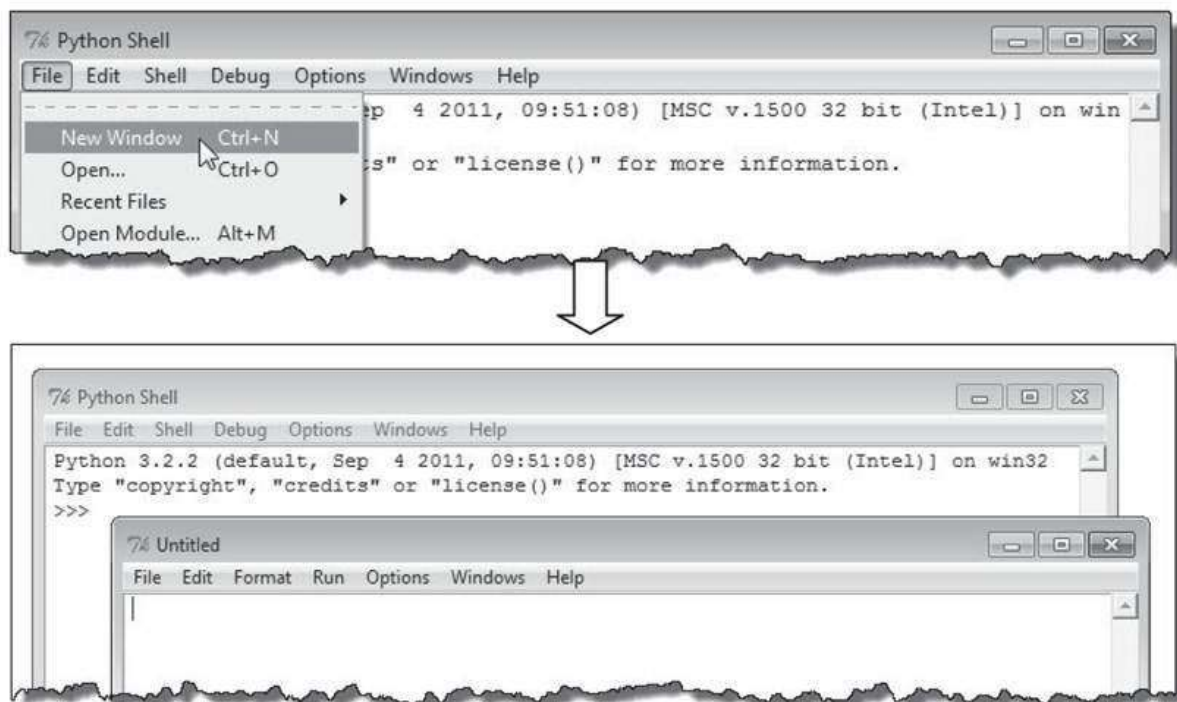


FIGURE A-9 Opening a Script Edit or Window

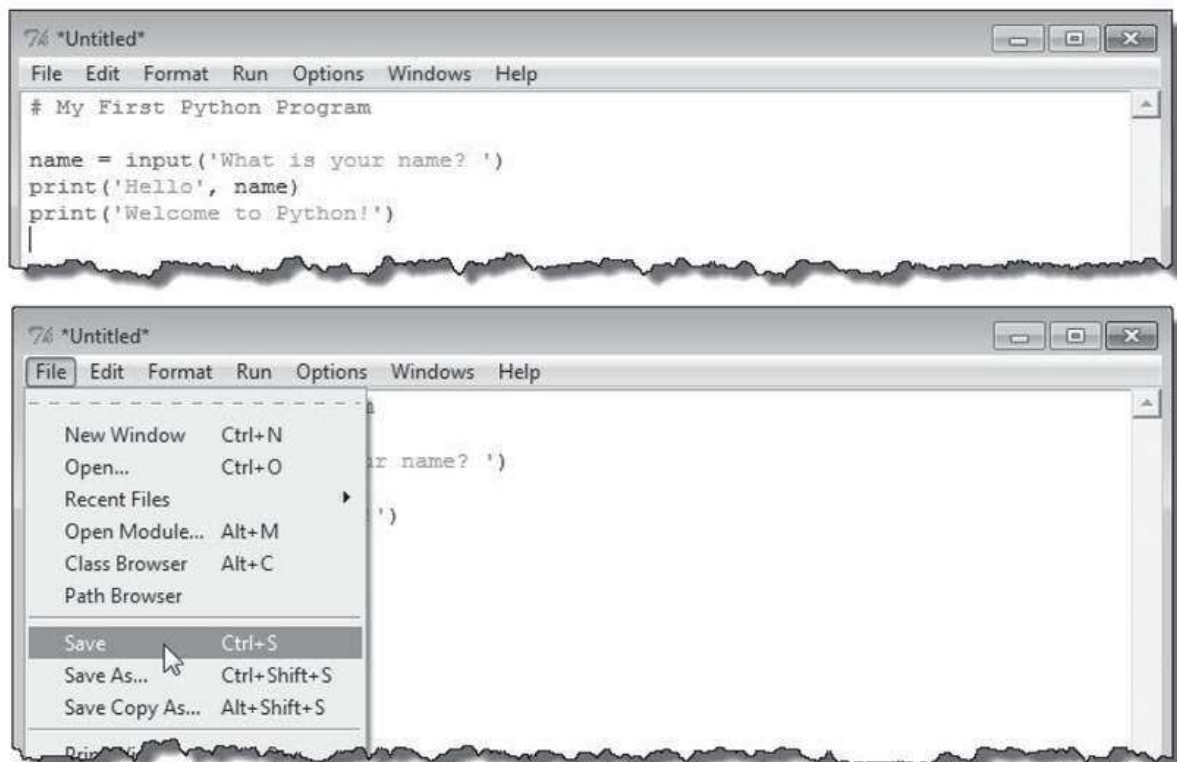


FIGURE A-10 Entering and Saving a Python Program File

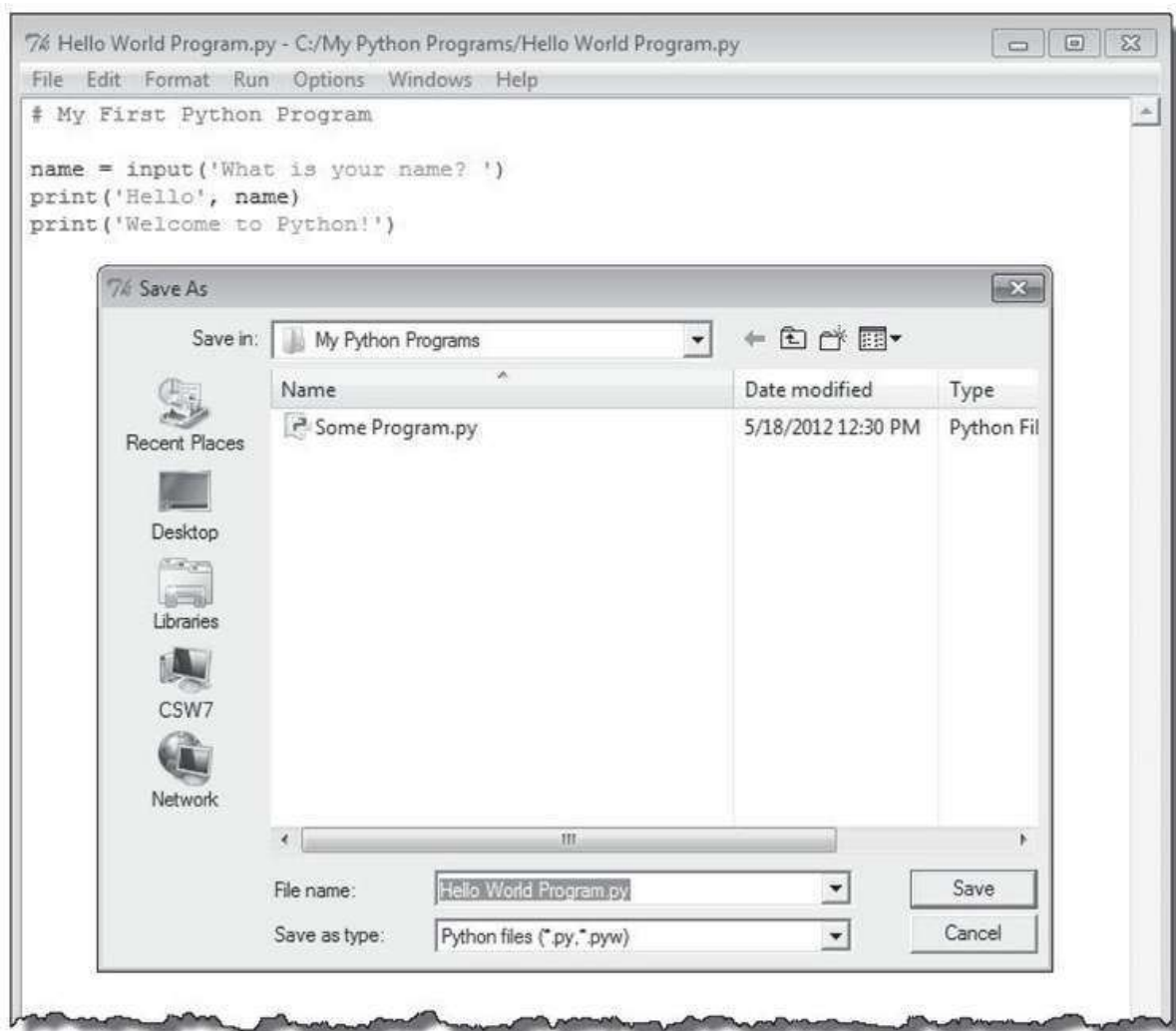


FIGURE A-11 The File Window and Naming and Saving a File

When saving a Python program, *remember to add the file extension .py to the filename*. Otherwise, the saved file will not appear in the file window when working with Python programs in IDLE. It also will not be considered a Python file by the operating system. In this case, Hello World Program.py will become a new file in the My Python Programs folder in addition to the existing file Some Program.py.

When creating and modifying programs, there are a number of editing features that IDLE provides. Many of these commands will likely be familiar to you. However, in addition to the familiar commands (such as “cut” and “paste”) there are editing features in IDLE that are specific to Python. These commands are listed in Figure A-12.

The backspace is for deleting characters *before* the cursor, and delete deletes characters *after*. Lines may be deleted, copied and pasted within a program. For navigating a program file, there is the ability to go to a specific line number, or search for lines containing a specific search string. Once a search has begun, the shortcut Ctrl 1G provides a convenient way to continue the search to each next line found.

Windows Shortcut	Mac Shortcut	Action Performed
Backspace Key	Delete	Erases character before cursor
Delete Key	-	Erases character after cursor
Ctrl+C	⌘+C	Copies highlighted set of lines to be pasted
Ctrl+X	⌘+X	Cuts highlighted set of lines to be pasted
Ctrl+V	⌘+V	Pastes most recently copied or cut highlighted lines
Alt+G	⌘+J	Prompts for line number to place cursor
Ctrl+F	⌘+F	Prompts for characters to search for
Ctrl+G	⌘+G	Continues current search
Ctrl+0	Ctrl+0	Shows the innermost matching parentheses
Ctrl+Z	⌘+Z	Undo all previous editing actions one-by-one
Ctrl+]	⌘+]	Indents selected lines
Ctrl+[⌘+[Unindents selected lines back to the left
Alt+3	Ctrl+3	Comments out selected lines
Alt+4	Ctrl+4	Removes ## notations from start of selected lines
Ctrl+S	⌘+S	Saves program as current file name

FIGURE A-12 Editing Commands in IDLE

The Show Matching command (Ctrl 10) identifies certain matching delimiters. The innermost set of matching parentheses, square brackets or curly braces are highlighted from the current position of the cursor. This is useful for identifying mismatched parentheses in expressions, as well as mismatched delimiters in data structures containing parentheses, square brackets and/or curly braces. The Undo command (Ctrl 1Z) will undo the most recent edit change in a file. This can be used to undo all the edit changes, even before the last time the file was saved, way back to the state of the file when it was first opened (or back to an empty file if being created).

One particularly useful pair of commands are the Indent / Dedent commands. These indent and “un-indent” a selected (highlighted) set of program lines. The number of spaces of indentation by default is four, which follows the Python convention for style. Therefore, it is recommended that you do not change this value. (It can be set under the Options / Configure IDLE menu selection.) Another useful set of commands is Comment Out and Uncomment. These commands add (and remove) a “double comment” symbol, ##, on a selected set of lines. This is useful for disabling certain portions of a program during program development and testing. A summary of some of the editing commands in IDLE is given in Figure A-12. *The shortcut keys given here are worth learning so that you can be more efficient in your programming.* Each command has a corresponding menu option.

We show all the menu options in the editor window of IDLE below. The **File menu** provides the ability to open, save and print files, shown in Figure A-13.

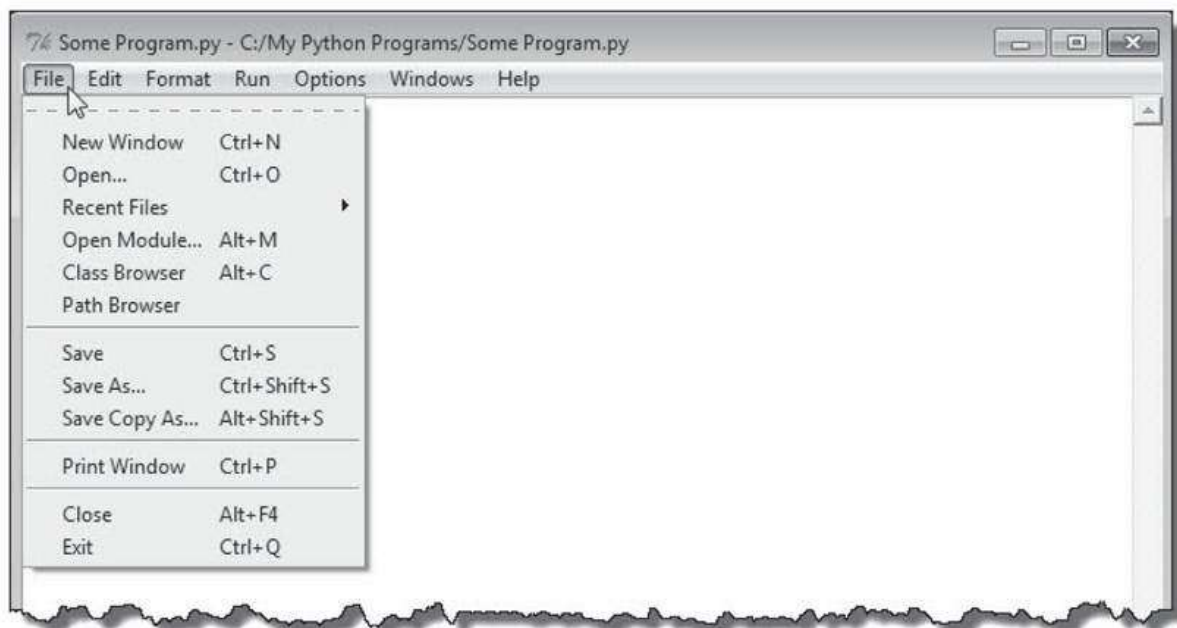


FIGURE A-13 The File Menu of the IDLE Edit or Window
The **Edit menu** provides options for the usual editing commands such as Undo/Do, Copy/Cut/ Paste, Find/Replace, shown in Figure A-14.

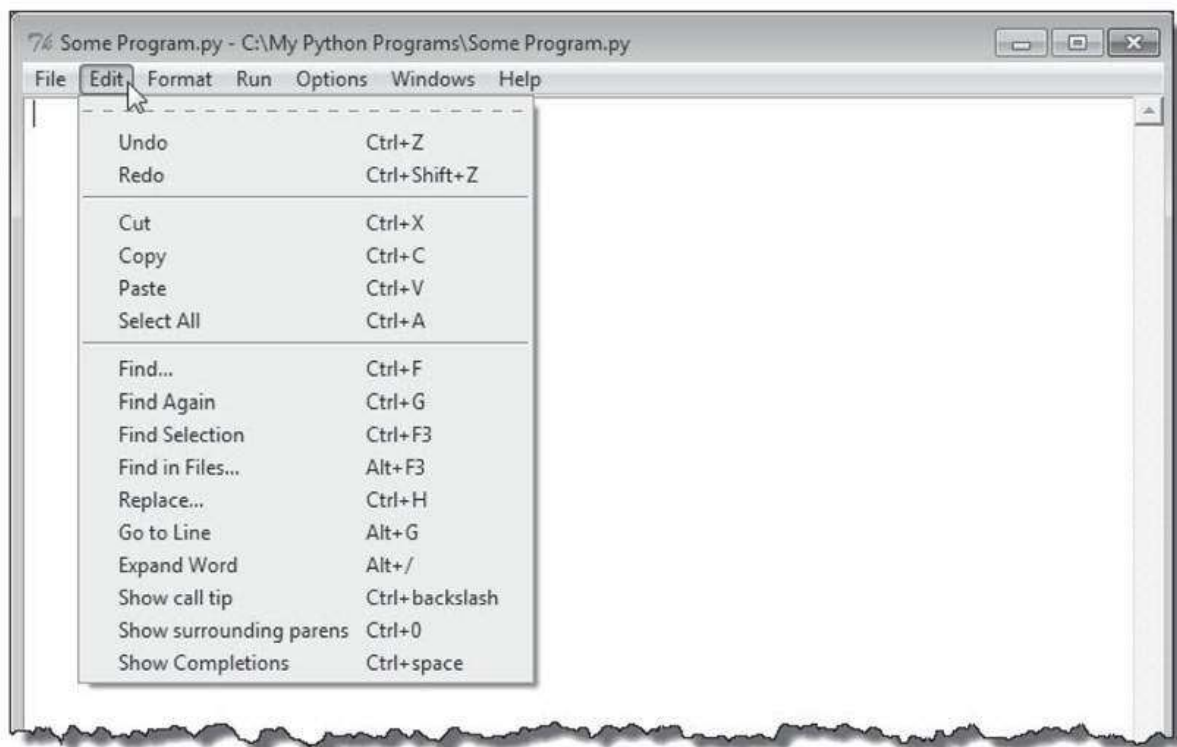


FIGURE A-14 The Edit Menu of the IDLE Edit or Window

The **Format** menu provides options for Indenting / Dedenting, and Commenting Out / Uncommenting a section of code, shown in Figure A-15.

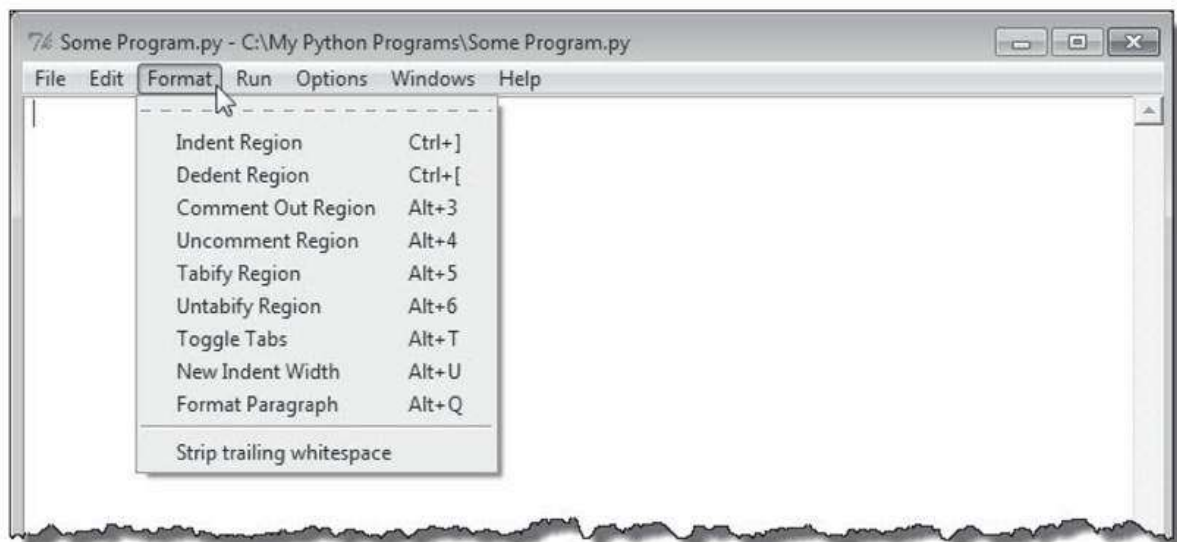


FIGURE A-15 The Format Menu of the IDLE Edit or Window

The **Run** menu provides the Run Module, which executes the Python code currently in the editor window, shown in Figure A-16. The *F5* shortcut key is a convenient way to execute a program.

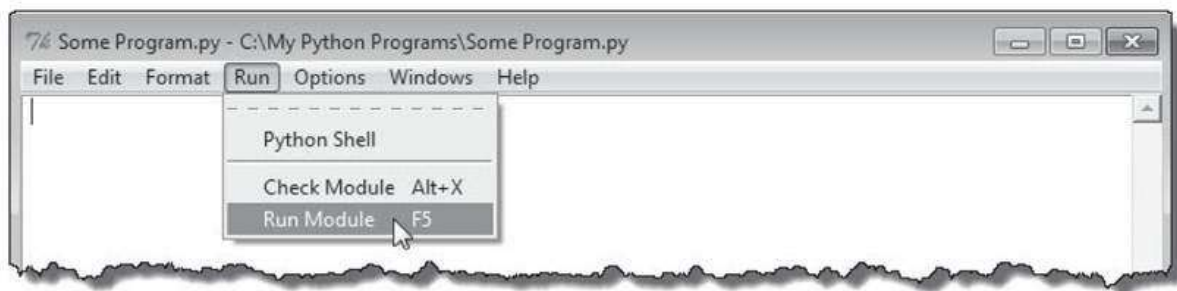


FIGURE A-16 The Run Menu of the IDLE Edit or Window

The **Options menu** provides the option Configure IDLE for configuring various aspects of IDLE, including the fonts used, the color of keywords, the color of comment lines, etc., and the number of spaces used for each tab character (for indentation of program lines). Although there is a lot of control provided for the “look and feel” of IDLE, it is recommended to stick with the standard

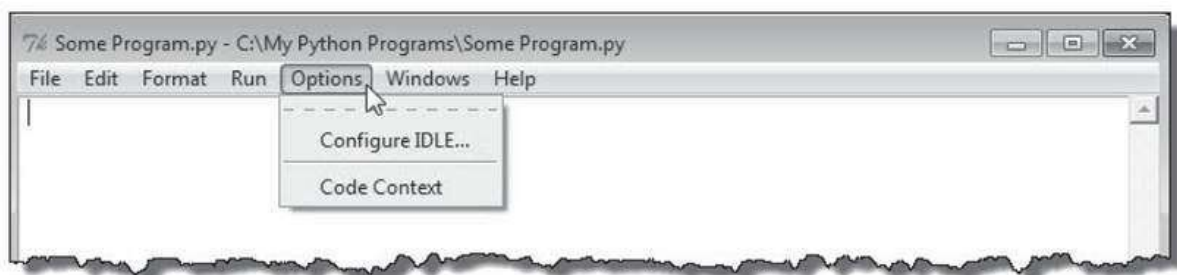


FIGURE A-17 The Options Menu of the IDLE Edit or Window

configuration options. (Code Content is a feature intended to help aid in keeping track of the lines in the same program block while scrolling through a file. We do not see the benefits of this option for our purposes.)

The **Windows menu** provides options for controlling the height of the window. It also provides a list of all currently open Python windows (by file name), including the Python Shell. This provides an easy way to switch from one window to another. The menu options are shown in Figure A-18.

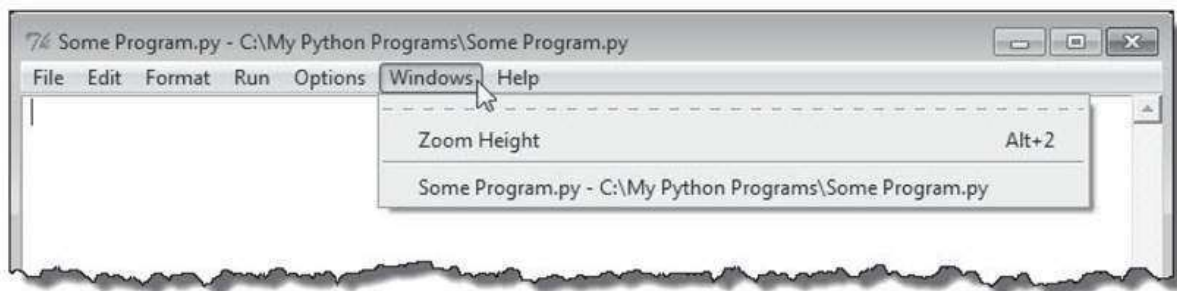


FIGURE A-18 The Windows Menu Options of the IDLE Edit or Window
Finally, the **Help menu** includes About IDLE, which includes the version number of IDLE installed; and IDLE Help, which gives a brief summary of the commands of the menu bar.

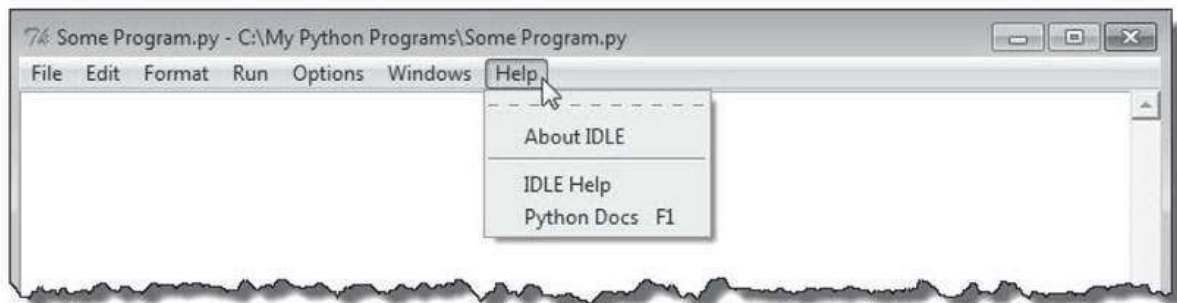


FIGURE A-19 The Help Menu Options of the IDLE Edit or Window

Using Python Docs from within IDLE A very useful feature of IDLE is the **Python Docs** option under the Help menu. This option links to the official documentation for the version of Python installed, shown in Figure A-20. The most relevant parts of the documentation for this text are marked with a (dark) checkmark. The Tutorial starts with an introduction to Python, and goes through all the features of the language, thus covering material beyond the scope of this text. It is a good starting point, however, for obtaining more information on a particular language feature. The Library Reference lists all the built-in functions, constants and types. In addition, it contains a categorized list of the Standard Library modules in Python. Finally, the Python Language Reference contains all information about the “core” of the language. This includes documentation on general syntax, expressions, statements, and compound statements (such as if and while statements). The Global Module Index, General Index and Python FAQs, indicated by lighter check marks, may also be of some help to the reader.



FIGURE A-20 Python Docs

A4. Common Python Programming Errors

We list the typical errors for novice programmers using Python. It would be wise to use the following as a checklist when developing and debugging your Python programs.

◆ Improper Indentation

All instructions in the same suite (block) must be indented the same amount. Each tab press will move the cursor the number of character spaces that is set under Options / Configure IDLE (four spaces by default, recommended).

◆ Forgetting About Truncated (Integer) vs. Real Division

Keep in mind the difference between the / operator ($5/4 \rightarrow 1.25$) and the // operator ($5/4 \rightarrow 1$). This is very easy to forget when using arithmetic expressions.

Confusing the Assignment Operator (=) with the Comparison Operator (==) ◆

This is a very common error for new programmers, but one that you should learn to avoid as early as possible.

◆ Forgetting to Convert String Values when Inputting Data

Remember that the input function always returns a string type. It is easy to forget when reading in numeric values to convert them to integer or float before using arithmetically.

◆ Forgetting to Use Colons Everywhere Needed

Don't forget to put colons where needed. Rather than trying to memorize where they are required, a simple rule can be followed. A colon is required after any keyword (such as if or while) in which the subsequent statements are indented.

◆ Forgetting about Zero-Based Indexing

Zero-based indexing, in which the first index value of an indexed entity starts at 0, can lead to “off-by-one” errors if the programmer is not careful.

◆ Confusing Mutable and Immutable Types

Remember that the value of a mutable type can be changed without the need for reassignment, for example, `list1.append(40)`. Since tuples and strings are immutable types and thus their values cannot be changed, reassignment of the variable is needed in order to “change” it. Thus, the statement `str1 = 'There! '` does not change the value of `str1`. To change its value, it must be reassigned, `str1 = 'There! '`

◆ Forgetting the Syntax for Tuples of One Element

Tuples are the only sequence type that requires a comma with tuples of only element, $(1,) \rightarrow (1)$. If the comma is left out, then the expression evaluates to that element, $(1) \rightarrow 1$.

◆ Improperly Ended Program Lines

Forgetting to use the backslash (\) when continuing a program line to the next line.

B. PYTHON QUICK REFERENCE

The references pages contained here summarize aspects of the Python programming language most relevant to the textbook. Therefore, the functions/operators listed, and the available optional arguments for each is not meant to be comprehensive. For complete coverage, see The Python Language Reference of the official Python site at <http://docs.python.org/reference/index.html> .

B1. Python Coding Style 539

B1. Python Coding Style

Use four spaces for each indentation level.

Use blank lines, sparingly, to separate logical sections of code. Separate function definitions with two blank lines. (Same for class definitions)

Limit length of lines to 79 characters (i.e., do not wrap lines around screen).

Statements containing open parentheses, square brackets or curly braces can be continued on the next line (except when containing open single or open double quotes):

```
result = (num1 + num2 + num3 + num4 + num5 +
num6 + num7 + num8 + num9 + num10)
```

Statements without such delimiters can be continued by use of the line continuation character (\):

```
response = \
int(input('(1)continue processing, (2)quit program '))
```

This: `i = i + 1`

Not This: `i=i+1`

This: `c = (a + b) * (a - b)` **Not This:** `c = (a+b) * (a-b)` **This:** `(a, b, c, d)`

Not This: `(a,b,c,d)`

This:

`def func1(real, imag=0.0):` **Not This:**

`def func1(real, imag = 0):`

This:

`def func1(real, imag=0.0):` **Not This:**

`def func1(real, imag = 0):`

B2. Python Naming Conventions

All uppercase, underscores used when aids readability: `RATIO`

`ANNUAL_RATE`

All lowercase, underscores used when aids readability: `n`, `line_count`

All lowercase using underscores when needed, or mixed case (“camel case”), in which first character is lowercase, and first letter of all other words is uppercase: `calcAverage`, `calc_average`

Mixed case, with first character a capital letter: `VehicleClass`

Short name, all lowercase, underscores used when aids readability: `math`, `conv_functions`

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

B3. Comment Statements in Python 541

B3. Comment Statements in Python

Comment line The hash sign (#) is used to make single line comments (when at the start of a line), or an in-line comment (when within a line). All characters following the hash sign until the end of line are treated as a comment.

In-line comment, e.g.,

```
state_tax_rate = 0.08 # 2008 tax rate
```

Single line comment serving as program section heading, e.g., `# sum all values greater than 0`

```
for i in range(0, len(values)):
```

```
if values[i] > 0:
```

```
sum = sum + values[i]
```

As an alternative to use of a triple-quoted docstring, e.g.,

```
def function1(n):
```

```
#-----# This function returns the largest integer #  
less than or equal to n.
```

```
#-----
```

Never include an in-line comment that states what is obvious from the program line, e.g.,

```
n = n + 1 # increment n
```

But this is ok, `n = n + 1 # adjust n to avoid off-by-one error`

If a # appears in a line with a prior #, then the second# is taken as part of the comment. There is no provision for block comments in Python (i.e., commenting out a sequence of lines with one set of comment delimiters).

B4. Literal Values in Python

None

None is a special “place marker” in Python that can be used when there is no available value.

Numeric literals never contain commas... **Yes:** 1200 204231

No: 1,200 204,231

Integer literals never contain a decimal point... **Yes:** 1200

No: 1200. 1200.0

There is no limit for the size of an integer Floats are limited to 10^{-308} to 10^{308} with 16 to 17 digits of precision

Floating-point literals must contain a decimal point...

Yes: 1200.0 1200.

No: 1200

Strings are delimited (“surrounded”) by single or double quotes: **Examples:**

'Hello World!' "Hello World!"

Strings must be delimited by the same type of quote characters: **Yes:** 'Hello World!'

No: 'Hello World!'"

Strings containing quotes must be delimited with the other quote type:

Yes: "This is John's car" **No:** 'This is John's car' 'We all yelled "Hey John!"' "We all yelled "Hey John!""

Single and double-quoted strings must be contained on one line. Triple-quoted strings, however, can span more than one line. Triple-quoted strings may be denoted with three single quotes, or three double quote characters. These strings are mainly used as docstrings for program documentation.

There are only two literal values for the Boolean type:

Yes: True False

No: true false TRUE FALSE 'True' 'False'

B5. Arithmetic, Relational, and Boolean Operators in Python 543

B5. Arithmetic, Relational, and Boolean Operators in Python

+ Addition

- Subtraction * Multiplication

** Exponentiation / Float division

// Truncated division

% Modulus < Less than > Greater than

<= Less than or equal to

>= Greater than or equal to

== Equal != Not equal

and or

not

logical AND logical OR

logical NOT

cond1 = True cond2 = False

2 + 3 - 1 4 2 < 3 True cond1 and cond2 False

2 * 6 12 3 <= 2 False cond1 and not cond2 True

2 ** 6 64 3 == 2 False cond1 or cond2 True

3 / 2 1.5 3 != 2 True not cond1 or cond2 False

3 // 2 1 'A' < 'B' True not(cond1 and cond2) True

3.0 // 2 1.0 'a' < 'B' False (2 < 3) and cond1 True

2025 // 100 20 'Hill' < 'Wu' True 2025 % 100 25

** exponentiation

*, /, //, % multiplication, division, modulus (remainder) +, addition, subtraction

<, <=, ... in, not in relational, membership, and identity operators not Boolean
not operator

and Boolean and operator

or Boolean or operator

NOTE: All listed operators associate left-to-right, except ** (which associates right-to-left).

B6. Built-in Types and Functions in Python

int Integers

float Floating point numbers complex Complex numbers bool Boolean values

dict Dictionary

str Strings list Lists tuple Tuples

abs(x)

Returns the absolute value of a number (for arguments of type int, long, and float). set Sets

frozenset Immutable sets

chr(i)

Returns a string of one character whose ASCII code is the integer i. The argument must be in the range [0...255] inclusive, otherwise a ValueError will be raised.

cmp(x, y)

Compare the two objects x and y and returns a negative value if x<y, zero if x==y, and a positive value if x>y.

dict(arg)

Creates a new dictionary where arg is a list of the form [(attr1, value1), (attr2, value2), ...].

float(arg)

Returns the absolute value of a number (for arguments of type int, long, and float).

format(value, format_spec)

Creates a formatted string from a provided sequence of format specifiers. (See details in Section B10)

frozenset(arg)

Creates a frozen (immutable) set where arg is a sequence (or other iterable type).

B6. Built-in Types and Functions in Python 545

(...continued)

help(arg)

Used in interactive mode in the Python Shell. Displays a help page related to the

string provided in `arg`. When an argument not provided, starts the built-in help system. **print(arg1, arg2, ...)**

id(object)

Prints arguments `arg1, arg2, ...` on the screen.

Returns the “identity” of an object, which is an integer unique to all other objects, and **range(opt_start, stop, opt_step)**

remains the same during an object’s lifetime.

Creates a list of integer values of a given progression. If `opt_start` not provided, then

input(prompt)

sequence begins at 0. If `opt_step` not provided, then increments by 1.

Prompts user with provided `prompt` argument to enter input, and returns the typed input as a string with the newline (`\n`) character removed. (See details in section B7)

Returns a string which is a printable representation of `arg`.

int(arg)

reload(module) Returns integer value of provided argument. Argument may be a string or an integer (in which case, the same integer value is returned). Reloads a given module that has already been loaded (imported). Useful for when changes to modules when in interactive mode.

len(arg)

Returns the length of `arg`, where `arg` may be a string, tuple, list, or dictionary. **round(x,**

opt_n)

Returns the floating point value `x` rounded to `n` decimal places. If `n` is omitted, decimal place **list(arg)**

returned as 0.

Returns a list with items contained in `arg`, where `arg` may be a sequence (or other iterable type). If no argument given, returns an empty list.

Returns a new set with values in `arg`, where `arg` is a sequence (or other iterable) type. **max(arg)**

Returns the largest value in a provided string, list or tuple.

Returns a new sorted list with values in `arg`, where `arg` is a sequence (or other iterable) type. **min(arg)**

str(arg) Returns the smallest value in a provided string, list or tuple.

Returns a string version of `arg`. If `arg` is omitted, returns the empty

string.
string.open(filename, opt_mode)

tuple(arg) Opens and creates a file object for `filename`. Optional

argument `opt_mode` either 'r' (reading),

'w'

(writing), or

'a'

(appending). If omitted, file opened for reading.

Returns a tuple whose items are the same as in `arg`, where `arg` may be a sequence (or other iterable type). If no argument is given, returns a new empty tuple.

ord(str)

type(arg) Given a character (string of length one), returns the Unicode code point (character encoding

value) for the character, e.g. `ord('a') = 97`

Returns the type of a value (object) provided in `arg`.

(...continued)

pow(x, y)

Returns `x` to the power `y`. Equivalent to `x**y`.

print(arg1, arg2, ...)

Prints arguments `arg1, arg2, ...` on the screen. (See details in section B7)

range(opt_start, stop, opt_step)

Creates a list of integer values of a given progression. If `opt_start` not provided, then sequence begins at 0. If `opt_step` not provided, then increments by 1.

repr(arg)

Returns a string which is a printable representation of `arg`.

reload(module)

Reloads a given module that has already been loaded (imported). Useful for when making changes to modules when in interactive mode.

round(x, opt_n)

Returns the floating point value `x` rounded to `n` decimal places. If `n` is omitted, decimal place returned as 0.

set(arg)

Returns a new set with values in `arg`, where `arg` is a sequence (or other iterable type).

`sorted(arg)`

Returns a new sorted list with values in `arg`, where `arg` is a sequence (or other iterable type).

`str(arg)`

Returns a string version of `arg`. If `arg` is omitted, returns the empty string.

`tuple(arg)`

Returns a tuple whose items are the same as in `arg`, where `arg` may be a sequence (or other iterable type). If no argument is given, returns a new empty tuple.

`type(arg)`

Returns the type of a value (object) provided in `arg`.

B7. Standard Input and Output in Python 547

B7. Standard Input and Output in Python

Standard input (`sys.stdin`) is a data stream used by a program to read data. By default, standard input is from the keyboard.

`input(prompt)` The `input` function sends (optional) string parameter, `prompt`, to the standard output (the screen) to prompt the user for input. It then returns the line read from the standard input (the keyboard) as a string, with the trailing newline character removed.

`print(value, ..., sep=' ', end='\n', file=sys.stdout)` The `print` function sends values to the standard output (the screen). Multiple values may be given, each separated by commas. The displayed values are separated by a blank character and ended with a newline character. Each of the default parameter values, `sep`, `end` and `file`, may be changed as keyword arguments. Standard output (`sys.stdout`) is a data stream used by a program to write data. By default, standard output goes to the screen.

```
>>> name = input('Enter name: ') Enter name: Audrey Smith >>> name
'Audrey Smith'
```

```
>>> age = 38
```

```
>>> print('His age is', age)) His age is 38
```

```
>>> n = int(input('Enter age: ')) Enter age: 28
```

```
>>>n
```

28

```
>>> print(1, 2, 3, sep='*') 1*2*3
```

```
>>> print(1, 2, 3, end='...') 1 2 3...
```

The prompt string provided for the function `input` often is ended with a blank character to provide space between the end of the prompt and the typed user input.

To keep the cursor on the same line when calling `print`, set default parameter `end` to the empty string as a keyword argument.

The `input` function reads and returns any input by the user without generating an error. When the input is converted to another type, such as an integer in (2) above, the type conversion function used may raise an exception. Thus, exception handling is needed to check for invalid input.

B8. General Sequence Operations in Python

len(s)

Returns the length of sequences.

s[i] (*selection*)

Returns the item at index *i* in sequence *s*.

s[i:j] (*slice*)

Returns subsequence (“slice”) of elements in *s* from index *i* to index *j*–1.

s[i:j:k] (*slice with step*)

Returns subsequence (“slice”) of every *k*th item in *s*[*i*:*j*].

s.count(x)

Returns the number of occurrences of *x* in sequence *s*.

s.index(x)

Returns the first occurrence of *x* in sequence *s*.

x in s (*membership*)

Returns True if *x* is equal to one of the items in sequence *s*, otherwise returns False.

x not in s (*membership*)

Returns True if *x* is not equal to any of the items in sequence *s*, otherwise returns False.

s1 + s2 (*concatenation*)

Returns the concatenation of sequence *s2* to sequence *s1* (of the same type).

n * s (or **s * n**)

Returns *n* (shallow) copies of sequence *s* concatenated.

min(s)

Returns the smallest item in sequence s.

max(s)

Returns the largest item in sequence s.

B 9 . String Operations in Python 549

B 9 . String Operations in Python

str.find(arg) / str.index(arg)

Both find and index return the lowest index matching substring provided. i

Method find returns -1 if substring not found, method index returns ValueError.

str.isalpha(arg) / str.isdigit(arg)

Returns True if str non-empty and all characters in str are all letters (isalpha), or all digits (isdigit), otherwise returns False.

str.isidentifier()

Returns True if str is an identifier in Python, otherwise returns False.

str.islower() / str.isupper() / str.lower() / str.upper() Methods islower / isupper return True if all letters in str are lower (upper) case, and there is at least one letter in str. Methods lower / upper return a copy of str with all letters in lower (upper) case.

str.join(arg)

Returns a string which is the concatenation of all strings in arg, where arg is a sequence (or some other iterable) type. If arg does not contain any strings, a TypeError is raised.

str.partition(arg)

For string separator in arg, returns a 3-tuple containing the substring before the separator in str, the separator itself, and the substring following the separator. If separator not found, returns a 3-tuple containing str, and two empty strings.

str.replace(arg1, arg2)

Returns a copy of str with all occurrences of arg1 replaced by arg2.

str.split(arg)

Returns a list of words in str using arg as the delimiter string. If arg not provided, then whitespace (a blank space) is used as the delimiter.

str.strip(arg)

Returns a copy of `str` with leading and trailing chars contained in string `arg` removed. If `arg` not provided, then removes whitespace.

B10. String Formatting in Python

`format(value, format specifier)` Built-in function `format` creates a formatted string from a provided sequence of format specifiers.

A format string may contain the following format specifiers in the order shown: `fill_chr`, `align`, `width`, `precision`, `type`

fill_chr Any character except '{' *align* '<' (left-justified), '>' (requires a provided *align* specifier) (right-justified), '^' (centered)

width An integer (total field width)

precision An integer (number of decimal places to display, rounded) *type* 'd' (base 10), 'f' (fixed point), 'e' (exponential notation)

```
avg = 1.1275, sales = 143235 factor = 134.10456
```

```
>>> print('Average rainfall in April:', format(avg, '.2f'), 'inches') Average rainfall  
for April: 1.13 inches
```

```
>>> print('Yearly Sales $', format(sales, ',')) Yearly Sales $ 143,235
```

```
>>> print('Conversion factor:', format(factor, '.2e')) Conversion factor: 1.34e+02
```

```
>>> print(format('Date', '^8'), format('Num Sold', '^12')) Date Num Sold
```

```
>>> print("\n" + format('Date', '<8') + format('Num Sold', '^12') + "\n" + format('----  
, '<8') + format('-----', '^12'))
```

```
Date Num Sold
```

```
-----
```

B11. Lists in Python 551

B11. Lists in Python

lst[i] = x

Element `lst[i]` replaced with `x` (for any object `x`).

lst[i:j] = t

Slice of `lst` from `i` to `j-1` replaced with sequence (or other iterable type) `t`.

del lst[i:j]

Removes slice `lst[i:j]` from `lst`.

lst[i:j:k] = t

Replaces items `lst[i:j:k]` in `lst` with `t`

del lst[i:j:k]

Removes slice `lst[i:j:k]` from `lst`.

s.append(x)

Adds x to the end of sequence s.

s.extend(x)

Adds the *contents* of sequence x to the end of sequence s.

s.insert(i, x)

Inserts x at index i in sequence s.

s.pop(i)

Removes and returns s[i]. When argument i not provided, removes last item in s.

s.remove(x)

Removes and returns the *first* item in sequence s that equals x. **s.reverse()**

Reverses the items in sequence s. **s.sort()**

Sorts sequence s from smallest to largest.

B12. Dictionaries in Python**dict(arg)**

Returns a new dictionary initialized with items in arg, where arg may be a list, tuple or string (or other iterable type). If no argument provided, returns an empty dictionary.

len(d)

Returns the number of items in dictionary d.

d[key]

Returns item of dictionary d with key key.

d[key] = value Sets d[key] to value.

del d[key]

Removes d[key] from dictionary d.

key in d

Returns True if d has key key, otherwise returns False.

key not in d

Returns False if d has key key, otherwise returns True.

d.clear()

Removes all items from dictionary.

d.get(key)

Returns the value for key in dictionary d.

d.pop(key)

Removes key/value pair in d for key and returns the associated value.

d.popitem()

Removes and returns an arbitrary key/value pair from dictionary d.

B13. Sets in Python 553

B13. Sets in Python

set(arg) / frozenset(arg)

Returns a new set or frozenset with items provided in arg, where arg is a sequence (or other iterable type).

len(s)

Returns the number of items in sets.

x in s (x not in s)

Returns True (False) if s has item equal to x, otherwise returns False (True).

s.add(x) / s.remove(x), s.pop(), s.clear() (*Set type only*) Adds / removes x, Removes-returns arbitrary item from s, Removes all items from s.

s.isdisjoint(other)

Returns True if sets have no items in common with the set provided by argument other, otherwise returns False.

s.issubset(other) (also set <= other)

Returns True if every item in s is also in other, otherwise returns False.

s.issuperset(other) (also set >= other)

Returns True if every item in other is also in s, otherwise returns False.

set < other (set > other)

Returns True if s is a proper subset (superset) of other

s.union(other, ...)

Returns a new set containing all items in set s and all other provided sets.

s.intersection(other, ...)

Returns a new set containing all items common to set s and all other provided sets.

s.difference(other, ...)

Returns a new set containing all items in set s that are not also in other provided sets. **s.symmetric_difference(other)**

Returns a new set containing items that are in set s or other, but not both.

B14. if Statements in Python

if condition: statements

elif condition: statements

else:

statements

An if statement may have zero or more elif clauses, optionally followed by an else clause. As soon as the condition of a given clause is found true, that clause's statements are executed, and the rest of the clauses are skipped. If none of the clauses are found true, then the statements of the else clause are executed (if present). As with all data structures, if statements may be nested.

```
if n == 0: print('n is zero')
```

```
if n == 0:
    print('n is zero')
else:
    print('n is non-zero')
```

```
if n < 0:
    print('n is less than 0')
elif n == 0:
    print('n is zero')
else:
    print('n is greater than zero')
if n1 == 0:
    print('n1 is zero')
if n2 == 0:
    print('n2 is zero')
```

```
if n1 == 0:
    print('n1 is zero')
if n2 == 0:
```

```
    print('n2 is zero')
else:
    print('n2 is not zero')
else:
    print('n1 is not zero')
```

It is not required to include elif or else clauses in if statements.

When selecting among a set mutually exclusive conditional expressions, the if/elif (with optional else) form of if statement should be used.

Forgetting to add a semicolon (;) after each conditional expression, and after keyword else.

2. Improper operator use when checking for equality (e.g., “==”, not “=”).
3. Improper Boolean expression evaluation resulting from unconsidered operator precedence.
4. Improper indentation of nested if/elif/else clause(s) resulting in faulty logic.

B15 . for Statements in Python 555

B15 for Statements in Python

for *k* in *sequence*: *statements*

The for statement is used to control a loop that iterates once for each element in a specified sequence of elements such as a list, string or other iterable type.

```
empty_str = "", space = ' ' for num in [2, 4, 6, 8]: print(num, end=space)
```

```
2 4 6 8
```

```
lst = [2, 4, 6, 8]
```

```
for k in range(len(lst)):
```

```
    lst[k] = lst[k] + 1
```

```
for num in range(2, 10, 2): print(num, end=space) 2 4 6 8
```

```
for num in range(10, 2, -2): print(num, end=space) 10 8 6 4
```

```
for char in 'Hello':
```

```
    print(char, end=empty_str) Hello
```

```
print(lst) 3 5 7 9
```

```
t = [[1, 2, 3],[4, 5, 6]]
```

```
for i in range(len(t)):
```

```
    for j in range(len(t[i])): print(t[i][j], end=space) print()
```

```
1 2 3
```

When the elements of a sequence need to be accessed but not altered, a for loop that iterates over the values in the sequence is the appropriate approach (1-4).

When the elements of a sequence need to be both accessed and updated, a for loop that iterates over the index values in the sequence is the appropriate approach (5,6). Nested for loops can be used to iterate through a list of sequence types (6).

1. Forgetting to add a semicolon (;) after the loop header.
2. Improper indentation of statements contained in the body of the for loop.
3. Forgetting that range(i,j) (and range(j)) generates values up to, but not including j.

B16. while Statements in Python

while *condition*: The statements contained in a while loop body continue

to *statements* execute until the condition evaluates to False.

```
empty_str = "", space = ' ' i = 1
```

```
while i < 10:  
    print(i, end=space) i = i + 1
```

```
1 2 3 4 5 6 7 8 9 10  
i = 10
```

```
while i != 0:  
    print(i, end=space) i = i - 1
```

```
10 9 8 7 6 5 4 3 2 1 while True:  
    print("This loop keeps running!")
```

This loop keeps running! This loop keeps running! etc.

```
file = open('filename.txt', 'r') line = file.readline()  
while line != empty_str:
```

```
line = file.readline()  
num = int(input('Enter a positive number: '))
```

```
while num < 0:  
    print('Your input was invalid.')  
    num = int(input('Enter a positive number: '))
```

Awhile loop can be used to implement any kind of loop, although sometime loops are more conveniently implemented as a for loop (1,2).

An *infinite loop* is caused by a condition that always evaluates to True (3).

A loop to execute an indefinite number of times, such as when reading from a file, must be implemented as a while loop (4,5).

A while loop is used when accepting and validating user input (5)

Forgetting to add a semicolon (;) after the conditional expression.

2. Not making progress in the body, thereby creating a loop that never ends (i.e. an infinite loop).

3. Improper indentation of statements contained in the body of the while loop.

B17. Functions in Python 557

B17. Functions in Python

A function is a named group of instructions accomplishing some task. A function is *invoked* (called) by providing its name, followed by a (possibly empty) list of arguments in parentheses. Calls to value-returning functions are expressions that evaluate to the returned function value. Calls to non-value returning functions are effectively statements called for their side effects. (Strictly speaking, they are value-returning functions since they return special value None).

```
def name(parameters): statements
return expression
```

```
def name(parameters): statements
num1 = 10, num2 = 25, num3 = 35, list1 = [1, 0, 3, 8, 0]
def avg3(n1, n2, n3):
    """Returns rounded avg."""
    return round((n1 + n2 + n3) / 3)
>>> avg3(num1, num2, num3) 23
```

```
def dashLine(len, head=""): empty_str = "
space = ' '
```

```
if head != empty_str:
    head = space + title + space
print(head.center(len, '-'))
```

```
>>> dashLine(16)
-----
>>> dashLine(16, head='price')
---- price ----
def countdown(from, to):
```

```
    for i in range(from, to-1, -1): print(i, end='...')
>>> countdown(from=5, to=1) 5...4...3...2...1...
def hello():
    print('Hello World!')
>>> hello()
Hello World!
def removeZeros(lst):
    for k in range(len(lst)-1, -1, -1): if lst[k] == 0:
        del lst[k]
```

```
>>> list1
[1, 0, 3, 8, 0]
>>> removeZeros(list1)
[1, 3, 8]
```

A triple-quoted string as the first line of a function serves as its docstring (1). Functions may define default arguments (2), and be called with keyword arguments (3). A function may be defined having no parameters (4). Mutable arguments passed to a function can become altered (5).

B18. Classes in Python

```
class classname(parentclass):

    def __init__(self, args): "docstring"
    .
    .

    def methodname(args): "docstring"
    .
    .

    def methodname(args): "docstring"
```

A class consists of a set of methods and instance variables. The instances variables are created in special method `init`. The `init` method must have an extra first parameter, by convention named `self`, that is not passed any arguments when the method is called.

A *docstring* is a single or multi-line string using triple quotes that provides documentation for methods, classes and modules in Python.

```
class XYCoord(object):

    def __init__(self, x, y): self.__x = x
    self.__y = y

    def getX(self):
    return self.__x

    def setX(self, x): self.__x = x .
```

.

```
def __repr__(self): __init__
__repr__
__str__
__neg__
__add__
__sub__
__mul__
__truediv__ __floordiv__ __mod__
__pow__

return '(' + str(self.__x) + ',' + \ __lt__ str(self.__y) + ')'__le__

__eq__ def __eq__(self, xycoord): __ne__ return self.__x == xycoord.getX() and
\ __gt__ self.__y == xycoord.getY() __ge__
```

Instance variables beginning with two underscores are treated as private. They are accessible only if the mangled form of the name is used: with a single underscore followed by the class name added to the front. For example, private instance variable `__x` in class `XYCoord` becomes `_XYCoord__x`.

B19. Objects in Python 559

B19. Objects in Python

identifier = classname(args)

An object is an instance of a class. All values in Python are objects.

- (1) `loc1 = XYCoord(5,10)` Creates a new `XYCoord` object, its reference assigned to `loc1`
- (2) `loc2 = XYCoord(5,10)` Creates a new `XYCoord` object, its reference assigned to `loc2`
- (3) `loc1`
- (4) `loc2`
- (5) `id(loc1)`
- (6) `id(loc2)`
- (7) `loc1 == loc2`
- (8) `loc1 is loc2`
- (9) `loc3 = loc1`
- (10) `loc3 == loc1`
- (11) `loc3 is loc1`
- (12) `loc3.setX(10,10)`

(13) loc3 == loc1

(5,10) , dereferenced value of identifier loc1

(5,10), dereferenced value of identifier loc2

37349072, reference value of identifier loc1

37419120, reference value of identifier loc2

True, comparison of their dereferenced values False, comparison their reference values

Assigns reference value of loc1 to loc3

True

True

Changes value of loc3 to (10,10)

False

When assigning an object to an identifier, the reference to the object is assigned, not the object itself. Thus, more than one identifier may reference the same object.

Each newly-created object has a unique id. For variables var1 and var2, if id(var1) == id(var2) (or var1 is var2 is True) they are referencing the same object.

Mutable objects (e.g., lists) can be altered without reassignment:

```
>>> lst = [1, 2, 3]
```

```
>>> lst.append(4)
```

```
>>> lst
```

```
[1, 2, 3, 4]
```

Immutable objects (e.g., strings) cannot be altered without reassignment:

```
>>> str1 = 'Hello'
```

```
>>> str1.replace('H', 'J') >>> str1
```

```
'Hello'
```

```
>>> str1 = str1.replace('H', 'J') >>> str1
```

```
'Jello'
```

B20. Exception Handling in Python

try: *statements*

except *ExceptionType*: *statements*

except *ExceptionType*: *statements*

etc.

Exception handling provides a means for functions and methods to report errors that cannot be corrected locally. In such cases, an *exception* (object) is *raised* that can be *caught* by its *client code* (the code that called it), or the client's client code, etc., until *handled* (i.e. the exception is caught and the error appropriately dealt with). If an exception is thrown back to the top-level code and never caught, then the program terminates displaying the exception type that occurred.

```
def genAsterisks(x): """x an integer in range 1-79."""
if x < 1 or x > 79:
    raise ValueError('Must enter 1-79')
return x * '*'

#---- main
valid_input = False

while not valid_input: try:
    num = int(input("How Many '*'s?: ")) print(genAsterisks(num))
    valid_input = True

except ValueError as errorMesg: print(errorMesg)
EOFError
FloatingPointError ImportError
IOError
IndentationError IndexError
KeyError
NameError
OverflowError
RuntimeError
SyntaxError
TypeError
ValueError
ZeroDivisionError
```

The exceptions built-in module is automatically imported in Python. Built-in exceptions, raised by the built-in functions/methods, may also be raised by user code. New exceptions may be defined as a subclass of the built-in `Exception` class. There may be any number of `except` clauses for a given `try` block.

In addition to the except clauses, an else clause may optionally follow, only executed if the try block does not raise any exceptions. Following that, an optional finally clause, if present, is always executed, regardless of whether an exception had been raised or not. The else and finally clauses are often used for resource allocation, such as closing an open file.

B21. Text Files in Python 561

B21. Text Files in Python

A *text file* is a file containing characters, structured as lines of text. Text files can be directly created and viewed using a text editor. In addition to printable characters, text files also contain *non-printing* newline characters, \n, to denote the end of each line of text.

A file that is open for input can be read from, but not written to.

```
fileref= open(filename, 'r')  
fileref.readline()  
fileref.close()
```

Opens and creates a file object for reading filename. Raises an IOError exception if file not found. Reads next line of file. Returns empty string if at end of file.

Includes newline character ('\n') in line read.

Closes file. File can be reopened to read from first line.

```
filename = input('Enter filename: ') inFile = open(filename, 'r')  
Enter filename: testfile.txt Hi,
```

```
line = inFile.readline() while line != "":  
print(line, end="")  
line = inFile.readline() This is a test file. Containing five lines. Including one  
blank line. >>>
```

A file that is open for output can be written to, but not read from.

```
fileref = open(filename, 'w') fileref.write(s)  
fileref.close()
```

Opens and creates a file object for writing to filename. Writes string s to file. Does *not* include output of '\n'. Closes file. If not closed, last part of output may be lost.

```
filename = input('Enter filename: ') outFile = open(filename, 'w')
```

```
line = input('Enter line of text:') while line != "":
    outFile.write(line + '\n')
    line = input('Enter line:')
```

outFile.close() Enter filename: newfile.txt This is the first entered line. This is the second entered line. This is the last entered line.

newfile.txt

This is the first entered line. This is the second entered line. This is the last entered line.

B22. Modules in Python

A Python module is a file containing Python definitions and statements. The module that is directly executed to start a Python program is called the *main module*. Python provides standard (built-in) modules in the Python Standard Library.

Each module in Python has its own *namespace*: a named context for its set of identifiers. The *fully qualified* name of each identifier in a module is of the form *modulename.identifier*.

```
import modulename
```

Makes the namespace of *modulename* available, but not part of, the importing module. All imported identifiers used in the importing module must be fully qualified: `import math`

```
print('factorial of 16 = ', math.factorial(16))
```

```
from modulename import identifier_1, identifier_2, ... identifier_1, identifier_2,
etc. become part of the importing module's namespace: from math import
factorial
```

```
print('factorial of 16 = ', factorial(16))
```

```
from modulename import identifier_1 as identifier_2
```

`identifier_1` becomes part of the importing module's namespace as `identifier_2`

```
from math import factorial as fact
```

```
print('factorial of 16 = ', fact(16))
```

```
from modulename import *
```

All identifiers of *modulename* become part of the importing module's namespace (except those beginning with an underscore, which are treated as

private).

```
from math import *  
print('factorial of 16 = ', fact(16))  
print('area of circle = ', pi*(radius**2))
```

Although the import * form of import is convenient in that the imported identifiers do not have to be fully qualified in the importing module, there is the risk of a name clash. In addition, it is not apparent which identifiers are imported, or which module they are imported from.

B22. Modules in Python 563

C. PYTHON STANDARD LIBRARY MODULES

These pages contain selected modules of the Python Standard Library. For a complete listing, see the official standard library at <http://docs.python.org/reference/index.html> .

The Module Search Path Python modules may be stored in various locations on a particular system. For this reason, when a module is imported, it must be searched for. The interpreter first searches for a built-in (standard) module with that name. If not found, it then searches for the module in the same directory as the executed program. If still not found, the interpreter searches in a list of directories contained in the variable `sys.path`, a variable of the built-in module `sys`. The `sys` module must be imported to access this variable:

```
.. . import sys  
.. .  
sys.path
```

```
['C: \Python32\Lib\idlelib', 'C: \Windows\system32\python32.zip',  
'C: \Python32\DLLs', 'C: \Python32\lib', 'C: \Python32', 'C: \Python32\lib\site-  
packages']
```

The particular value for `sys.path` depends on your particular Python installation. The `dir` Built-In Function Built-in function `dir` can be used to find out the names that a particular module defines. This may be used on any module:

```
.. . import math  
.. .  
dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',  
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',  
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',  
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',  
'sqrt', 'tan', 'tanh', 'trunc']
```

Note that in addition to the available mathematical methods, there are three special identifiers `__doc__`, `__name__` and `__package__` listed. The first two provide the docstring (providing brief documentation of the module's contents) and the module name, respectively:

```
.. . print(math.__doc__)
```

This module is always available. It provides access to the mathematical functions defined by the C standard.

```
.. . print(math.__name__)
```

`math`

The `__package__` special identifier is used for modules that contain submodules, called packages.

C1. The math Module

This module contains a set of commonly-used mathematical functions, including number-theoretic functions (such as factorial); logarithmic and power functions; trigonometric (and hyperbolic) functions; angular conversion functions (degree/radians); and some special functions and constants (including `pi` and `e`). A selected set of function from the `math` module are presented here.

`math.ceil`

`math.fabs(x)`

`math.factorial(x)` `math.floor()`

`math.fsum(s)`

`math.modf()`

`math.trunc(X)`

returns the ceiling of `x` (smallest integer greater than or equal to `x`). returns the absolute value of `x`

returns the factorial of `x`

returns the floor of `x` (largest integer less than `x`).

returns an accurate floating-point sum of values in `s` (or other iterable). returns

the fractional and integer parts of x .
returns the truncated value of x .

`math.exp(x)`
`math.log(x, base)` `math.sqrt(x)`

returns $e^{**}x$, for natural log base e .
returns $\log x$ for base. If base omitted, returns $\log x$ base e . returns the square root of x .

`math.cos(x)` `math.sin(x)` `math.tan(x)` `math.acos(x)` `math.asin(x)` `math.atan(x)`

returns cosine of x radians. returns sine of x radians. returns tangent of x radians.
returns arc cosine of x radians. returns arc sine of x radians. returns arc cosine of x radians.

`math.degrees(x)` `math.radians(x)`
returns x radians to degrees. returns x degrees to radians.

`math.pi` `math.e`

mathematical constant $\pi = 3.141592$ mathematical constant $e = 2.718281$

C2. The random Module 565

C2. The random Module

This module provides a pseudorandom number generator using the Mersenne Twister algorithm, allowing users to generate random numbers with near uniform distribution over a long period.

`random.random()` returns random float value x , where $0 \leq x < 1$.

`random.uniform(a, b)` returns random float value x , where $a \leq x \leq b$.

`random.randint(a, b)` returns random integer value x , where $a \leq x \leq b$.

`random.randrange(start, stop, step)`

returns random integer value x , where $start \leq x < stop$ and $x = start + n * step$, where n is an integer greater than or equal to zero

`random.choice(seq)` `random.shuffle(seq)` `random.sample(seq, k)`

returns random element from sequence seq (must be non-empty). randomly reorders sequence seq in place.

returns list (length k) of unique items randomly chosen from seq .

`list = ['a', 'b', 'c', 'd', 'e']`

`>>> random.random()` 0.8568285775611655

```
>>> random.uniform(1, 10) 9.71538746497116
>>> random.randrange(0, 10, 2) 6
>>> random.randrange(0, 10, 2) 8
>>> random.randrange(0, 10, 2) 2

>>> random.randint(1, 10) 6
>>> random.sample(list, 3) ['d', 'c', 'e']

>>> random.choice(list) 'e'
>>> random.shuffle(list) >>> print(list)
['c', 'b', 'e', 'a', 'd']
```

The random module must be imported before it can be used.
The current system time is used to initialize the random number generator. If comparability and reproducibility are important, supply a seed value x by invoking `random.seed(x)` before generating any random numbers.
To generate only even numbers, use the `randrange` function with start of 0, step of 2.

C3. The turtle Module

This module provides both procedure-oriented and object-oriented ways of controlling *turtle graphics*. A turtle is a graphical entity in an x/y coordinate plane (Turtle screen) that can be controlled in various ways including: its shape, size, color, position, movement (relative and absolute), speed, visibility (show/hide), and drawing status (pen up/pen down).

```
# set screen size
turtle.setup(800,600) # getting the default turtle t = turtle.getturtle()

# get reference to turtle screen screen = turtle.Screen()
# creating a new turtle t = turtle.Turtle()

# set window title bar screen.title('My Turtles')

screen.bgcolor(args) ..... background color, specified by name, (RGB) or hex
screen.bgpic(filename) ..... sets GIF file as screen background screen.clear()
..... clears the screen
screen.reset() ..... resets all turtles to their initial state screen.bye()
..... closes turtle screen window
```

screen.exitonclick() closes turtle screen window on mouse click

showturtle() / hideturtle() makes turtle visible / invisible

turtle.register_shape(filename) .. GIF file name, registers image for use as turtle shape
shape(shape) sets turtle to regular or registered shape

turtle.isvisible() returns True if turtle currently visible
turtle.position() returns current position of turtle as x,y coordinate

turtle.towards(x,y) returns angle between turtle and coordinate x,y

turtle.xcor() returns turtle's current x coordinate

turtle.ycor() returns turtle's current y coordinate

turtle.heading() returns turtle's current heading

turtle.distance(arg) returns distance to (x,y) or to another turtle

C3. The turtle Module 567

turtle.pendown() puts turtle's pen down so draws when it moves

turtle.penup() lifts turtle's pen so doesn't draw
turtle.isdown() returns True if pen currently down

turtle.pencolor(color) sets pen's color by color name or (RGB)

turtle.pensize(size) sets pen's line thickness, size a positive number

turtle.fillcolor(color) sets pen's fill color by color name or (RGB)

turtle.clear() deletes turtle's drawing from screen

turtle.write(arg) writes text representation of arg at turtle's location

turtle.forward(x)/backward(x)... moves turtle forward/backward x pixels

turtle.right(angle)/left(x).....moves turtle left/right by angle

turtle.goto(x,y).....moves turtle to coordinate (x,y)

turtle.setx(x)/sety(y) moves turtle to x/y locations

turtle.setheading(angle) sets heading of turtle by angle

turtle.undo() undoes last turtle action

turtle.tilt(angle).....rotates turtle relative to its current tilt-angle

turtle.settiltangle(angle) sets turtle to angle

turtle.tiltangle() returns the turtle's current tilt angle

turtle.speed(speed) 1-slowest, 2-faster, ... 10-fast, 0-fastest

turtle.home() moves turtle to original position

turtle.circle(radius) draws a circle if size radius

turtle.dot(size, color) draws a dot with given size and color

turtle.stamp() stamps turtle shape screen, return unique stamp id

turtle.clearstamp(id) clears turtle stamp with provided id

turtle.clearstamps(n) clears last n stamps

stamps, if argument omitted, clears all

`turtle.onclick(func)` `func` a function of two arguments called with the x,y coordinates of the location of mouse click

`turtle.onrelease(func)` `func` a function of two arguments called with the x,y coordinates of the location of mouse release

`turtle.ondrag(func)` `func` a function of two arguments called with the x,y coordinates of the location of mouse click when mouse dragged

`turtle.mainloop()` starts event loop. Must be last statement in a turtle graphics program

C4. The webbrowser Module

This module provides functionality to open Web-based documents in a browser from within a Python program. The module launches the system's default browser to open the target URL, unless a different (supported) browser is specified.

`webbrowser.open(url, new=0, autoraise=True)`

Opens the page at the specified `url` in a browser window. If possible, a value of 0 for `new` opens the page in an existing browser window, 1 in a new browser window, and 2 in a new browser "tab." When `autoraise` is passed as `True`, the browser window is raised to the top of the stack (this may happen on some systems regardless of `autoraise` value).

`webbrowser.open_new(url)/webbrowser.open_new_tab(url)` Always opens the specified `url` in a new browser window or tab, respectively.

`webbrowser.get(name)`

Returns a controller object for the specified browser type name, allowing the user to use any browser that is registered with or predefined in the module.

```
import webbrowser
webbrowser.open('http://docs.python.org/', 0, True)
import webbrowser
webbrowser.open_new('http://docs.python.org/')
```

```
import webbrowser
browser = webbrowser.get('firefox')
browser.open('http://docs.python.org/')
```

Python scripts can be written to generate HTML code, which can then be opened in a browser as part of the program using this module.

This module is not limited to opening Web pages. On some platforms, it can also be used to open documents and media files. Instead of passing an http URL to `open`, pass the address of the target file (e.g. `file:///host/path` or, on local machine, `file:///path`). The functionalities of this module are also available through the command line.

Index

A

569

B C D E

F

G H

I

J K M

L N

O

P Q R

S

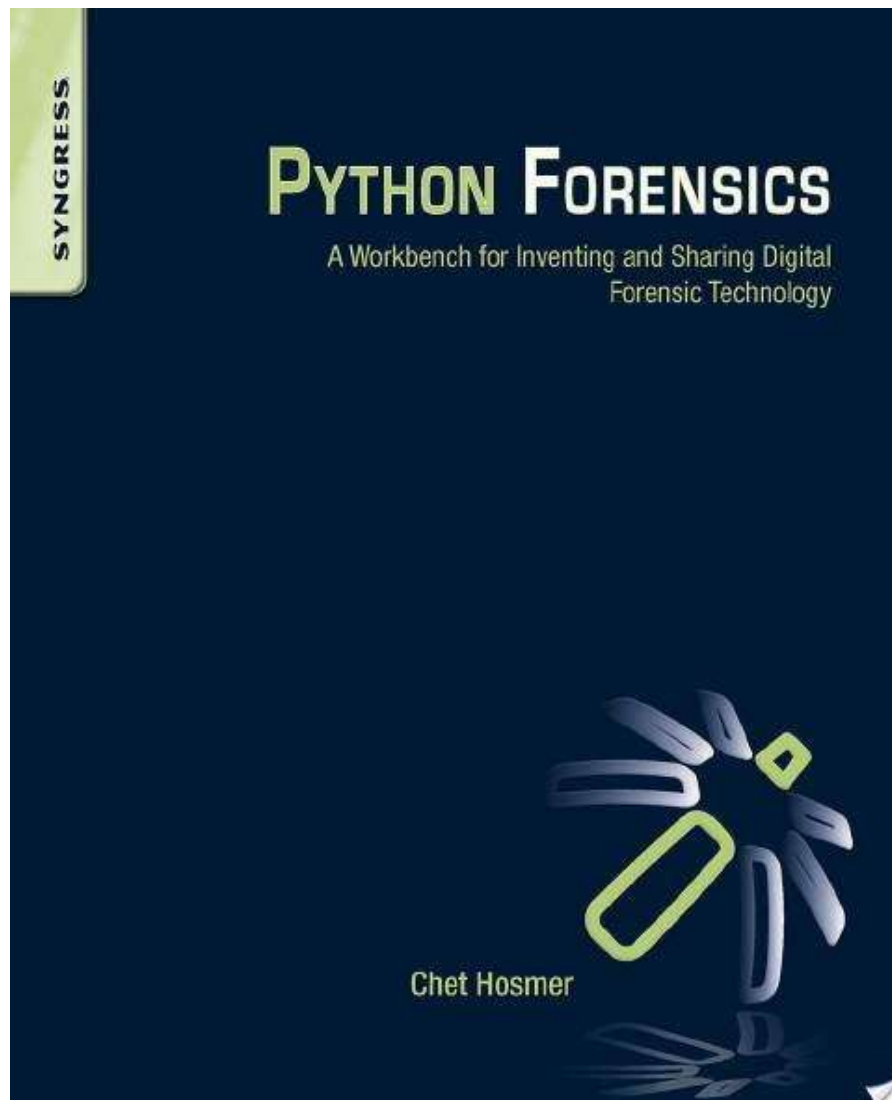
T U V

W

X

Y

Z



A Workbench for Inventing and Sharing Digital Forensic Technology

Chet Hosmer Technical Editor: Gary C. Kessler

AMSTERDAM • BOSTON • HEIDELBERG • LONDON NEW YORK •
OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Syngress is an Imprint of Elsevier

Acquiring Editor: Steve Elliot

Editorial Project Manager: Benjamin Rearick Project Manager: Priya

Kumaraguruparan Designer: Mark Rogers

Syngress is an imprint of Elsevier

225 Wyman Street, Waltham, MA 02451, USA

Copyright# 2014 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described here in. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data Application Submitted

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-12-418676-7

For information on all Syngress publications, visit our website at store.elsevier.com/syngress

Printed and bound in the United States of America 14 15 16 17 18 10 987 65432
1

To my wife Janet, for your love, kindness, patience, and inspiration that you give every day. I am the luckiest guy in the world.

Acknowledgments

My sincere thanks go to:

Dr. Gary Kessler, the technical editor for this book. Gary, your insights, fresh perspective, deep technical understanding, and guidance added great value to the book. Your constant encouragement and friendship made the process enjoyable.

Ben Rearick and Steve Elliot at Elsevier, for your enthusiasm for this topic and all the guidance and support along the way. This spirit helped more than you can know.

The many teachers that I have had over the years in software development and forensics that have helped shape the content of this book. Ron Stevens, Tom Hurbanek, Mike Duren, Allen Guillen, Rhonda Caracappa, Russ Rogers, Jordon Jacobs, Tony Reyes, Amber Schroader, and Greg Kipper.

Joe Giordano, who had the vision in 1998 to create the first U.S. Air Force research contract to study forensic information warfare. This one contract was the catalyst for many new companies, novel innovations in the field, the establishment of the digital forensic research workshop (DFRWS), and the computer forensic research and development center at Utica College. You are a true pioneer.

vii

Endorsements

“Not only does Hosmer provide an outstanding Python forensics guide for all levels of forensics analysis, but also he insightfully illustrates the foundation of a rich collaborative environment that significantly advances the forensic capabilities of the individual, organization, and forensic community as a whole. For analysts, investigators, managers, researchers, academics, and anyone else with an interest in digital forensics: this is a must read!”

Michael Duren (CISSP), Founder of Cyber Moxie

“With today’s rapid changes in technology digital forensics tools and practices are being forced to change quickly just to remain partially effective; and the technical skills investigators relied on yesterday are quickly becoming obsolete. However, with new technology comes new tools and methods, and the Python language is in one of the best possible positions to be leveraged by investigators. Python Forensics is quite simply a book that is ahead of its time, and because of this, it is the perfect book for both the beginner and the experienced investigator. Chet Hosmer does a great job of helping the reader refresh older skills and create new ones by offering step-by-step instructions and intelligently framing the information for maximum understanding and contextual awareness. The skills you will learn from Python Forensics will help you develop a flexible and innovative toolkit that will be usable for years to come.”

Greg Kipper, Senior Security Architect and Strategist at Verizon

“This book presents a refreshing, realistic view on the use of Python within modern, digital forensics; including valuable insight into the strengths and weaknesses of the language that every knowledgeable forensics investigator should understand.”

Russ Rogers, President of Peak Security, Inc.

“This book is extremely useful for the forensic Python programmer also for those with little or no programming experience, and an excellent reference cookbook for the experienced programmer. The book considers issues relating to Daubert including testing and validation which is vital for the accreditation of forensic solutions.”

Zeno Geradts, Senior Forensic Scientist and R&D coordinator at the Netherlands Forensic Institute

ix x Endorsements

“As always, Chet Hosmer provides a comprehensive and groundbreaking evaluation of a contemporary platform applicable to digital forensics. Extremely well written and user friendly, the book provides a solid foundation for all levels of forensic Python programmers, and includes a much-needed discussion on empirical validation. Quite simply, the book is a must have for all who maintain a digital forensics library.”

Dr. Marjie T. Britz, Clemson University

List of figures

- Figure 1.1 Narrowing the gap 3
- Figure 1.2 The future digital crime scene 4
- Figure 1.3 Data vs. semantics 5
- Figure 1.4 The next-generation cyber warrior 5
- Figure 1.5 Programming language popularity according to codeview.com 7
- Figure 1.6 Test-then code-then validate 7
- Figure 2.1 The Python Programming Language Official Web site 17
- Figure 2.2 Downloading the Windows installer 18
- Figure 2.3 Windows download confirmation 18
- Figure 2.4 Executing the Python 2.7.5 Installer 19
- Figure 2.5 Python Installation user selection 20
- Figure 2.6 Python Installation directory 20
- Figure 2.7 Python customization user manual 21
- Figure 2.8 TCL/TK install when needed 21
- Figure 2.9 Windows user account control 22
- Figure 2.10 Successful installation of Python 2.7.5 22
- Figure 2.11 Python directory snapshot 23
- Figure 2.12 Windows taskbar with Python icon 23
- Figure 2.13 Python startup prompt and messages 24
- Figure 2.14 Python Hello World 24
- Figure 2.15 Windows execution of hashPrint.py 26
- Figure 2.16 Ubuntu Linux execution of hashPrint.py 27
- Figure 2.17 Python Shell session using hex() and bin() 27
- Figure 2.18 Python Shell session entering hex values 28
- Figure 2.19 Python Shell creating lists using the range() built-in Standard Library function 28
- Figure 2.20 ipRange execution 29
- Figure 2.21 Built-in True and False constants 31
- Figure 2.22 Python is not a strongly typed language 31
- Figure 2.23 Apply an Exclusive OR (XOR) 33
- Figure 2.24 Example using the os module from the Standard Library 35
- Figure 2.25 Python IDLE integrated development environment 38
- Figure 2.26 Snapshot of WingIDE 4.1 Personal 40
- Figure 2.27 WingIDE Python Shell display 40
- Figure 2.28 WingIDE in action 41
- Figure 2.29 Using WingIDE to step through code 42

Figure 2.30 WingIDE examination of the completed list 43
Figure 2.31 WingIDE auto complete feature 43
Figure 2.32 Ubuntu download Web page 12.04 LTS 44
Figure 2.33 Ubuntu terminal window Python command 45
Figure 2.34 Ubuntu software center 45
Figure 2.35 WingIDE running on Ubuntu 12.04 LTS 46

xvii xviii List of figures

Figure 2.36 Python Shell running on iOS 47
Figure 2.37 iOS implementation of HashPrint 48
Figure 2.38 Apple App Store page regarding Python for iOS 48
Figure 2.39 Windows 8 Phone screenshot of PyConsole launching 49
Figure 2.40 Python Console “Hello World” on a Windows 8 Phone 49
Figure 2.41 Windows 8 Phone HashPrint application execution 50
Figure 2.42 Python Console Windows App Store page 50
Figure 3.1 Cryptographic SmartCard 54
Figure 3.2 Context diagram: python-file system hashing (p-fish) 61
Figure 3.3 p-fish internal structure 62
Figure 3.4 p-fish WingIDE setup 64
Figure 3.5 Demonstration of ParseCommandLine 70
Figure 3.6 pfish -h command 70
Figure 3.7 Test run of pfish.py 84
Figure 3.8 Result directory after pfish execution 84
Figure 3.9 Examining the Result File with Microsoft Excel 85
Figure 3.10 Contents of the pFishLog file 86
Figure 3.11 Linux Command Line Execution 87
Figure 3.12 Linux Execution Results pfish Result File 88
Figure 3.13 Linux Execution Results pFishLog File 89
Figure 4.1 Snapshot of stackdata displaying the baTarget object and the size in bytes of the baTarget bytearray 95
Figure 4.2 p-search context diagram 97
Figure 4.3 p-search internal structure 98
Figure 4.4 WingIDE p-search execution 99
Figure 4.5 p-search execution using only the -h or help option 102
Figure 4.6 Execution test directory for p-search 108
Figure 4.7 Keyword file dump 108
Figure 4.8 p-search sample execution 109
Figure 4.9 Log file contents post execution 109

Figure 4.10 Execution of p-search running on Ubuntu Linux 12.04 LTS 110
Figure 4.11 Execution of p-search running on iMac 111
Figure 4.12 Diagram of word weighting approach 111
Figure 4.13 Weighted characteristics illustration 112
Figure 4.14 p-search execution with indexing capability 115
Figure 5.1 Downloading Python Image Library for Windows 127
Figure 5.2 Windows installation Wizard for the Python Image Library 128
Figure 5.3 Installing Python Image Library on Ubuntu 12.04 LTS 129
Figure 5.4 Internet Photo Cat.jpg 135
Figure 5.5 Map of GPS coordinates extracted From Cat.jpg 138
Figure 5.6 p-gpsExtractor context diagram 140
Figure 5.7 WingIDE Project Overview 140
Figure 5.8 p-gpsExtractor.py execution 159
Figure 5.9 Mapping the coordinates extracted from photos 160
Figure 5.10 Map zoom into Western Europe 161

List of figures xix

Figure 5.11 Map zoom to street level in Germany 162
Figure 5.12 Snapshot of Results.csv file 162
Figure 5.13 Snapshot of the Forensic Log file 163
Figure 6.1 John Harrison H1 clock 166
Figure 6.2 A very brief history of time 168
Figure 6.3 Python ntplib download page 174
Figure 6.4 Download of ntplib-0.3.1.tar.gz 175
Figure 6.5 Decompressed ntplib-0.3.1 176
Figure 6.6 Install ntplib 176
Figure 6.7 Verifying the installation 177
Figure 6.8 dir(ntplib) results 177
Figure 6.9 Partial list of NIST time servers 178
Figure 6.10 European NTP Pool Project 178
Figure 7.1 NLTK.org Installation url 185
Figure 8.1 Simplest local area network 206
Figure 8.2 Isolated localhost loopback 208
Figure 8.3 server.py/client.py program execution 211
Figure 8.4 Photo of the actual USS Dallas Los Angeles-class nuclear-powered attack submarine 212
Figure 8.5 Command line launch of the guiPing.py as root 218
Figure 8.6 GUI interface for Ping Sweep 218

Figure 8.7 Ping Sweep execution 224
Figure 8.8 Error handling for misconfigured host range 225
Figure 8.9 Port Scanner GUI 228
Figure 8.10 Port Scanner program launch 228
Figure 8.11 Port Scanner execution with Display All selected 234
Figure 8.12 Port Scanner execution with Display NOT selected 234
Figure 9.1 SPAN port diagram 239
Figure 9.2 SPAN port connections 239
Figure 9.3 Raw TCP/IP packet contents 242
Figure 9.4 Typical IPv4 packet header 243
Figure 9.5 Typical TCP packet header 246
Figure 9.6 Typical UDP packet header 248
Figure 9.7 WingIDE environment for the PSNMT application 250
Figure 9.8 psnmt TCP sample run 259
Figure 9.9 psnmt UDP sample run 260
Figure 9.10 Sample TCP output file shown in Excel 261
Figure 9.11 Sample UDP output file shown in Excel 262
Figure 10.1 Plaintext Rainbow Table output abridged 287
Figure 11.1 Typical cloud configuration 290
Figure 11.2 Cloud execution from iPad 291
Figure 11.3 Desktop execution of the simple and multiprocessing
Python applications executing in the cloud 291
Figure 11.4 Python Anywhere Home Page 292
Figure 11.5 Python Anywhere Plans 293

xx List of figures

Figure 11.6 PICloud Home Page 293
Figure 11.7 PICloud Plans 294
Figure 11.8 Digital Ocean Home Page 294
Figure 11.9 Digital Ocean Plans 294
Figure 11.10 Python Anywhere Single Core Execution Results 299
Figure 11.11 Python Anywhere Multi-Core Execution Results 299
Figure 11.12 Standalone Linux Single/Multi-Core Execution Results 299
Figure 11.13 elPassword 8-character combinations of lowercase letters 300
Figure 11.14 elPassword 8-character full ASCII character set 301
Figure 11.15 Last Bit calculation lowercase using 1 computer 301
Figure 11.16 Last Bit calculation lowercase using 100 computers 302
Figure 11.17 Last Bit calculation ASCII set using 100 computers 302

Figure 11.18 Last Bit calculation ASCII set using 10,000 computers 302

Figure 12.1 Multiprocessing in the Cloud 311

Figure 12.2 AMD 6300 Series 16 Core Processor 311

Figure 12.3 Intel Xeon E7 Series 10 Core 20 Thread Processor 311

About the Author

Chet Hosmer is a Founder and Chief Scientist at WetStone Technologies, Inc. Chet has been researching and developing technology and training surrounding forensics, digital investigation, and steganography for over two decades. He has made numerous appearances to discuss emerging cyber threats including National Public Radio's Kojo Nnamdi show, ABC's Primetime Thursday, NHK Japan, Crime Crime TechTV, and ABC News Australia. He has also been a frequent contributor to technical and news stories relating to cyber security and forensics and has been interviewed and quoted by IEEE, The New York Times, The Washington Post, Government Computer News, Salon.com, and Wired Magazine.

Chet also serves as a Visiting Professor at Utica College where he teaches in the Cybersecurity Graduate program. He is also an Adjunct Faculty member at Champlain College in the Masters of Science in Digital Forensic Science Program. Chet delivers keynote and plenary talks on various cyber security related topics around the world each year.

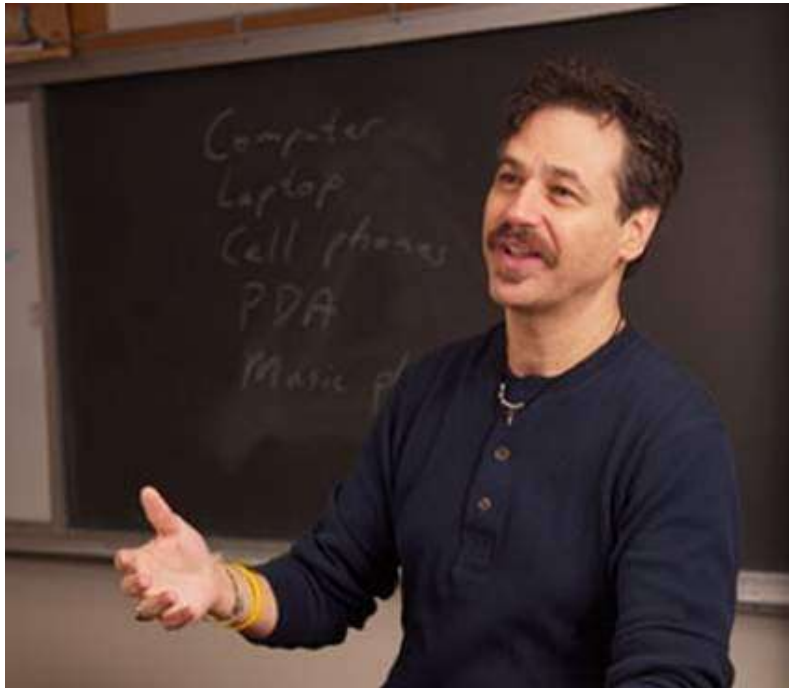


xxi

About the Technical Editor

Gary C. Kessler, Ph.D., CCE, CCFP, CISSP, is an Associate Professor of Homeland Security at Embry-Riddle Aeronautical University, a member of the North Florida Internet Crimes Against Children (ICAC) Task Force, and president and janitor of Gary Kessler Associates, a training and consulting company specializing in computer and network security and digital forensics.

Gary is also a part-time member of the Vermont ICAC. He is the coauthor of two professional texts and over 70 articles, a frequent speaker at regional, national, and international conferences, and past editor-in-chief of the Journal of Digital Forensics, Security and Law. More information about Gary can be found at his Web site, [http:// www.garykessler.net](http://www.garykessler.net).



xxiii

Foreword

On June 16, 2008 a user in the home of 2-year old Caylee Anthony googled for the term fool proof suffocation. A minute later, the same user logged onto the MySpace website with the profile of Casey Anthony. Tragically within months, the police found the decomposed remains of the young girl. Prosecutors charged Casey Anthony with first-degree murder and subsequently tried her 3 years later. The trial lasted 6 months and contained over 400 pieces of unique evidence. Sadly, the details of the computer search never made it to trial. The prosecutor's computer forensic examiner employed a tool to retrieve the browser history results. In use of that tool, the examiner only searched the Internet Explorer browser history and not that of the Firefox browser. The moral of the story is that we are only as good as our tools and our understanding how they work.

In contrast to the failure of the computer forensic examiner, let us consider the most feared military force—The Spartan Army. The strength of the Spartan army relied upon the professionalism of its soldiers. From a young age, the elite warriors learned only one occupation—to wage war. In that profession, they relied heavily on the quality of their weaponry and armor. Instead of weapons being issued to a warrior, it was the responsibility of every Spartan warrior to bring his own weapons and armor to war. Fathers passed these tools to their sons

before entering battle. In the following pages, Chet Hosmer passes down modern tools and weapons. As a forensic investigator, your terrain may include a hard drive's unallocated space instead of the pass of Thermopylae. However, just like the Spartan elders, Chet will teach you to forge your own tools. Building your own weaponry is what separates a forensic examiner that makes the gross mistake of missing browser artifacts versus that of a professional examiner.

The following chapters cover a breadth of topics from hashing, keyword searching, metadata, natural language processing, network analysis, and utilizing cloud multiprocessing. Chet covers this range of exciting topics as he teaches you to build your own weapons in the Python programming language. A visiting professor in the Cyber Security Graduate Program at Utica College, Chet is both an educator and a practitioner. He has served as the principal investigator on over 40 cyber security, digital forensic and information assurance research programs and received international recognition and awards for his work. Like the Spartan elder, his knowledge will hopefully translate to the growth of a new breed of professional. So please enjoy the following pages. And as Spartan wives used to tell their husbands upon entering battle, come back with your shield or on it.

TJ OConnor SANS Red&Blue Team Cyber Guardian xxv

Preface

Over the past 20 years I have had the privilege to work with some of the best, brightest, and dedicated forensic investigators throughout the world. These men and women work tirelessly to find the truth—usually working under less than ideal conditions and under the stress of real deadlines. Whether they are tracking down child predators, criminal organizations, terrorists, or just good old fashion criminals trying to steal your money, these investigators are under the gun and need the best of the best at their fingertips.

I communicate regularly with industry leaders developing the latest forensic products, while evolving their current software baseline to meet the needs of the broadest audience possible. I also communicate with customers trying to solve real-world problems that require immediate answers to hard questions, while the volume of data holding the answer gets larger by the second.

As a scientist and teacher, I see a thirst from students, law enforcement

personnel, and information technology professionals who possess a burning desire, unique investigative skills, an understanding of the problem, and most importantly innovative ideas pertaining to the problems at hand. However, in many cases they lack the core computer science skills necessary to make a direct contribution to the cause.

The Python programming language along with the global environment that supports it offers a path for new innovation. Most importantly the language opens the door for broad inclusion and participation of free tools and technology that can revolutionize the collection, processing, analysis, and reasoning surrounding forensic evidence. This book provides a broad set of examples that are accessible by those with zero or little knowledge of programming, as well as those with solid developer skills that want to explore, jump start, and participate in the expanded use of Python in the forensic domain. I encourage you to participate, share your knowledge, apply your enthusiasm, and help us advance this cause.

INTENDED AUDIENCE

I have written the book to be accessible by anyone who has a desire to learn how to leverage the Python language to forensic and digital investigation problems. I always thought of this as an on-ramp and a beginning that I hope this will inspire you to create something great and share it with the world.

PREREQUISITES

Access to a computer, familiarity with an operating system (Windows, Linux, or Mac) and access to the Internet, coupled with a desire to learn.

xxvii xxviii Preface

READING THIS BOOK

The book is organized with the first two chapters focused on introductory material and setting up the free Python development environment. [Chapters 3 through 11](#) focus on differing problems or challenges within digital investigation, and provide guided solutions along with reference implementations that focus on the core issues presented. I encourage you to use, expand, evolve, and improve the solutions provided. Finally, [Chapter 12](#) looks back and then forward to consider the path ahead.

SUPPORTED PLATFORMS

All the examples in the book are written in Python 2.7.x in order to provide the greatest platform compatibility. The associated web site has solutions for both Python 2.7.x and 3.x whenever possible. As more third party libraries complete support for Python 3.x, all the examples will be available for 2.7.x and 3.x. Most of the examples have been tested on Windows, Linux, and Mac operating systems and will most likely work correctly on other environments that fully support at least Python 2.7.x

DOWNLOAD SOFTWARE

Those purchasing the book will also have access to the source code examples in the book (in both 2.7.x and 3.x—when possible) from the python-forensics.org web site.

COMMENTS, QUESTIONS, AND CONTRIBUTIONS

I encourage you to contribute in a positive way to this initiative. Your questions, comments, and contributions to the source code library at python-forensics.org will make this a resource available to all.

I challenge you all to share your ideas, knowledge, and experience. **CHAPTER**

Why Python Forensics? 1

CHAPTER CONTENTS

Introduction	1
Cybercrime Investigation Challenges	2
How can the Python Programming Environment Help Meet these Challenges?	6
Global Support for Python	6
Open Source and Platform Independence	8
Lifecycle Positioning	8
Cost and Barriers to Entry	8
Python and the Daubert Evidence Standard	9
Organization of the Book	10
Chapter Review	

.....	10
Summary Questions	
.....	11
Additional Resources	
.....	11

INTRODUCTION

The Python programming language and environment has proven to be easy to learn and use and is adaptable to virtually any domain or challenge problem. Companies like Google, Dropbox, Disney, Industrial Light and Magic, and YouTube just to mention a handful are using Python within their operations. Additionally, organizations like NASA's Jet Propulsion Lab; the National Weather Service; The Swedish Meteorological and Hydrological Institute (SMHI); and Lawrence Livermore National Laboratories rely on Python to build models, make predictions, run experiments, and control critical operational systems.

Before diving straight in, I am sure you would like a little more information about what I will be covering and how a programming environment like Python matches up with digital investigations. Also, you might be interested to know what you will be learning about, generally what the scope of this book is, and how you can apply the concepts and practical examples presented.

The primary purpose and scope of the book is to show you how Python can be used to address problems and challenges within the cybercrime and digital investigation domain. I will be doing this by using real examples and providing the full source code along with detailed explanations. Thus the book will become a set of reference implementations, a cookbook of sorts, and at the end of the day, will hopefully get you involved in developing your own Python forensic applications.

Python Forensics ¹ © 2014 Elsevier Inc. All rights reserved.

I will be presenting the material without any preconceived notion about your programming expertise (or lack thereof). I only expect that you have an interest in using the examples in the book, expanding on them, or developing derivatives that will fit your situation and challenge problems. On the other hand, this is not a how to programming book, many of those exist for Python along with a

plethora of online resources.

So, let us get started by defining just some of the challenges we face in cybercrime and digital investigation. These challenges after all were the catalyst behind the book and have come from the past two decades of working on solutions to assist law enforcement; defense and corporate entities collect and analyze digital evidence.

CYBERCRIME INVESTIGATION CHALLENGES

Some of the challenge problems that we face in cybercrime investigation include: The changing nature of investigations: Much of the work over the past two decades has focused on the postmortem acquisition, search, format, and display of information contained on various types of media. I can clearly remember the phone call I received almost two decades ago from Ron Stevens and Tom Hurbanek at the New York State Police. They were investigating a case that involved a Linux computer and were quite concerned about files and other data that might have been deleted that could be impeding the investigation. At that point no technology existed to extract deleted files or fragments that were buried away inside deleted Linux inodes, although several solutions existed for the Windows platform at the time. We worked together to develop algorithms that eventually became a tool named “extractor” that we provided free to law enforcement. The move from simply extracting data, recovering deleted files, and scouring unallocated or slack space from computers has rapidly shifted just in the last couple of years. Today we focus most of our attention on smart mobile devices, dynamically changing memory, cloud applications, real-time network forensics, automotive data analysis, and weather-based forensics, just to mention a few. In addition, new work is addressing the association of direct digital forensic evidence with a broad range of

instantly available electronic information. Whether this information comes from text

messages, Facebook posts, tweets, LinkedIn associations, metadata embedded in digital photographs or movies, GPS data that tracks our movements or the digital fingerprints left from every Web site we surf, all may be relevant and used in civil

or criminal cases. The question is how do we connect these dots while maintaining

forensic efficacy?

The widening gap between technology developers and investigators:

Investigators, examiners, incident response personnel, auditors, compliance experts tend to

come into this field with a background in social science, whereas technology developers tend to have backgrounds in computer science and engineering.

Clearly, there are some excellent examples of crossovers in both directions, but the vocabulary, thought process, and approach to problem solving can be quite different. Our goal, as depicted in [Figure 1.1](#), is to leverage Python forensic solutions to close that gap and create a collaborative nonthreatening environment whereby computer

science and social science can come together.

The challenge is to develop a platform, vernacular environment where both social

scientists and computer scientists can comfortably communicate and equally participate in the process of developing new forensic solutions. As you will see, the Python

environment provides a level playing field, or common ground at least, where new

innovations and thought can emerge. This has already shown to be true in other scientific fields like Space Flight, Meteorology, Hydrology, Simulation, Internet Technology advancement, and Experimentation. Python is already providing valuable contributions in these domains.

Cost and availability of new tools: With a couple of exceptions (for example, EnCase[®] App Central), most new innovations and capabilities that come through vendor channels take time to develop and can add significant cost to the investigator's toolkit. In the past, investigators carried with them just a handful of hardware

and software tools that they used to extract and preserve digital evidence. Today, to

address the wide range of situations they may encounter, 30-40 software

products

may be necessary just to perform acquisition and rudimentary analysis of the digital

crime scene. Of course this is just the start of the investigative process and the number and variety of analytic tools continues to grow.

The true cost and cost of ownership of these technologies can be staggering, especially when you factor in education and training. The barrier to entry into the

field can easily reach high five or even six figures. This is in a field where backlogs

continue to grow at law enforcement agencies around the world. Backlogs are also

growing within the corporate sector, which is dealing with human resource actions,

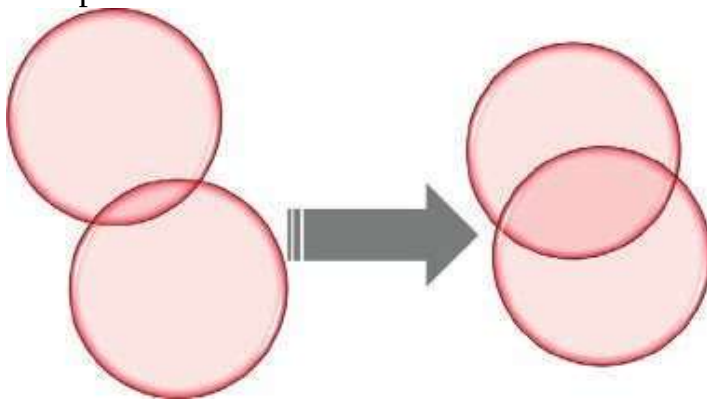
corporate espionage, insider leaks, and massive amounts of regulatory requirements.

Social science Social science

Python forensics Computer

science

Computer science



Today Future FIGURE 1.1
Narrowing the gap.

GPS location into mapped data. These GPS data also include timestamps that allows us to place a device at a particular location at a specific date and time.

The next-generation investigator :We must generate interest in this field of study to attract the best and brightest to a career in cybercrime investigation. In order to do so, this new breed needs to not only use tools and technology but also must play an important role in researching, defining, evaluating, and even developing some of these high demand and sophisticated next-generation capabilities (Figure 1.4).

Lack of a collaboration environment: Cybercriminals have the advantage of untethered collaboration, along with access to resources that assist in the execution

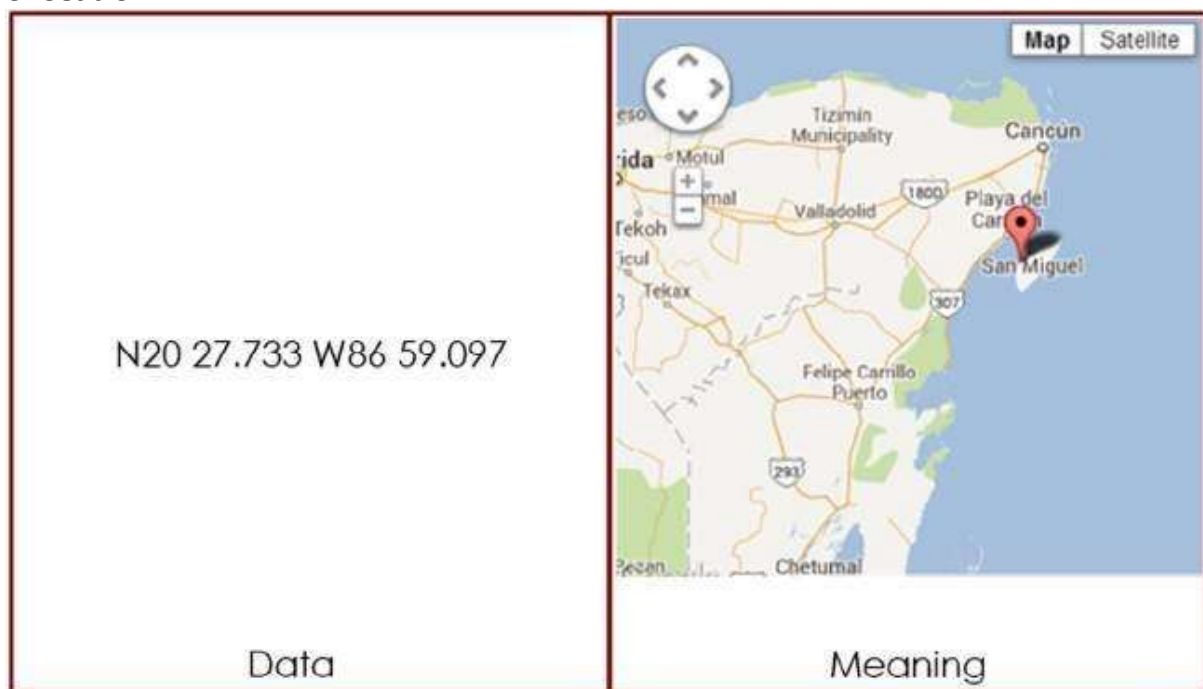


FIGURE 1.3
Data vs. semantics.



FIGURE 1.4
The next-generation cyber warrior.

of distributed attacks and sophisticated cybercrime activities. Investigators and developers of new methods and techniques need that same advantage. They require a platform for collaboration, joint development, and access to new innovations that could be directly applied to the situation at hand.

HOW CAN THE PYTHON PROGRAMMING ENVIRONMENT HELP MEET THESE CHALLENGES?

Creating an environment where social and computer scientists can collaborate and work together is challenging. Creating a platform to develop new technology-based solutions that address the broad range of digital investigation challenges outlined earlier in this chapter is difficult. Doing them both together is a real challenge, and whenever you take on a challenge like this it is important to consider the underpinnings so that you create the best chance for success.

I personally have a few important considerations that have served me well over the years, so I will share them with you here.

1. Does the platform you are building on have broad industry support?
2. Is there an ample supply of technical data regarding the subject along with a large cadre of talent or subject matter experts?
3. Is the technology platform you are considering open or closed?
4. Where does the technology exist along its lifecycle (i.e., too early, too late or mature, and evolving)?
5. What is the cost or other barriers to entry? (especially if you are trying to attract a broad array of participants)
6. Finally, since we are trying to bridge the gap between social and computer science, is the environment well suited for cross-disciplinary collaboration?

Global support for Python

Python was created by Guido van Rossum in the late 1980s with the fundamental premise that Python is programming for everyone. This has created a groundswell of support from a broad array of domain-specific researchers, the general software development community, and programmers with varying backgrounds and abilities. The Python language is both general purpose and produces easily readable code that can be understood by nonprogrammers. In addition, due to Python's intrinsic extensibility, copious amount of third-party libraries and modules exist. Numerous web sites provide tips, tricks, coding examples, and training for those needing a deeper dive into the language. It may surprise you that Python was ranked as the number 1 programming language in 2013 by codeeval.com (see [Figure 1.5](#)) edging out Java for the first time. A great place to start is at the official Python programming language web sites python.org.

Finally, sophisticated integrated software development environments exist that allow even the novice developer to innovate new ideas and design, and then build

Python Programming Environment 7

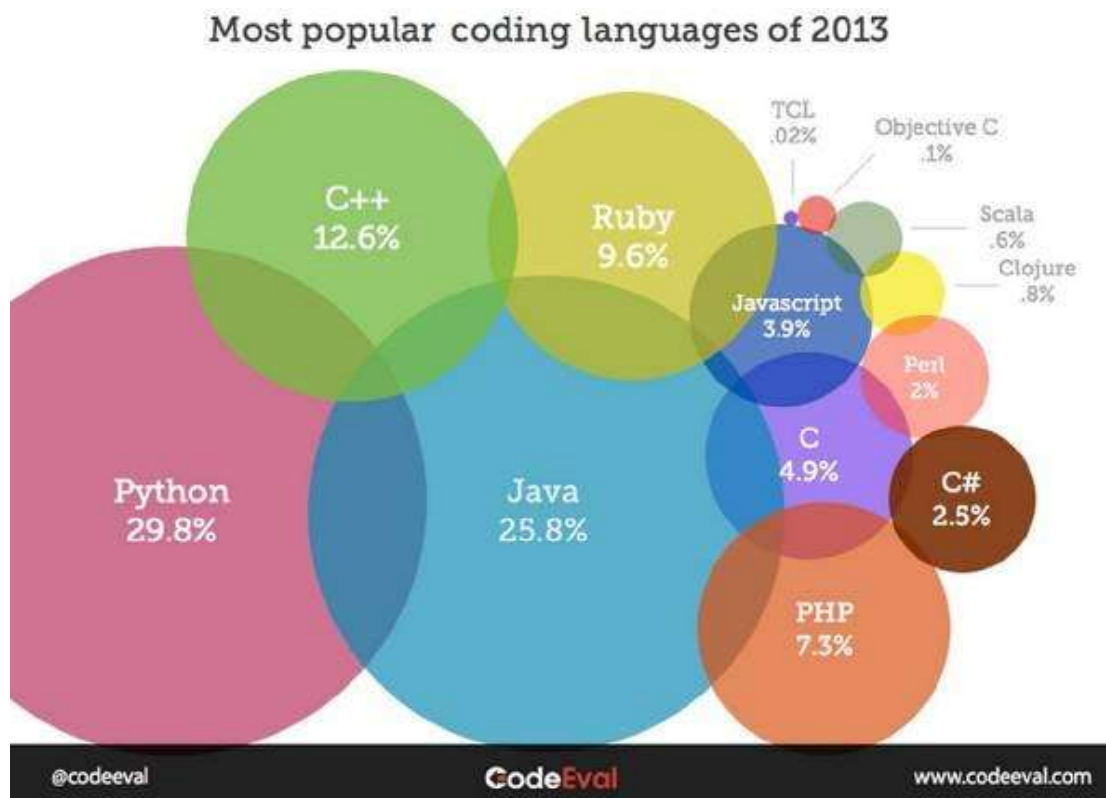


FIGURE 1.5

Programming language popularity according to codeview.com.

and test their inventions and prototypes. Python is an interpreted language; however, compilers are available as well, of course. As shown in [Figure 1.6](#), developers have adopted the test-then code-then validate mindset.

By utilizing the Python Shell, experienced and novice users alike can experiment with the language, libraries, modules, and data structures before attempting to integrate them into a complete program or application. This promotes experimentation with the language, language constructs, objects, experimentation with performance considerations, and libraries, and allows users to explore and tradeoff approaches

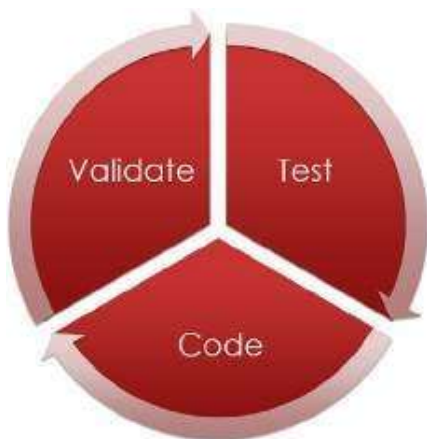


FIGURE 1.6

Test-then code-then validate.

prior to putting them into practice. Once confident with the use, characteristics and behaviors of the language, the integration into working programs tends to go more smoothly. In addition, this experimentation often leads to testing considerations that can then be applied to the working programs once they have been completed, thus completing the cycle of test-code-validate.

Open source and platform independence

Since Python is an open source environment, developers continue to create compatible versions that run across multiple platforms. Python implementations exist for today's most popular platforms including Windows, Linux, and Mac OS X as you would expect. However, the support for Python is much broader. Support for mobile device operating systems such as Android, iOS, and Windows 8 is also available. In addition, Python is supported on platforms that you might not expect such as AIX, AS/400, VMS, Solaris, PalmOS, OS/2, and HP-UX, just to mention a few. What this means for cybercrime investigators is portability of these applications for today's platforms, yesterday's platforms, and future platforms.

In March 2013, NVIDIA announced support for Python developers by opening the door to GPU-Accelerated Computing using NVIDIA CUDA, allowing for parallel processing capabilities, and advancing the performance of virtually any Python-developed application. This will deliver the ability to handle and process big data, perform advanced analytical operations, perform deductive and inductive reasoning, and meet future computational challenges. This versatility ensures that the investment made in creating new investigative solutions will be useable and sharable with colleagues that use different flavors of computing platforms.

Lifecycle positioning

Python today sits in the best position possible to be leveraged for investigative applications. The language is mature, hundreds of thousands of developers are experienced, a strong support organization is in place, extensible libraries are plentiful, applications are portable to a wide range of computing platforms, the source code is open and free, and new innovations to the core language are keeping pace with hardware and operating system advancements.

Cost and barriers to entry

One of the keys to Python's success is the lack of virtually any barrier to entry. The development environment is free, the language is platform independent, the code is as easy to read and write as English, and support is vast and worldwide. In my opinion, it should revolutionize the development of new cybercrime, forensic, and incident response-based solutions. The key is to develop outreach that will encourage, attract, and open the doors to social scientists, computer scientists, law enforcement organizations, forensic labs, standards bodies, incident response teams, academics,

Python and the Daubert Evidence Standard 9

students, and virtually anyone with domain expertise within the broadest definition of cybercrime.

PYTHON AND THE DAUBERT EVIDENCE STANDARD

As many of us have encountered, the Daubert standard provides rules of evidence at the U.S. Federal level along with about one-third of the states that deal with the admissibility of expert testimony including scientific data. Digital data collected and analyzed with forensic software employed by an "expert" can be challenged by using, in laymen's terms, a Daubert motion to suppress or challenge the efficacy of the expert and/or the evidence produced by technology utilized.

In 2003, Brian Carrier [Carrier] published a paper that examined rules of evidence standards including Daubert, and compared and contrasted the open source and closed source forensic tools. One of his key conclusions was, "Using the guidelines of the Daubert tests, we have shown that open source tools may more clearly and comprehensively meet the guideline requirements than would closed source tools."

The results are not automatic of course, just because the source is open. Rather,

specific steps must be followed regarding design, development, and validation.

1. Can the program or algorithm be explained? This explanation should be explained in words, not only in code.
2. Has enough information been provided such that thorough tests can be developed to test the program?
3. Have error rates been calculated and validated independently?
4. Has the program been studied and peer reviewed?
5. Has the program been generally accepted by the community?

The real question is how can Python-developed forensic programs meet these standards? [Chapters 3–11](#) comprise the cookbook portion of the book and each example attempts to address the Daubert standard by including:

1. Definition of the Challenge Problem
2. Requirements Definition
3. Test Set Development
4. Design Alternative and Decisions
5. Algorithm Description (English readable)
6. Code Development and Walk-Through
7. Test and Validation Process
8. Error Rate Calculation
9. Community Involvement

This approach will provide two distinct advantages. First, the cookbook examples provided in the text will be useable out of the box and should meet or exceed the rules of evidence standards. Second, the process defined will assist both experienced and novice developers with an approach to developing digital investigation and forensic solutions. The cookbook examples then provide a model or reference implementations using Python that are designed as an educational and practical example.

ORGANIZATION OF THE BOOK

In order to support the broadest audience of potential contributors to new cybercrime investigative technologies, I have arranged the book to be accessible to those with little or no programming experience, as well as for those that want to dive directly into some of the more advanced cookbook solutions.

[Chapter 2](#) will provide a walk-through for those wishing to setup a Python

software environment for the first time. This step-by-step chapter will include environments for Linux and Windows platforms and will include considerations for Python 2.x and Python 3.x. I will also cover the installation and setup of high-quality thirdparty libraries that I will leverage throughout the book, along with integrated development environments that will make it easier to master Python and manage your projects.

[Chapter 3](#) covers the development of a fundamental Python application that will outline one of the most common digital investigation applications—File Hashing. A broad set of one-way hash algorithms that are directly implemented within the core Python distributions will be covered. I will then show how this simple application can be transformed into a more sophisticated cyber security and investigation tool that could be applied immediately.

Each of the [Chapters 4–11](#) tackles a unique cybercrime investigation challenge and delivers a Python cookbook solution that can be freely used, shared, and evolved, and includes opportunities for you to participate in the future expansion.

[Chapter 12](#) takes a look at future opportunities for the application of Python within cybercrime investigation, a broader set of cyber security applications, and examines high-performance hardware acceleration and embedded solutions.

Finally, each chapter includes a summary of topics covered, challenge problems, and review questions making the book suitable for use in college and university academic environments.

CHAPTER REVIEW

In this chapter, we took a look at the challenges that are facing cybercrime investigators, incident response personnel, and forensic examiners that are dealing with a plethora of digital evidence from a multitude of sources. This chapter also discussed the computer science and social science gap that exists between the users of forensic/ investigative technologies and the developers of the current solutions. We examined

Additional Resources 11

the key characteristics of the Python programming environment which make it well suited to address these challenges. These characteristics include the open

source nature of the Python environment, the platform independent operational model, the global support and available technical data, and the current lifecycle positioning. Also, Python-developed solutions (provided they are done right) will meet or exceed the Daubert rules of evidence requirements. Finally, the organization of the book was discussed to give readers a better understanding of what to expect in upcoming chapters.

SUMMARY QUESTIONS

1. What are some of the key challenges that face forensic investigators today and what potential impacts could these challenges pose in the future?
2. From what has been presented or based on your own research or experience, what do you believe are the key benefits that Python could bring to forensic investigators?
3. What other organizations are using Python today for scientific endeavors and how is the use of Python impacting their work?
4. What other software languages or platforms can you think of that are open source, cross-platform, have global support, have a low barrier to entry, are easily understood, and could be used to collaborate among both computer scientists and social scientists?
5. What forensic or investigative applications can you think of that either do not currently exist or are too expensive for you to buy into?

Additional Resources

Open Source Digital Forensic Tools—The Legal Argument.

DigitalEvidence.org, [http:// www.digital-evidence.org/papers/opensrc_legal.pdf](http://www.digital-evidence.org/papers/opensrc_legal.pdf); 2003.

Python Programming Language—Official Website. [Python.org](http://www.python.org), <http://www.python.org>.

Basu S. Perl vs Python: why the debate is meaningless. The ByeBaker Web site, <http://bytebaker.com/2007/01/29/perl-vs-python-why-the-debate-is-meaningless/>; 2007 [29.01.07].

Raymond E. Why Python? The Linux Journal 73, <http://www.linuxjournal.com/article/3882>; 2000 [30.04.03].

CHAPTER

Setting up a Python Forensics Environment 2

CHAPTER CONTENTS

Introduction	14
Setting up a Python Forensics Environment	14
The Right Environment	15
The Python Shell	16
Choosing a Python Version	16
Installing Python on Windows	17
Python Packages and Modules	24
The Python Standard Library	25
What is Included in the Standard Library?	27
Built-in Functions	27
hex() and bin()	27
range()	28
Other Built-in Functions	30
Built-in Constants	31
Built-in Types	32
Built-in Exceptions	33
File and Directory Access	34
Data Compression and Archiving	34
File Formats	35
Cryptographic Services	35
Operating System Services	35
Standard Library Summary	36
Third-Party Packages and Modules	36
The Natural Language Toolkit [NLTK]	36
Twisted Matrix [TWISTED]	37
Integrated Development Environments	37
What are the Options?	37
IDLE	38

WingIDE	39
Python Running on Ubuntu Linux	42
Python on Mobile Devices	46
iOS Python App	46
Windows 8 Phone	49
A Virtual Machine	51
Chapter Review	51
Summary Questions	51
Looking Ahead	51
Additional Resources	52

Python Forensics 13 © 2014 Elsevier Inc. All rights reserved.

INTRODUCTION

A couple of decades ago, I was working for a large defense contractor and was part of a team that was developing a secure embedded device. My initial role was to setup a development environment for the team to use. This may seem like a pretty simple task. However, the embedded security hardware was completed and handed off to me, but the device itself did not include an operating system or supporting libraries—it was basically an open slate. Thus, the first painstaking task was to develop a boot loader that would allow me to load a program onto the device. Once that was completed, I needed to develop the ability to interface with the board and load additional software (the operating system, shared libraries, applications, etc.), that would bring the security hardware contained within the device online.

This interface software needed to include a debugger that would allow us to control the operating system and application software being developed by the team while it was running on the device. For example, the ability to start the program, stop the program, inspect variables, registers, etc., perform single steps, and set breakpoints in the code, all this was accomplished through a 19,200 baud RS232 interface.

You might be asking, what does this twentieth century example have to do with Python? The answer is simple, the requirements for a stable feature-rich development environment still exist in the twenty-first century, but now we have

modern tools. Without the proper development environment, the chance of successfully developing high-quality, function-rich forensic or digital investigation software is quite low.

SETTING UP A PYTHON FORENSICS ENVIRONMENT

Many considerations exist before setting up an environment. I will explore some of the key areas that I myself consider when setting up an environment like this. Here are some important considerations:

1. What is the right environment for your situation? Are you a professional software developer; a novice developer with solid investigative skills and ideas that you would like to explore; do you work in a forensic lab or support an incident response team with new tools and methods; or maybe you might work in the IT Security group at a corporation and need better ways to collect and analyze what is happening on your network.
2. How do you choose the right third-party libraries and modules that enhance your program and allow you to focus on your application and not on reinventing the wheel?

The Right Environment 15

3. What is the right integrated development environment (IDE) and what capabilities should be included in that environment?
 - a. Code intelligence that provides automatic completion, built-in error indicators source browser, code indices, and fast symbol lookup.
 - b. A robust graphical debugger that allows you to set breakpoints, single-step through code, view data, and examine variables.
 - c. A powerful programmer's editor that has a complete understanding of the Python language rules, advanced search tools, bookmarking, and code highlighting.
 - d. Cross-platform support such that you can choose your platform and environment, (i.e., Windows, Linux, or Mac), and select any Python version from 2.x to 3.x and Stackless Python.

Stackless Python is a relatively new concept, allowing Python programs to execute which are not limited by the size of C Stack. In many environments, the size of stack memory is limited in comparison to the amount of heap memory.

Stackless Python utilizes the heap and not the stack which provides greater distributed processing possibilities. For example, this allows for thousands of independently running tasklets to be launched. Online multiuser gaming platforms use this to support thousands of simultaneous users. I am sure if you are thinking ahead you can imagine some interesting digital investigation and forensic applications for such an environment.

e. A unit testing capability that enables you to thoroughly validate your code within common testing frameworks such as unittest, doctest, and nose.

f. Built-in revision control for advanced projects along with direct integration with popular revision control systems such as Mercurial, Bazaar, Git, CVS, and Perforce. When building larger applications that contain many moving components managing the many revisions becomes important. This selection will ultimately determine if the tools and applications developed will meet the standards of quality that are essential ingredients for digital investigation and forensic applications.

It is important right up front to realize this book is about building forensic applications that must meet the Daubert standard, thus we are not just hacking out some code that will work most of the time, but rather code that should work all the time or fail gracefully. And most importantly, we need to develop code that will create admissible evidence.

THE RIGHT ENVIRONMENT

One of the initial choices you will have to make is the platform you intend to use for the development of Python forensic applications. As discussed in [Chapter 1](#), Python and Python programs will execute on a variety of platforms including the latest desktop, mobile, and even legacy systems. However, that does not mean you need to develop your applications on one of those platforms. Instead you will most likely be developing your applications on a Windows, Linux, or Mac platform that supports the latest development tools.

One great thing about Python ... If you follow the rules, develop quality programs and make sure you consider cross-platform idiosyncrasies, no matter what platform you choose for development, the resulting Python programs you create should easily run on a variety of operating systems that have a properly installed Python system.

The Python Shell

The Python Shell delivers an object-oriented, high-level programming language that includes built-in data structures and an extensive Standard Library. Python employs a simple easy to use syntax that virtually anyone can learn, provides support for thirdparty modules and packages which encourages program code reuse and sharing, and is supported on virtually all major platforms and can be freely distributed. What that delivers to the digital investigator or forensic specialist is the ability to quickly develop programs that will augment or even replace current tools and then immediately share them with the community. One of the other benefits of the interpreted environment is the ability to experiment with the Standard Library, third-party modules, commands, functions, and packages without first developing a program. This allows you to ensure that the commands, functions, and modules you are planning to use provide the results and performance you are looking for. Since there are many

options and sources for these functions with more arriving on the scene every day, the interpreter allows you to easily experiment before you commit to a final approach.

CHOOSING A PYTHON VERSION

As with any programming environment, many currently supported versions of Python are available. However, two basic standards of Python exist today in versions 2.x and 3.x. The change from Python 2.x to 3.x has caused some difficulties in portability, and programs and libraries written for 2.x require modification to work within 3.x.

Some of the core Python functions have changed mainly due to full support of Unicode in version 3.x. This not only affects programs but also affects previous modules that have been developed and have not yet been ported and or validated for Python 3.x. Based on this conundrum, I have decided to develop the examples supplied in this book to conform to the 2.x standard, giving us access to the broadest set of third-party modules and compatibility across more platforms. The source code in the book will be available online as well, whenever possible I will provide both a 2.x and 3.x version of the source.

In addition, the 2.x class of Python has been proven, validated, and deployed in a broad set of applications. Therefore, developing forensic or digital investigation applications using the 2.x version provides us with a solid underpinning and broadest deployment platform. Also, once 3.x becomes broadly embraced and

the third-party libraries become available and certified, we will have all the information necessary to port and even enhance the applications in this book.

Now that we have selected a version to begin with, I will walk you through setting up Python on a Windows desktop.

INSTALLING PYTHON ON WINDOWS

If you Google “Python Installation” you will get a little over 7 million page hits, as of this writing. In my opinion, the best and safest place to obtain tested standard Python installations is at www.python.org, which is the Python Programming Language Official Web site [PYTHON], shown in [Figure 2.1](#). At this page, I selected Python version 2.7.5.

Next, I navigate to the download page and select the appropriate version for my situation. In this case I am going to select:

Python 2.7.5 Windows x86 MSI installer 2.7.5 (sig)

This will download the Windows runtime environment (as shown in [Figure 2.2](#)). Also shown is the hash of the download file allowing you to validate the download.

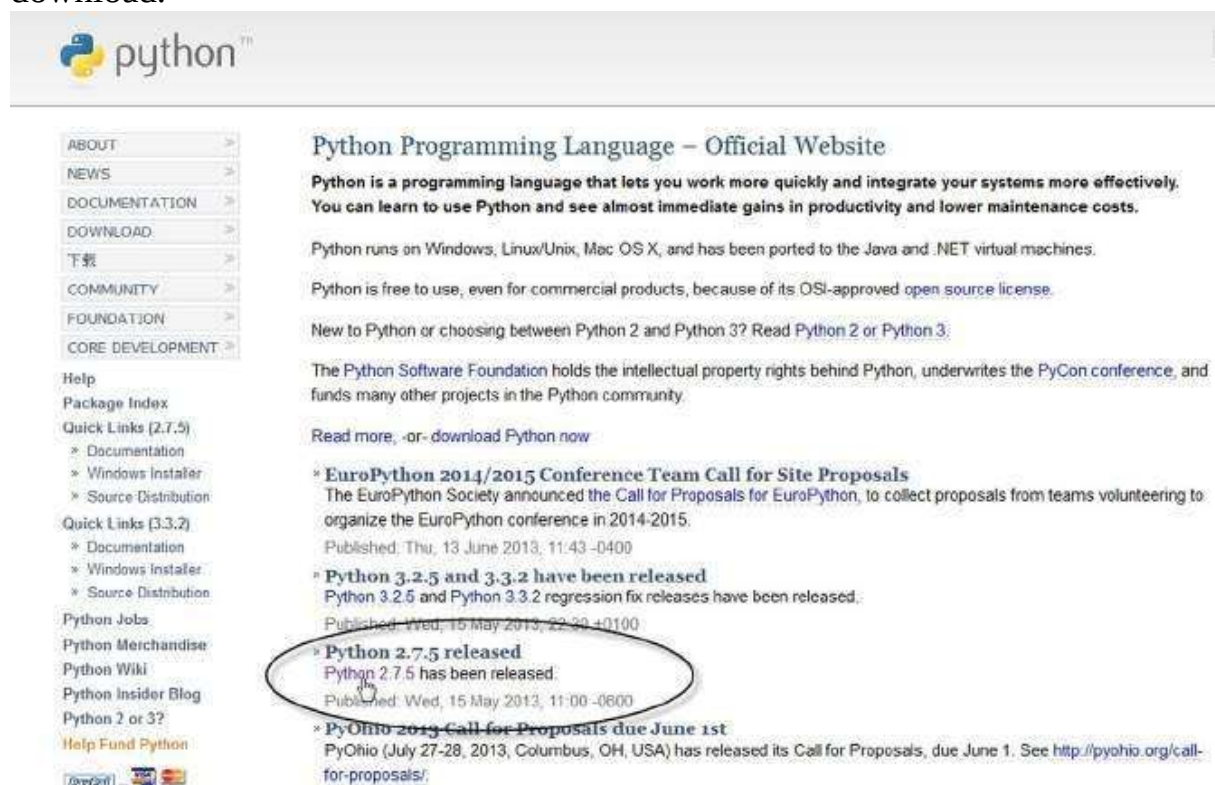


FIGURE 2.1

The Python Programming Language Official Web site.

This is a production release. Please report any bugs you encounter.

We currently support these formats for download:

- XZ compressed source tar ball (2.7.5) (sig)
- Gzipped source tar ball (2.7.5) (sig)
- Bzipped source tar ball (2.7.5) (sig)
- Windows x86 MSI Installer (2.7.5) (sig)
- Windows x86 MSI program database (2.7.5) (sig)
- Windows x86-64 MSI Installer (2.7.5) [1] (sig)
- Windows x86-64 program database (2.7.5) [1] (sig)
- Mac OS X 64-bit/32-bit x86-64/386 Installer (2.7.5) for Mac OS X 10.6 and later [2] (sig). [You may need an updated Tcl/Tk install to run IDLE or use Tkinter, see note 2 for instructions.]
- Mac OS X 32-bit i386/PPC Installer (2.7.5) for Mac OS X 10.3 and later [2] (sig).

The source tarballs are signed with Benjamin Peterson's key (fingerprint: 12EF 3DC3 8047 DA38 2D18 A5B9 99CD EA9D A413 5B38). The Windows installer was signed by Martin von Löwis' public key, which has a key id of 7D9DC8D2. The Mac installers were signed with Ned Deily's key, which has a key id of 6F5E1540. The public keys are located on the download page.

MD5 checksums and sizes of the released files:

b4f01a1d0ba0b46b05c73b2ec909b1df	14492759	Python-2.7.5.tgz
6334b666b7ff2038c761d7b27ba699c1	12147710	Python-2.7.5.tar.bz2
5eea8462f69ab1369d32f9c4cd6272ab	10252148	Python-2.7.5.tar.xz
e632ba7c34b922e4485667e332096999	18236482	python-2.7.5-pdb.zip
55cc56948dccc3afb53f65d8bb425f20	17556546	python-2.7.5.amd64-pdb.zip
83f5d9ba639bd2e33d104df9ea969f31	16617472	python-2.7.5.amd64.msi
0006d6219160ce6abe711a71c835ebb0	16228352	python-2.7.5.msi
ead4f83ec7823325ae287295193644a7	20395084	python-2.7.5-macosx10.3.dmg
248ec7d77220ec6c770a23d3cb537bc	19979778	python-2.7.5-macosx10.6.dmg

[1] (1, 2) The binaries for AMD64 will also work on processors that implement the Intel 64 architecture (formerly EM64T), i.e. the architecture that Microsoft calls x64, and AMD called x86-64 before calling it AMD64. They will not work on Intel Itanium Processors (formerly IA-64).

[2] (1, 2) There is important information about IDLE, Tkinter, and Tcl/Tk on Mac OS X here.

FIGURE 2.2

Downloading the Windows installer.

As you would expect, selecting this link presents you with a Windows dialog box as shown in Figure 2.3, confirming that you wish to save the installation file. Selecting OK will download the file and save it in your default download directory.

Examining the contents of my download directory, you can see that I have actually downloaded both the latest 2.x and 3.x releases. I am now going to select and execute the 2.7.5 Installer (Figure 2.4).

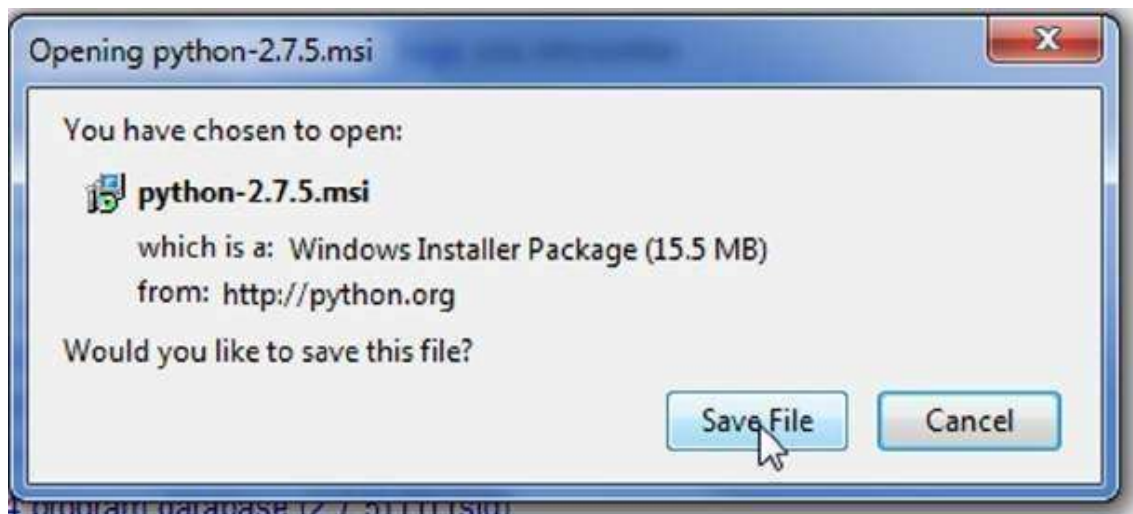


FIGURE 2.3
Windows download confirmation.

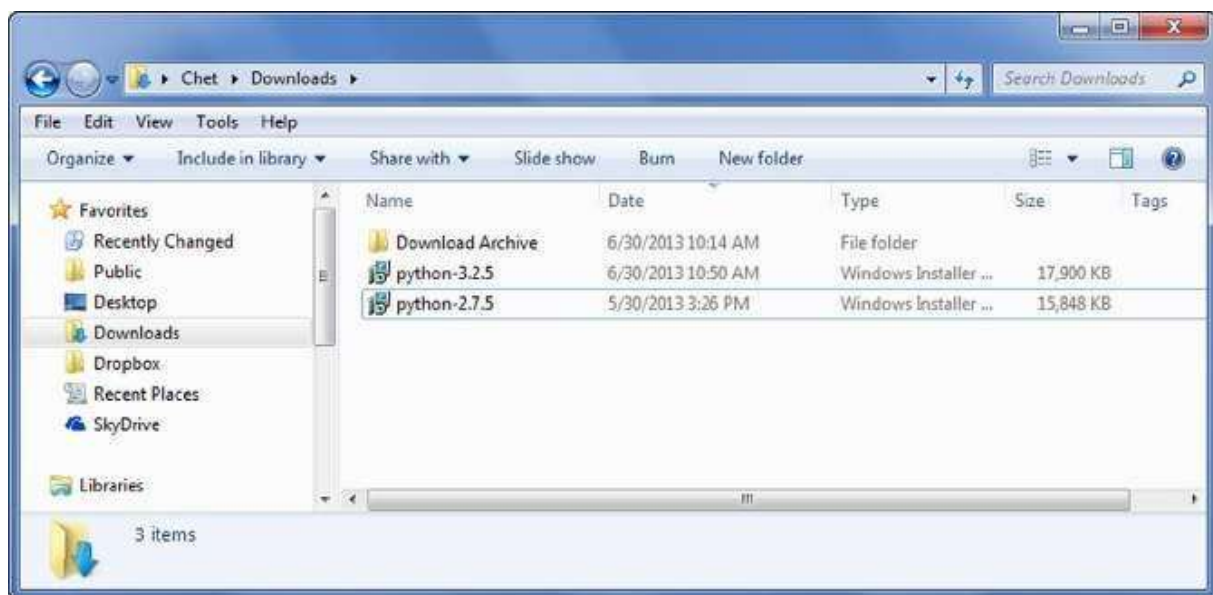


FIGURE 2.4
Executing the Python 2.7.5 Installer.

Upon execution, the Installer prompts you to determine whether Python should be installed for the current user or for all users on this machine. This is your choice of course, but if you choose to install this for every user make sure they are trusted and know the risks. The Python environment runs at a high privilege level and has access to operating system functions that not everyone should necessarily need (Figure 2.5).

Python allows you to select where the environment will be installed. The default location for Python 2.7.5 is C:\Python27, although you can specify another location ([Figure 2.6](#)). You should make a note of this so you can inspect and reference the directories and files stored there. This will be especially helpful later when you are looking for certain libraries, modules, tools, and documentation.

I have decided to customize my installation slightly. I want to ensure that the Python documentation is stored on my local hard drive so I can have access anytime. Also, I do not expect to need the TCL/TK Graphical User Interface (GUI) module, so I have selected this to only be installed when needed; this will reduce the size of the installation. If you have no restriction on disk storage, including this with your install is not an issue. We will be installing other easier to use GUI packages and modules later in the book ([Figures 2.7 and 2.8](#)).

Next, Windows will display the typical User Account Control (UAC) to help you stay in control of your computer by informing you when a program makes a change that requires administrator-level permission ([Figure 2.9](#)). Windows will also verify the digital signature of Python and display the name of the organization associated with the certificate used to sign the installer. Selecting Yes will install Python and allow the appropriate system changes to be made, provided that the user has administrative privilege. You might notice that this screenshot was taken with my iPhone because during UAC acknowledgment Windows locks out all other program access until either Yes or No is selected.



FIGURE 2.5

Python Installation user selection.

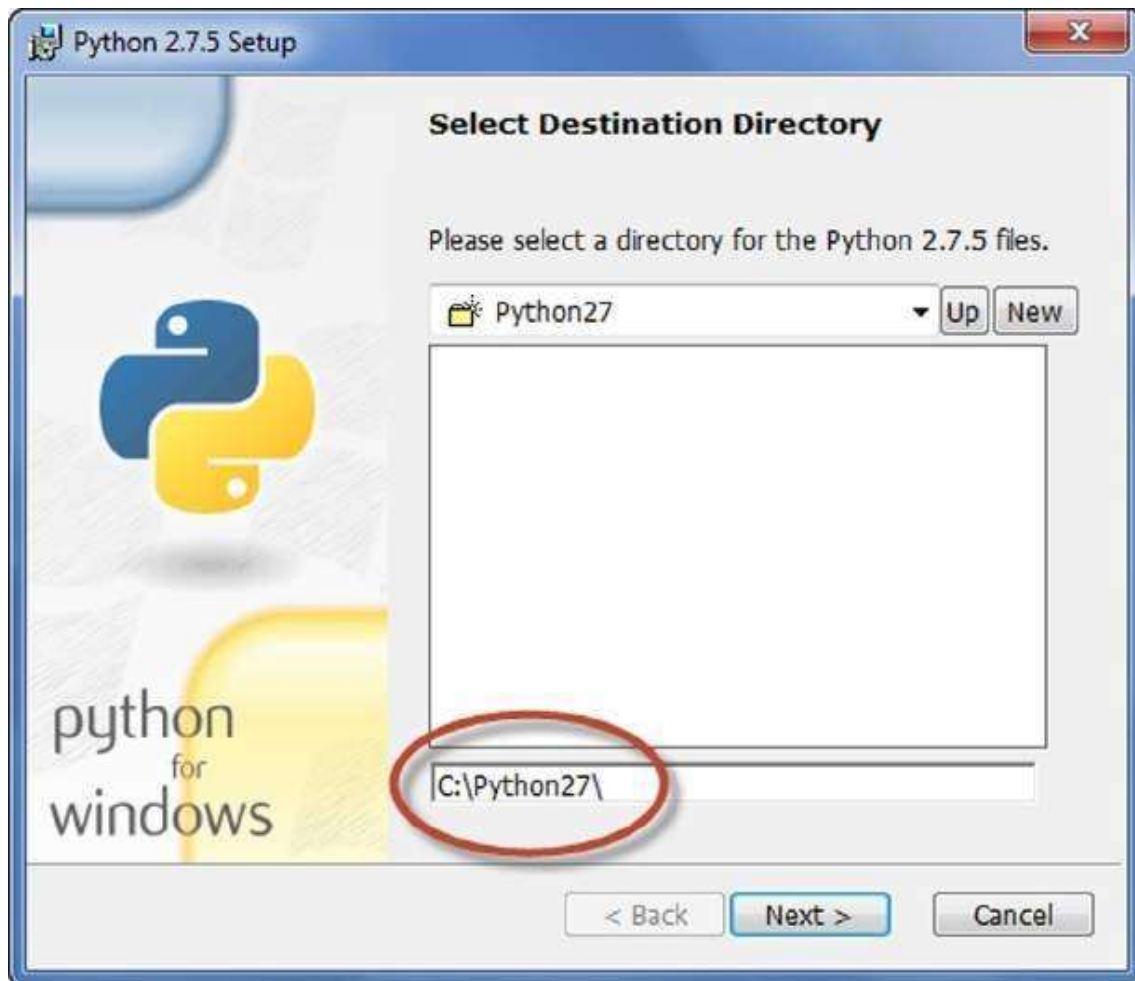


FIGURE 2.6
Python Installation directory.



FIGURE 2.7
Python customization user manual.

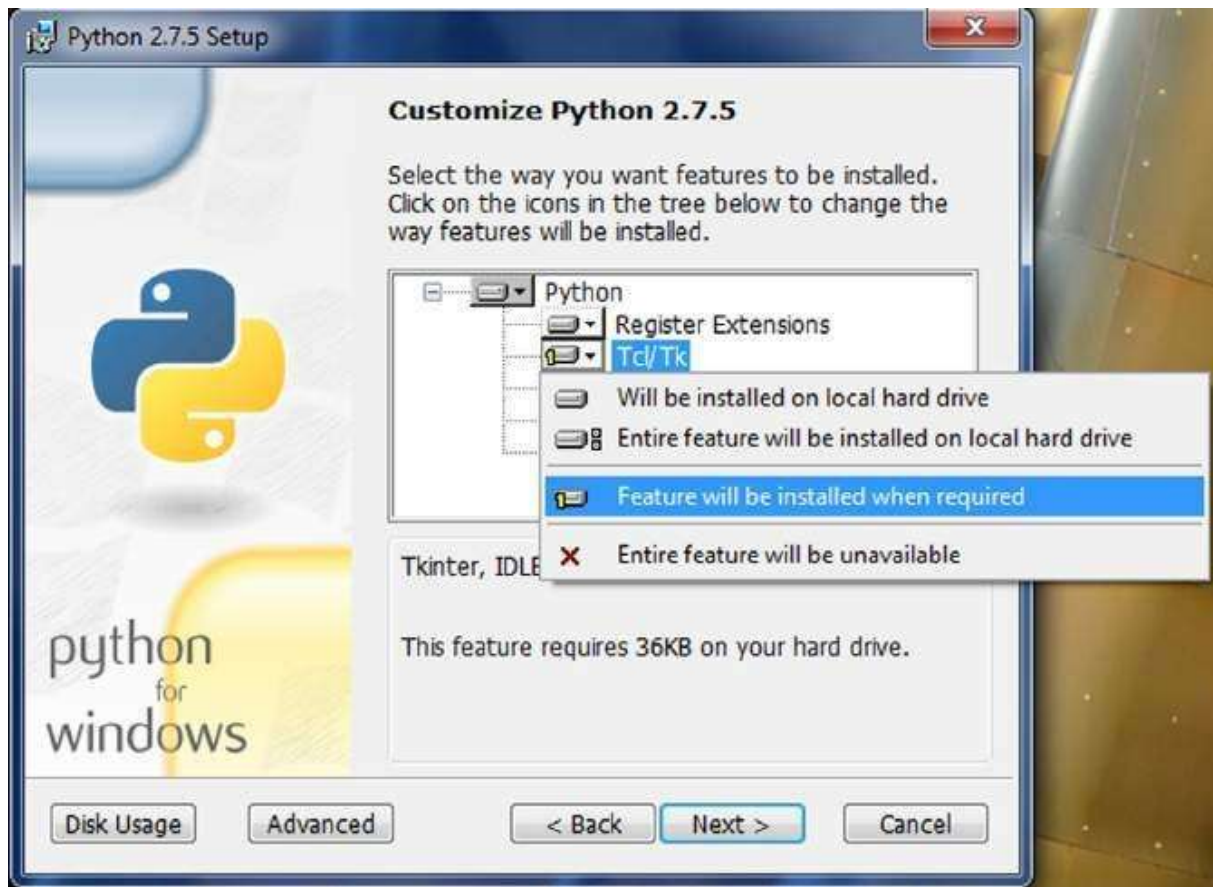


FIGURE 2.8
TCL/TK install when needed.



FIGURE 2.9
Windows user account control.

Finally, Python has been installed and displays the successful completion of the installation as shown in [Figure 2.10](#).

We can now navigate to the C:\Python27 directory (or the path name you specified for Python installation) and examine the contents. As you can see in [Figure 2.11](#),



FIGURE 2.10
Successful installation of Python 2.7.5.
Installing Python on Windows 23

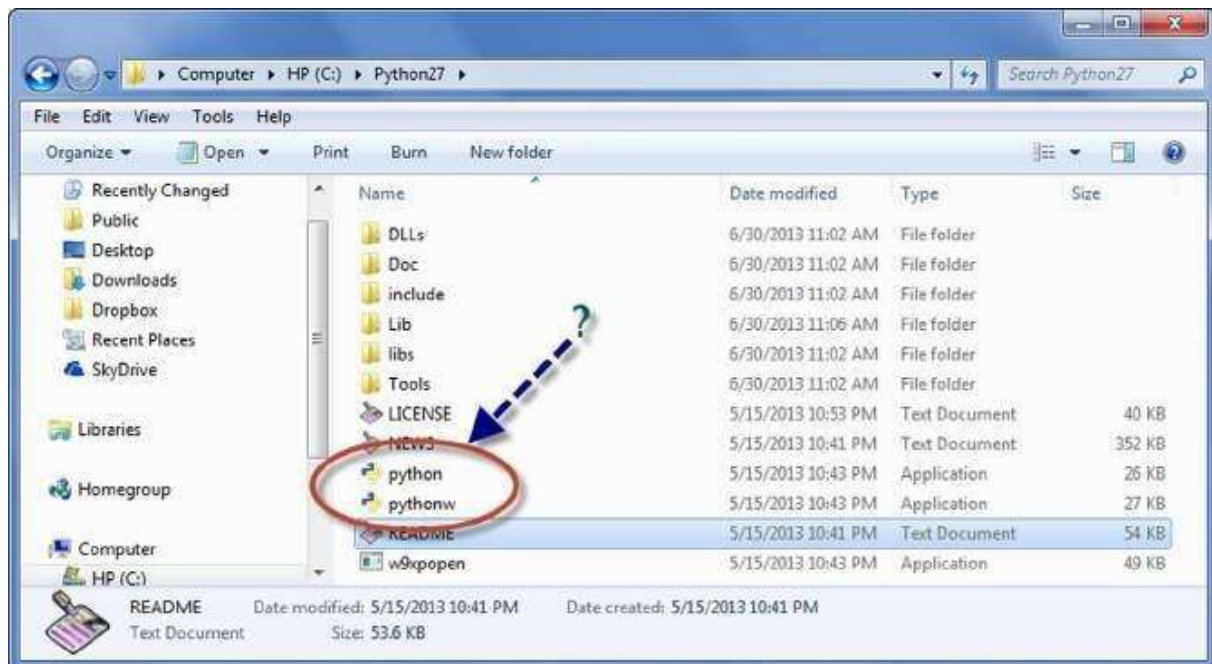


FIGURE 2.11
Python directory snapshot.

there are folders that contain Docs, Lib, DLL, and Tools along with key files License, News, and a Readme. Most importantly there are two application files Python and pythonw. These are the Python executable files.

To utilize the Python interactive interpreter you would double click on Python. If you have an existing Python program, and you do not want to display the interpreter window you would execute pythonw along with the associated Python file to execute (more on this later). For our immediate use, we will be using the Python application file. For easy access you can add this application to your Windows Taskbar and you will see the Python icon as highlighted in [Figure 2.12](#).

By launching the Python Taskbar icon (clicking it), the Python Interactive window is displayed as shown in [Figure 2.13](#). Your window will look a bit different from mine, as I have set mine up to be black on white to provide easier book publication. The title bar shows the directory the application was started from C:\Python27. The initial text lists the version of Python in this case Python 2.7.5, the date of the build and information regarding the processor, and Windows version in this case win32. The next line provides some helpful commands such as license, credits, help, and the copyright message. The next line starting with >>> is the Python prompt awaiting instructions. The tradition

here would be of course to test the installation by writing the standard programming language Hello World program, which only takes a single



FIGURE 2.12
Windows taskbar with Python icon.

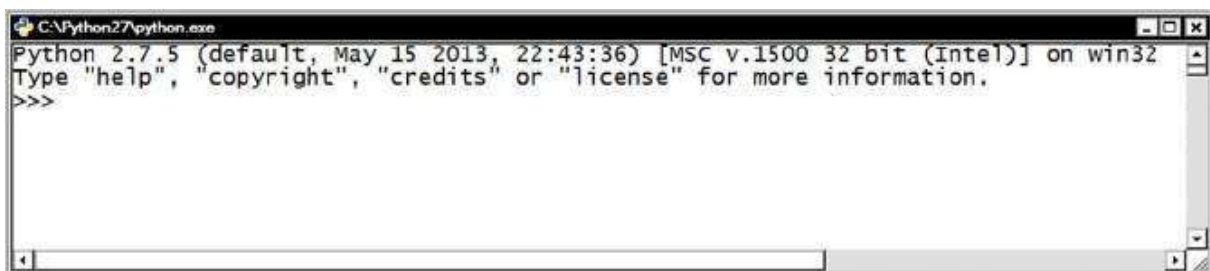


FIGURE 2.13
Python startup prompt and messages.

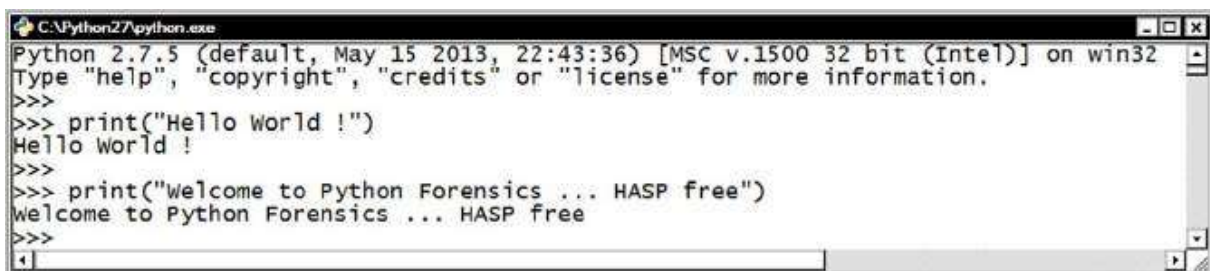


FIGURE 2.14
Python Hello World.

line in Python as shown in [Figure 2.14](#). I added a second print statement as well, promoting the idea of open and free forensic software that can be developed in Python, without the need for a HASP.

For those unfamiliar with a HASP (Hardware Against Software Piracy) (sometimes referred to as a Dongle) this is a device that provides copy protection for certain software programs. Without the HASP inserted, the software will not operate, thus only allowing licensed users to utilize the software on a single machine or network.

Now that we have at least verified that the Python interpreter is running and

accepting commands let us take a look at the commands, language structure, packages, and modules that are available.

PYTHON PACKAGES AND MODULES

Adding core functionality to an existing programming language is a standard part of software development. As new methods and innovations are made, developers supply these functional building blocks as modules or packages. Within the Python network, the majority of these modules and packages are free, with many including the full source code, allowing you to enhance the behavior of the supplied modules and to independently validate the code. Before we jump into adding third-party modules to Python, we need to understand what comes standard out of the box or more

Python Packages and Modules 25

specifically what is included in the Python Standard Library. I think you will be pleasantly surprised at the breadth of capability delivered as part of the Standard Library along with the built-in language itself.

The Python Standard Library

Python's Standard Library [Standard Library] is quite extensive and offers a broad range of built-in capabilities. These built-in functions are written primarily in the C programming language for both speed and abstraction. Since the Standard Library layer is compatible across systems, interfacing with platform-specific APIs (application programming interfaces) are abstracted or normalized for the Python programmer.

One of the fundamental operations that investigators perform is the generation of one-way cryptographic hash values.

A one-way cryptographic hash is used to create a signature of an existing string of bytes regardless of length (typically referred to as a message digest). One-way hashes have four basic characteristics: (1) A function that easily calculates and generates a message digest. (2) Possessing the message digest value alone provides no clues as to the original message or file. (3) It is infeasible (or computationally difficult) to change the contents of the message or file without changing the associated message digest. (4) It is infeasible to find two messages or files that differ in content, but produce the same message digest. It should be noted that attacks against known hash methods like MD5 and SHA-1 have been

successful under certain controlled situations.

The Python Standard Library contains a built-in module named hashlib that can perform one-way cryptographic hashing. The following simple hashing example can be executed on virtually any Python platform (Windows, Linux, Mac, iOS, Windows 8 Phone, Android, etc.) and will produce the same result.

```
#
# Python forensics
# Simple program to generate the SHA-256 # one-way cryptographic hash of a
given string

# Step 1
# Instruct the interpreter to import the # Standard library module hashlib

import hashlib
# print a message to the user

print
print("Simple program to generate the SHA-256 Hash of the String 'Python
forensics'")
print
# define a string with the desired text myString ¼ "Python forensics"

# create an object named hash which is of type sha256 hash ¼ hashlib.sha256()
# utilize the update method of the hash object to generate the # SHA 256 hash of
myString
hash.update(myString)

# obtain the generated hex values of the SHA256 Hash # from the object
# by utilizing the hexdigest method

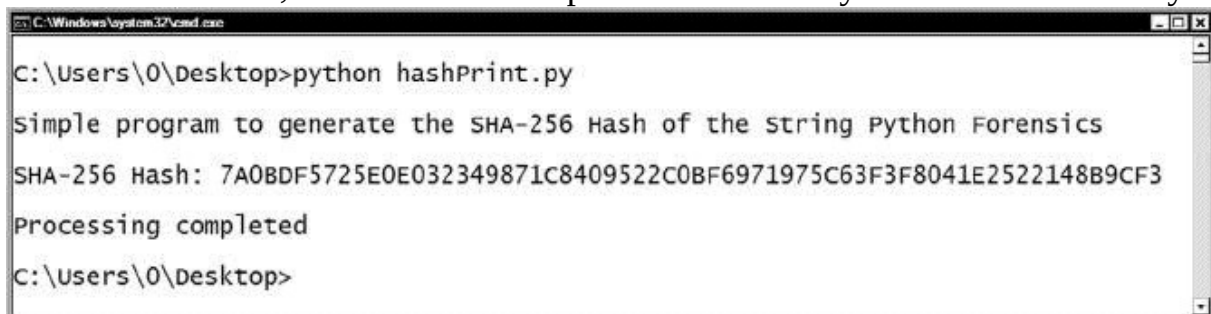
hexSHA256 ¼ hash.hexdigest()
# print out the result and utilize the upper method # to convert all the hex
characters to upper case
print("SHA-256 Hash: " + hexSHA256.upper()) print
print("Processing completed")
```

This simple example demonstrates how easy it is to access and utilize the

capabilities of the Python Standard Library, and how the same code runs on different platforms and generates the correct SHA256 hash value (see [Figures 2.15](#) and [2.16](#)).

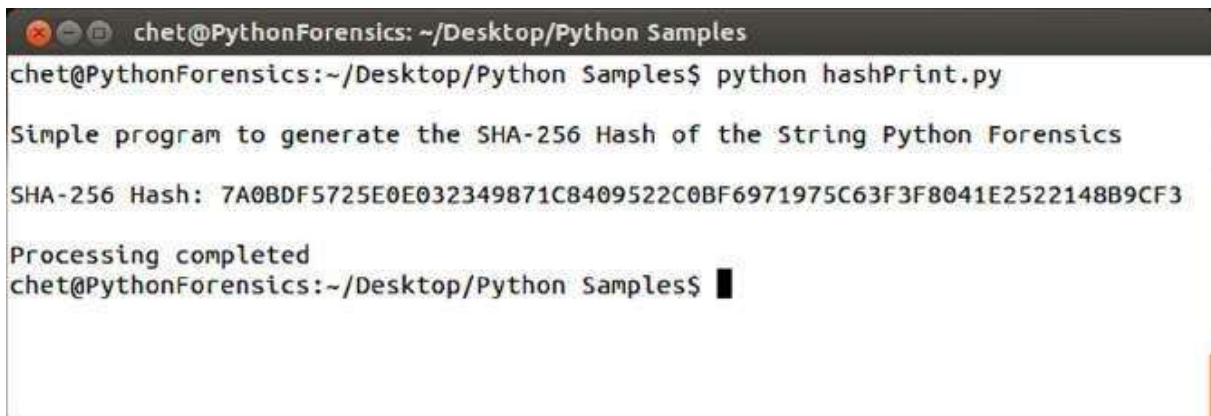
In [Chapter 3](#), we will spend considerable time and focus on the application of the one-way cryptographic hash algorithms for use in digital investigation and forensics.

In the next section, we will take a deeper dive into the Python Standard Library

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The command prompt shows the user at 'C:\Users\O\Desktop' running 'python hashPrint.py'. The output of the script is displayed: 'Simple program to generate the SHA-256 Hash of the String Python Forensics', 'SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C0BF6971975C63F3F8041E2522148B9CF3', and 'Processing completed'. The prompt returns to 'C:\Users\O\Desktop>'.

```
C:\Windows\system32\cmd.exe
C:\Users\O\Desktop>python hashPrint.py
Simple program to generate the SHA-256 Hash of the String Python Forensics
SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C0BF6971975C63F3F8041E2522148B9CF3
Processing completed
C:\Users\O\Desktop>
```

FIGURE 2.15
Windows execution of hashPrint.py.

A screenshot of a Linux terminal window. The title bar shows 'cheta@PythonForensics: ~/Desktop/Python Samples'. The prompt is 'cheta@PythonForensics:~/Desktop/Python Samples\$'. The command 'python hashPrint.py' is entered. The output is: 'Simple program to generate the SHA-256 Hash of the String Python Forensics', 'SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C0BF6971975C63F3F8041E2522148B9CF3', and 'Processing completed'. The prompt returns to 'cheta@PythonForensics:~/Desktop/Python Samples\$'.

```
cheta@PythonForensics: ~/Desktop/Python Samples
cheta@PythonForensics:~/Desktop/Python Samples$ python hashPrint.py
Simple program to generate the SHA-256 Hash of the String Python Forensics
SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C0BF6971975C63F3F8041E2522148B9CF3
Processing completed
cheta@PythonForensics:~/Desktop/Python Samples$
```

FIGURE 2.16
Ubuntu Linux execution of hashPrint.py.

WHAT IS INCLUDED IN THE STANDARD LIBRARY?

The Python Standard Library is broken down into broad categories: I will give you a brief summary description of the contents of each category with special attention on categories that I believe are unique or have forensic or digital investigation value. They are:

Built-in functions

Built-in functions, as the name implies, are always available to the Python programmer and provide fundamental capabilities that are additive to the language itself.

hex() and bin()

One of the capabilities that forensic investigators often encounter is the need to display variables in different bases; for example, from decimal (base 10), binary (base 2), and hexadecimal, or hex (base 16). These display capabilities are part of the built-in functions. The simple Python Shell session depicted in [Figure 2.17](#) demonstrates their behavior. In this example, we set the variable `a` equal to the decimal

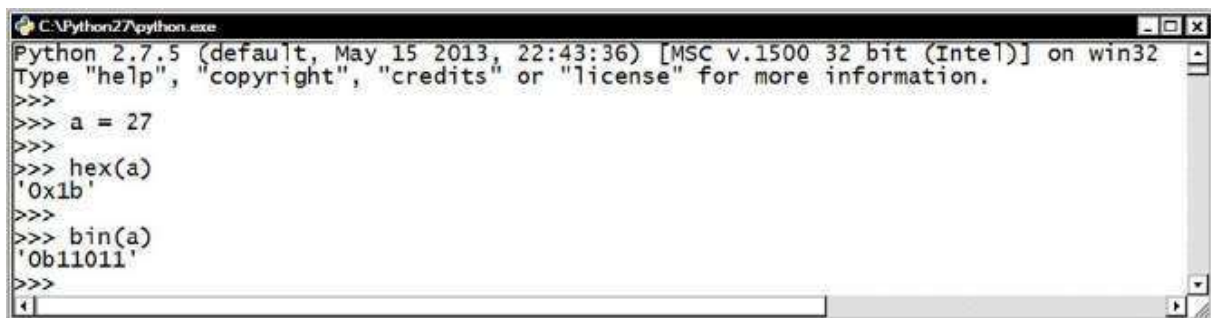
A screenshot of a Windows command prompt window titled "C:\Python27\python.exe". The window shows the Python 2.7.5 shell interface. The user has entered the following commands: `>>>`, `>>> a = 27`, `>>>`, `>>> hex(a)`, which returns `'0x1b'`, `>>>`, `>>> bin(a)`, which returns `'0b11011'`, and `>>>`. The window title bar includes standard Windows window controls (minimize, maximize, close) and the file path.

FIGURE 2.17
Python Shell session using `hex()` and `bin()`.

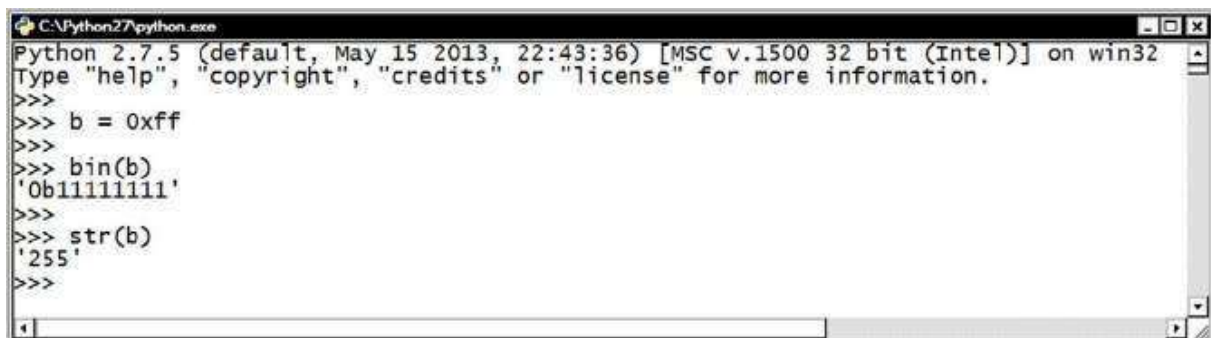
A screenshot of a Windows command prompt window titled "C:\Python27\python.exe". The window shows the Python 2.7.5 shell interface. The user has entered the following commands: `>>>`, `>>> b = 0xff`, `>>>`, `>>> bin(b)`, which returns `'0b11111111'`, `>>>`, `>>> str(b)`, which returns `'255'`, and `>>>`. The window title bar includes standard Windows window controls (minimize, maximize, close) and the file path.

FIGURE 2.18
Python Shell session entering hex values.

number 27 (the default is a decimal number). We then execute the function `hex(a)`, it displays the value of `a` in hexadecimal form; note the use of `0x` prefix to

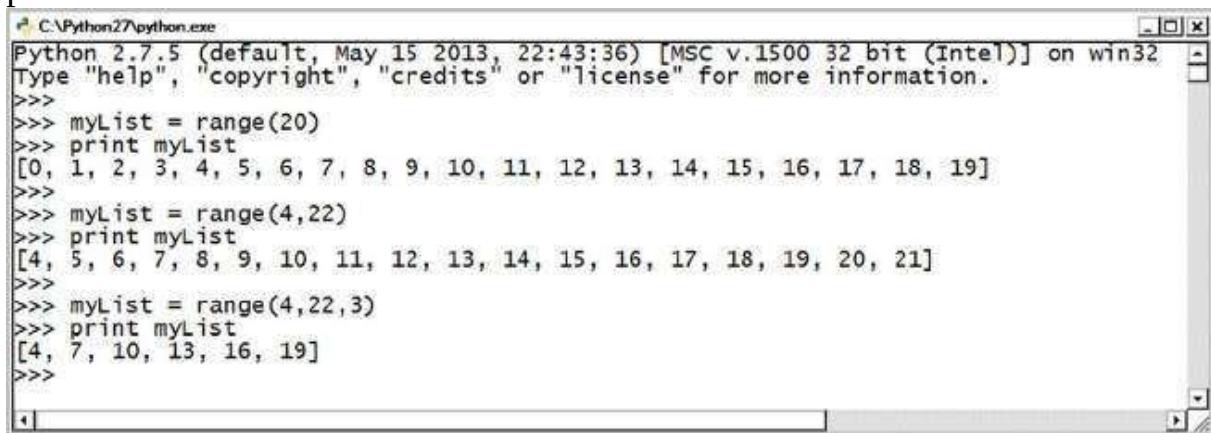
denote hex. Utilizing the binary conversion function `bin(a)` renders the decimal value 27 as a binary string of 1 s and 0 s, or 11011; note the use of the 0b prefix to denote binary.

As shown in [Figure 2.18](#), you can perform the reverse by specifying a variable as a hex number and then displaying the variable in binary or decimal form. Note that the values stored in the variable `a` or `b` shown in these examples are simply integer values, the function `hex()`, `bin()`, and `str()` simply render the variables in hexadecimal, binary, or decimal notation. In other words the variable does not change, just the way we view it does.

`range()`

Another useful built-in function is `range()`. We often need to create a list of items, and this built-in function can assist us in automatically creating such a list. Lists, Tuples, and Dictionaries are quite powerful constructs in Python, and we will be using each of these to solve challenge problems throughout the book. For now I am just going to show you the basics.

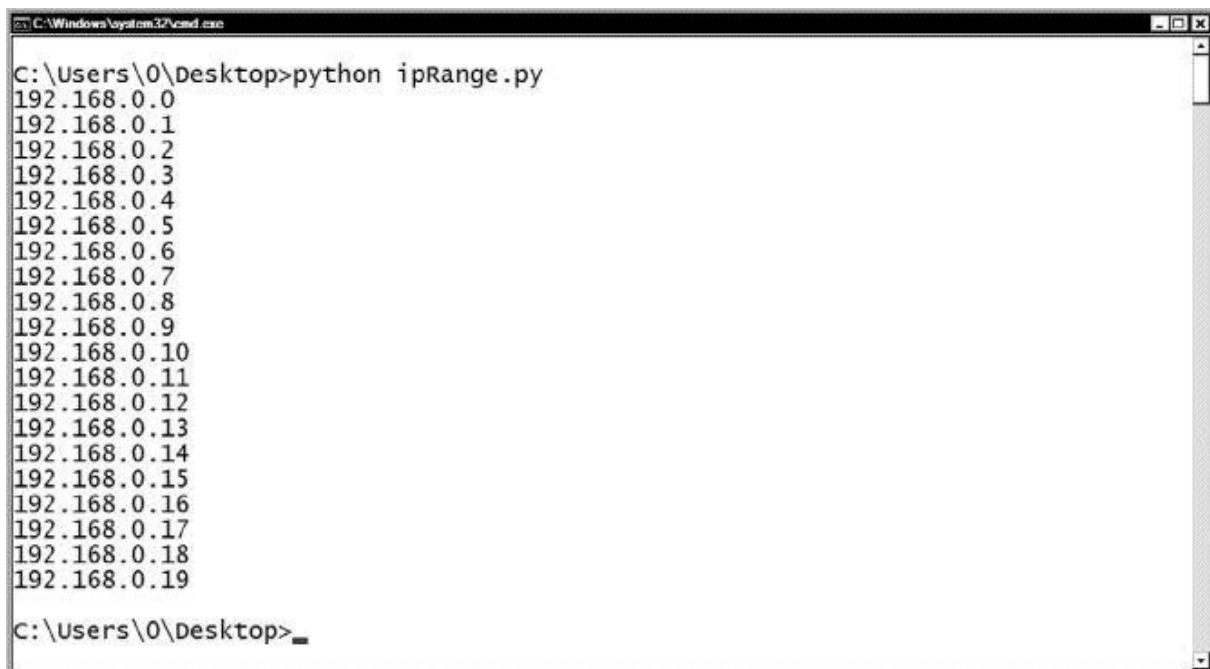
I have executed a few examples using the Python Shell as shown in [Figure 2.19](#). The first one creates a list of the first 20 integers starting at zero. The second provides



```
C:\Python27\python.exe
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> myList = range(20)
>>> print myList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>>
>>> myList = range(4,22)
>>> print myList
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
>>>
>>> myList = range(4,22,3)
>>> print myList
[4, 7, 10, 13, 16, 19]
>>>
```

FIGURE 2.19

Python Shell creating lists using the `range()` built-in Standard Library function.



```
C:\Windows\system32\cmd.exe
C:\Users\O\Desktop>python ipRange.py
192.168.0.0
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
192.168.0.5
192.168.0.6
192.168.0.7
192.168.0.8
192.168.0.9
192.168.0.10
192.168.0.11
192.168.0.12
192.168.0.13
192.168.0.14
192.168.0.15
192.168.0.16
192.168.0.17
192.168.0.18
192.168.0.19
C:\Users\O\Desktop>
```

FIGURE 2.20
ipRange execution.

a list of integers starting at 4 and ending at 22 (notice this list stops at 22 and does not include it in the list). Finally, I create a list from 4 to 22 in steps of 3.

You will see later how using the `range()` function and lists can be helpful in automatically generating lists of useful values. For example, the program below generates a list that contains the first 20 host IP addresses for the class C address starting with 192.168.0. You can see the program output in [Figure 2.20](#).

```
# define a variable to hold a string representing the base address baseAddress ¼  
"192.168.0."  
# next define a list of host addresses using the range  
# standard library function (this will give us values of 1-19 hostAddresses ¼  
range(20)  
# define a list that will hold the result ip strings  
# this starts out as a simple empty list  
ipRange ¼ []  
# loop through the host addresses since the list hostAddresses # contains the  
integers from 0-19 and we can create  
# a loop in Python that processes each of the list elements # stored in
```

```

hostAddresses, where i is the loop counter value for i in hostAddresses:
# append the combined ip strings to the ipRange list
# because ipRange is a list object, the object has a set of # attributes and
methods. We are going to invoke the append method # each time through the
loop and concatenate the base address # string with the string value of the integer
ipRange.append(baseAddress+str(i))

# |||||__ value of the host address # |||__ function to convert int to str # |||__
The string "192.168.0"
# |__ The append method of the list ipRange # |__ The list object ipRange
#
# Once completed we want to print out the ip range list # here we use the print
function and instruct it to print out # the ipRange list object, I wanted to print out
each of the # resulting ip address on a separate line, so I looped
# through the ipRange list object one entry at a time.
for ipAddr in ipRange:

print ipAddr

```

Other built-in functions

The following is the complete list of built-in functions taken from the Python Standard Library documentation. You can get more information on each of the functions at the URL <http://docs.python.org/2/library/functions.html>

Built-in functions for Python version 2.x are given in [Table 2.1](#). I will be using a good number of these functions in [Chapters 3–11](#), the Cookbook sections.

Table 2.1 Python 2.7 built-in functions

```

abs()
all()
any()
basestring() bin()
bool()
bytearray()
callable()
chr()
classmethod() cmp()
compile()
complex()

```

delattr()
dict()
dir()
divmod()
enumerate() eval()
execfile()
file()
filter()
float()
format()
frozenset() getattr()
globals()
hasattr()
hash()
help()
hex()
id()
input()
int()
isinstance() issubclass() iter()
len()
list()
locals()
long()
map()
max()
memoryview() min()
next()
object()
oct()
open()
ord()
pow()
print()
property() range()
raw_input() reduce()
reload()
repr()
reversed() round()

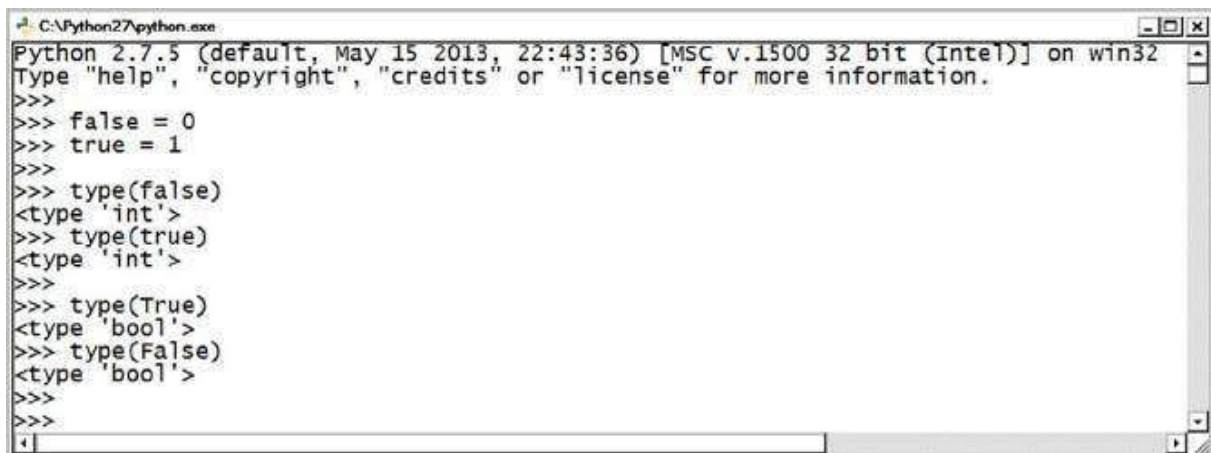
```
set()
setattr()
slice()
sorted()
staticmethod() str()
sum()
super()
tuple()
type()
unichr()
unicode()
vars()
xrange()
zip()
__import__() apply()
buffer()
coerce()
intern()
```

Built-in constants

There are several built-in constants within Python. Constants are different from variables in that constants, as the name implies, never change—while variables can take on ever changing values.

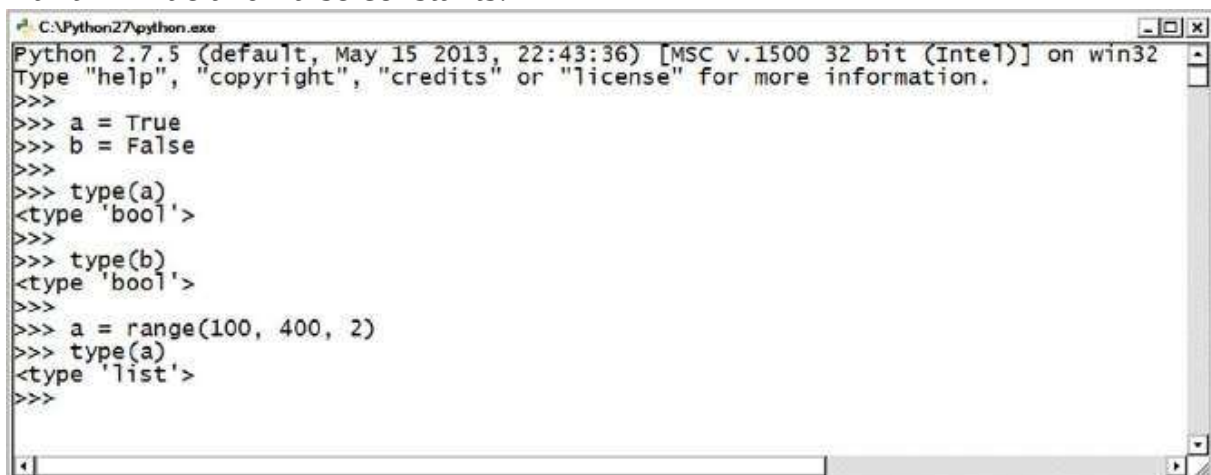
The two most important built-in constants within Python are True and False, which are Boolean values. Take a look at [Figure 2.21](#), I have used the Python Shell to define two variables false¼ 0 and true¼ 1 (notice these are lower case). When I use the built-in Python function type() to identify the variable type, you notice that it returns the value of int (integer) for both true and false. However, when I use type() on True and False (the built-in constants) it returns type bool (Boolean).

By the same token you can create variables a¼ True and b¼ False (as shown in [Figure 2.22](#)) and check the type() and you see that they are of type bool. Do not get too dependent on this, however, as Python is NOT a strongly typed language. In a strongly typed language, a variable has a declared type that cannot change during the



```
C:\Python27\python.exe
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> false = 0
>>> true = 1
>>>
>>> type(false)
<type 'int'>
>>> type(true)
<type 'int'>
>>>
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
>>>
>>>
```

FIGURE 2.21
Built-in True and False constants.



```
C:\Python27\python.exe
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> a = True
>>> b = False
>>>
>>> type(a)
<type 'bool'>
>>>
>>> type(b)
<type 'bool'>
>>>
>>> a = range(100, 400, 2)
>>> type(a)
<type 'list'>
>>>
```

FIGURE 2.22
Python is not a strongly typed language.

life of the program. Python does not have this restriction; I could just as easily in the very next statement declare `a = range(100, 400, 2)` and change the type of variable `a` from a `bool` into a `list`. To a hard core developer in a strongly typed language such as Ada, C#, Java, or Pascal, this lack of structure would have them run for the hills, however, with the proper discipline in naming objects we can control most of the chaos.

When we develop our first real application in [Chapter 3](#), I will cover some tips on how to develop forensically sound programs using a weakly typed language.

A strongly typed language must define the variable or object along with the specific type. This type must not change once declared and the compiler itself

must enforce the typing rules, not the programmer. For example, if a variable is declared as an integer, it would be illegal to assign the value 3.14 to the variable.

Built-in types

Python has many built-in types that we can take advantage of in our forensic applications. For a detailed description of each of the standard built-in types, the most up-to-date information is available at:

<http://docs.python.org/2/library/stdtypes.html>

The basic categories include:

Numeric Types : int, float, long and complex

Sequence Types: list, tuple, str, unicode, bytearray and buffer Set Types: set and frozenset

Mapping Types: dict or dictionary

File Objects: file

Memory: MemoryView Type

We have already seen a few of these in action during the simple examples. We will be extensively leveraging some of the more advanced types throughout our example programs including: bytearray, lists, dictionaries, unicode, sets, frozensets, and the MemoryView Type. All of these have very unique forensic applications.

Some of the bitwise operations that are included in Python are vital to many forensic and examination operations. They are quite standard, but it is worth covering to give you a flavor of what is available. Bitwise operations are exclusive to integer data types of virtually any size. For the examples below the values of x and y are the integers.

$x|y$: bitwise or of the variables x and y

$x \wedge y$: bitwise exclusive or of variables x and y

$x \& y$: bitwise and of variables x and y

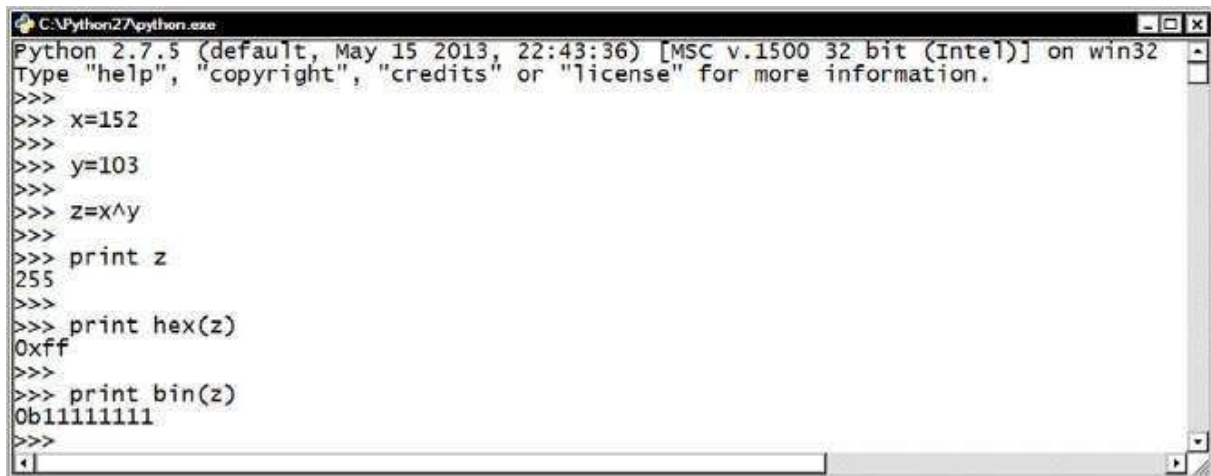
$x \ll n$: variable x is shifted n bits left

$x \gg n$: variable x is shifted n bits right

$\sim x$: the variable x bits are inverted

In [Figure 2.23](#), I use the bitwise exclusive or operator \wedge which is used in many one-way hash and cryptography operations. We start out by setting up our

variables

A screenshot of a Windows command prompt window titled "C:\Python27\python.exe". The window shows the following text:

```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> x=152
>>> y=103
>>> z=x^y
>>>
>>> print z
255
>>>
>>> print hex(z)
0xff
>>>
>>> print bin(z)
0b11111111
>>>
```

FIGURE 2.23

Apply an Exclusive OR (XOR)

x and y. We set x equal to 152 and y equal to 103—both decimal numbers. We then set z equal to the exclusive or of x and y. We can then print out the decimal, hex, and binary representations of the result.

As you can see exclusive or (XOR) differs from a simple OR operation. XOR requires that each bit of the two numbers be evaluated and a one is returned if and only if, only one of the bits is set to a one. For example:

Binary 101

Binary 001

Produces:100

The least significant bit of each binary value is 1, therefore it does not meet the exclusivity test and the resulting value is 0.

Other more complex built-in types such as memory view types, byte arrays, list, unicode, and dict provide building blocks to tackle difficult data carving, search, indexing, and analytical analysis of the most complex digital evidence types available. I will be leveraging those extensively as we move into the example chapters.

Built-in exceptions

Most modern software languages support inline exception handling and Python is no different. Exception handling in most applications is important, but within

forensic and digital investigation applications it is vital. When developing these applications it is important that you can demonstrate that you have handled all the possible situations. When developing an application we all tend to test our code using “happy mode testing.” Ronda Caracappa, one of the best software quality people I have ever known, coined the phrase “happy mode testing” many years ago. This means that we test the normal flow of our programs with all the inputs, events, and third-party modules behaving exactly as we expect them to. The problem is they rarely, if ever do just that.

To facilitate exception handling, Python uses the try/except model. Here is an example: We setup our program to divide two numbers—in this case integer 27 divided by 0. If this were to happen while a program was running and we did not setup to catch such errors, the program would fault and crash. However, by using exception handling we can avoid trouble and handle the fault. In [Chapter 3](#), our first real program, we will encounter several real-life conditions where exception handling is vital. And, with the help of Python’s built-in exception handling, we can deal with the potential pitfalls with ease and use these exceptions to log any processing difficulties.

```
x = 27
y = 0
try:

    z = x/y
except:
    print("Divide by Zero")
```

Note this is a simple example of using the try/except method to catch execution-based exceptions, in a neat and tidy manner. Other examples dealing with operating system, file handling, and network exceptions will be demonstrated during the cookbook chapters.

File and directory access

The Python Standard Library provides a rich set of file and directory access services. These services not only provide the expected ability to open, read, and write files but they also include several very unique built-in capabilities; for example, common path name manipulations. Operating systems and platforms handle directory paths differently (i.e., Windows uses the c:\users\... notation while Unix environments utilize /etc/... notation), and some operating

environments support simple ASCII file and directory naming conventions, while others support full Unicode naming conventions. All file and directory access functions need to support these differences uniformly otherwise cross-platform operations would not be possible. In addition, built into the Python Standard Library file are directory compare functions, automatic creation of temporary directories and high-level file input handling which makes dealing with file systems easier for both novice and pros.

Data compression and archiving

Instead of having to use third-party libraries to perform standard compression and archiving functions like zip and tar, these functions are built-in. This includes not only compressing and decompressing archives but also includes extracting content information for zip files. During investigation, we often encounter encrypted zip files, and Python handles these as well (provided you either have the password or you can of course apply dictionary or brute force attack methods).

File formats

Also built into the Standard Library are modules to handle special file formats; for example, comma separated value (CSV) files, the Hypertext Markup Language (HTML), the eXtensible Markup Language (XML), and even the JavaScript Object Notation (JSON) format. We will be using these modules in later chapters to carve data from Web pages and other Internet content, and to create standardized XML output files for generating reports.

Cryptographic services

We will make extensive use of the hashlib cryptographic module to solve some basic problems that face digital investigators in [Chapter 3](#). The hashlib module not only directly supports legacy one-way cryptographic hashing such as MD5 but also directly supports modern one-way hashing algorithms such as SHA-1, SHA256, and SHA512. The ease of integration and use of these libraries along with the optimized performance provides direct access to functions that are vital to digital investigation.

Operating system services

The operating systems (OS) services provide access to core OS functions that work across platforms. In [Figure 2.24](#) I utilize the os module to list the contents of current working directory. The steps are pretty simple:

1. Import the os module
2. We use the os modules `os.getcwd()` method to retrieve the current working directory path, in other words the path we are currently on
3. We store that value in the variable `myCWD`

A screenshot of a Windows command prompt window titled "C:\Python27\python.exe". The window shows a Python 2.7 interpreter session. The user has entered the following commands: `>>> import os`, `>>> myCWD = os.getcwd()`, `>>> dirContents = os.listdir(myCWD)`, and `>>> for names in dirContents:` followed by an indented `print names`. The output of the `print` statements is a list of files and directories: `DLLs`, `Doc`, `include`, `Lib`, `libs`, `LICENSE.txt`, `NEWS.txt`, `python.exe`, `pythonw.exe`, `README.txt`, `Tools`, and `w9xpopen.exe`. The prompt `>>>` is shown at the end of the last line of code.

```
>>> import os
>>> myCWD = os.getcwd()
>>> dirContents = os.listdir(myCWD)
>>> for names in dirContents:
...     print names
...
DLLs
Doc
include
Lib
libs
LICENSE.txt
NEWS.txt
python.exe
pythonw.exe
README.txt
Tools
w9xpopen.exe
>>>
```

FIGURE 2.24

Example using the os module from the Standard Library.

4. We then use the `os.listdir()` method to obtain the names of the files and directories in the current working directory
5. Instead of just printing out `os.listdir()`, i.e., `print(os.listdir())` I decided to store the results into a new list named `dirContents`
6. I then use `dirContents` to print out each name on a separate line by processing through the list with a simple for loop

Having the files in a list enables us to process each of the files or directories based upon our needs.

Other operating system services such as stream-based io, time, logging, parsing, and platform modules all provide easy access to operating system services. Once again, on a cross-platform basis this is to easily handle Windows, Linux, Mac, and numerous legacy systems.

Standard Library summary

This has been just a quick introduction and tour of only a few of the Standard

Library capabilities built directly into Python. There are many good references and complete books dedicated to this subject for those that want to delve further. However, there are also online resources available directly at [Python.org](https://python.org), where you can find a plethora of tutorials, code examples, and support for every Standard Library data type, module, attribute, and method. My goal here was just to give you a taste of what you can do and how easy it is to use the Standard Library.

In the next section, I will introduce you to some of the third-party libraries that we will be using and what they bring to the party.

THIRD-PARTY PACKAGES AND MODULES

As I mentioned earlier, many third-party packages and modules are available and I will introduce each during the cookbook [Chapters 3–11](#). However, I wanted to mention a couple here to give you an idea why third-party modules are available and the kind of capabilities that they bring to digital investigation.

The natural language toolkit [NLTK]

Today when performing investigations, e-discovery, or incident response, it is important to determine what was communicated via e-mail, text message, written documents, and other correspondence. We utilize crude tools and technologies today like simple keyword and phrase searching and grep to discover and parse these communications.

Grep, which is short for global regular expression print, allows users to search text for the occurrence of a defined regular expression. Regular expressions are composed of meta characters and usually a sequence of text characters that represent a given pattern, which allow the computer program to perform advanced or simple pattern matching.

The problem is these technologies do not consider language or semantics, thus it is easy to miss certain communications or maybe misunderstand them. The NLTK modules provide an infrastructure to create real Natural Language programs and build small or even large corpus.

A corpus is a collection of material which is most often written, but can be spoken material as well. The material is digitally organized in such a way that it lends itself to the study of linguistic structures which helps to understand meaning.

This gives us the ability to search, understand, and derive meaning. We will be using NLTK in one of our Cookbook examples to help produce specific applications to assist digital investigators. For example, we can use linguistics to help determine if specific documents were likely to have been written by a specific person.

Twisted matrix [TWISTED]

In a world where everything is connected, our ability to collect and examine evidence requires both a postmortem and a live investigation mindset. This implies that we must have a reliable network library that provides an asynchronous and event-based environment that is open source and written in Python. This includes TCP, UDP, Web Services, Mail Services, Authentication, Secure Shell support, and quite a bit more.

INTEGRATED DEVELOPMENT ENVIRONMENTS

As I mentioned at the outset of this chapter, one of the keys to success would be the establishment of an IDE that would provide value, instill confidence, and create efficiencies during the development process. In our case, we also want to ensure that the quality of what we create will exceed the standards of evidence. You have several options for an IDE—some of which are completely free—and some which have a nominal cost of entry. I will reveal my choice at the end of the section, but you can choose your own path, as in many ways this is like choosing between Nike[®] or Adidas[®] and is mostly defined by what you like and are comfortable with.

What are the options?

Actually, it may require a whole book to answer that question! Here is just a short list of the available options: IDLE, PyCharm, PyDev, WingIDE, and mDev. I will narrow my focus to just two—IDLE and WingIDE.

IDLE

IDLE specifically stands for Integrated Development Environment and the name was chosen by Guido Van Rossum the creator of Python. IDLE is written in Python and has a nice set of features that can handle basic integrated development. It includes a Python Shell to experiment with commands; it highlights source code and provides cross-platform support. IDLE does support a debugger allowing users to set breakpoints and step through code ([Figure 2.25](#)). However, it was not developed as a professional development

environment, but it does support users that are developing simple applications, and students that are learning the language. Best thing is that it is completely free. Please note, I am sure that there are many Python developers

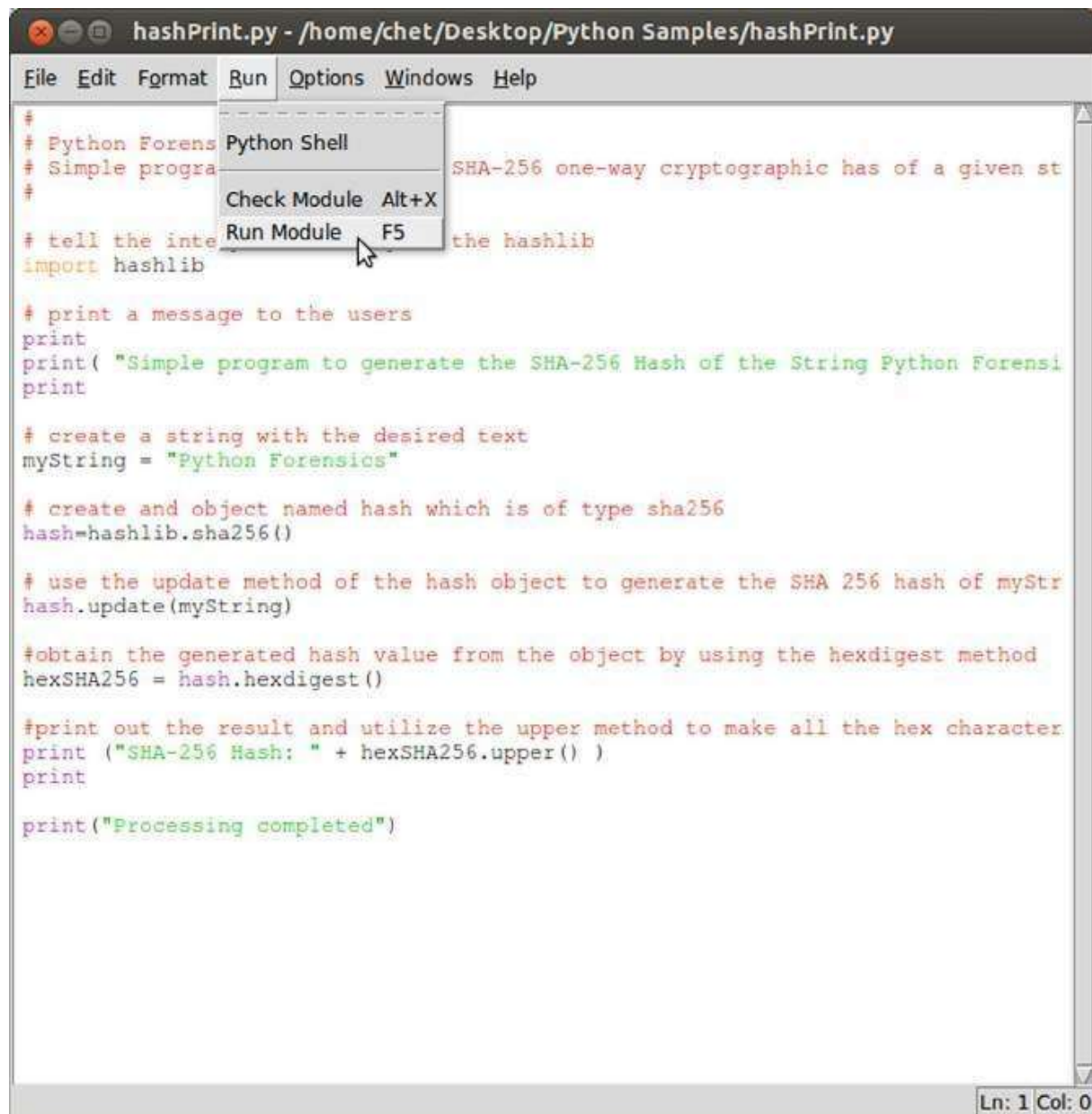


FIGURE 2.25

Python IDLE integrated development environment.

out there that use Python IDLE and will disagree with my statement about not considering it as a professional development environment, as I am sure many of you have developed quite sophisticated applications with IDLE and I accept that.

But I think for developing digital investigation applications today, there are several other alternatives that have more advanced capabilities.

WingIDE

WingIDE on the other hand is not free (you can obtain a free version if you are a student or unpaid open source developer), but does have a rich feature set that will easily support the development of sophisticated Python Applications.

WingIDE comes in three flavors based on your intended use:

1. Free for students and unpaid open source developers
2. WingIDE Personal—As the name implies, limits some features
3. WingIDE Professional—For those wanting all the features and support, for example, this version contains unit testing, direct interface with revision control systems, advanced debugging and breakpoint settings, and pyLint integration.

pyLint is a Python source code analyzer that helps to identify poor coding practices, points out potential bugs, and warns you about the likelihood of failure. It will also assign an overall grade for your code to help you to keep improving.

You can see the layout I have chosen for WingIDE in [Figure 2.26](#). The IDE is configurable and resizable to allow you to customize the GUI. I have placed labels in each of the major sections in order to cover them here.

Section A : In this section, we find the current local and global variables associated with the current running program, since I have the program momentarily stopped at a breakpoint. As you can see, the text box at the top indicates that module ipRange.py is currently processing line 10. In the window it displays the variable baseAddress currently containing the string “192.168.0.”

Section B : Displays the program output so far. The initial print statement has successfully printed the string “Generating ip Range.” At the bottom of that window you can see several other options; the currently selected option is “Debug I/O.” Another quite useful option is the “Python Shell.” If this option is selected you can execute, learn, or experiment with any of the Python functions or even write a sequence of statements to try before you code (see [Figure 2.27](#)).

Section C : This window contains the project information. Due to the fact this is a very simple project only containing a single file, ipRange.py, that is all that is displayed in that window. As we build more complex applications this window

will help you keep track of all the components of your program.

Section D : This window contains the source code of the current program, along with debug settings. Notice that line 10 contains a dot where the breakpoint was set and it is highlighted because the program reached that point and the breakpoint stopped execution.

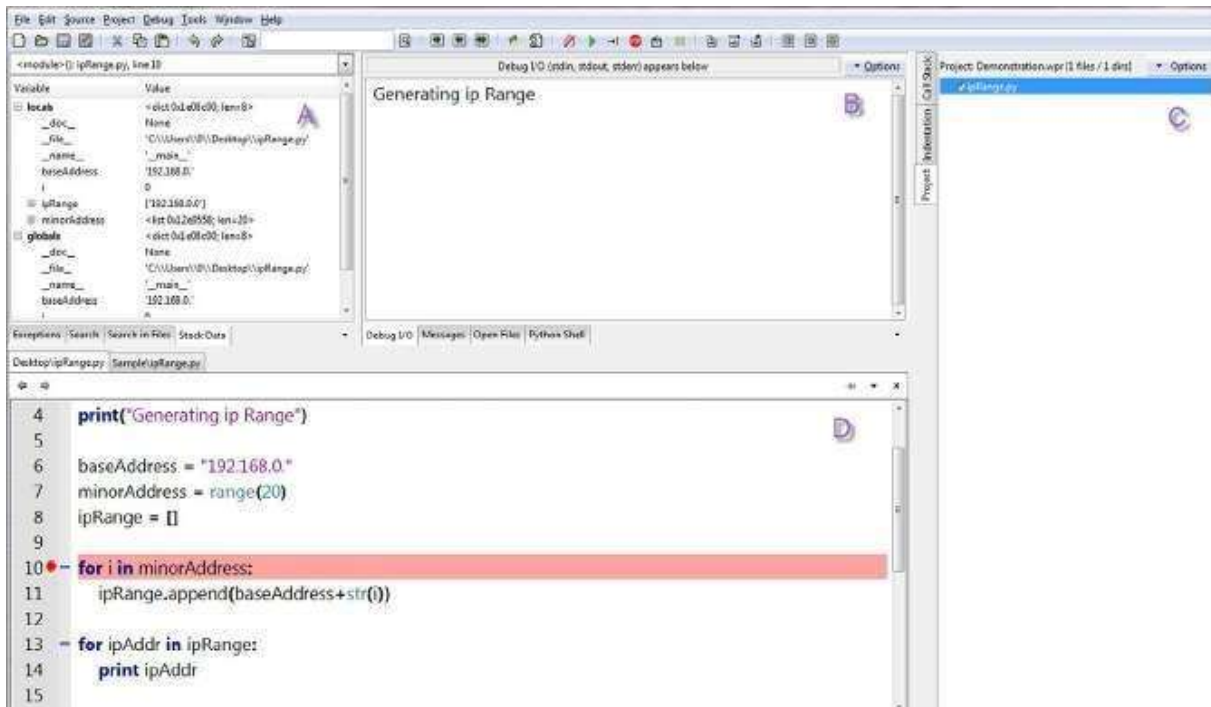


FIGURE 2.26
Snapshot of WingIDE 4.1 Personal.



WingIDE Python Shell display.

FIGURE 2.27

Now that you have the general idea, let us actually step through this simple program with WingIDE and watch the changes happen. In [Figure 2.28](#) we step over (I used the [F6] key to do this, you could also select the option under the debug menu) the code below and stop before executing the ipRange initialization.

```
baseAddress ¼ "192.168.0."  
minorAddress ¼ range(20)  
ipRange ¼ []
```

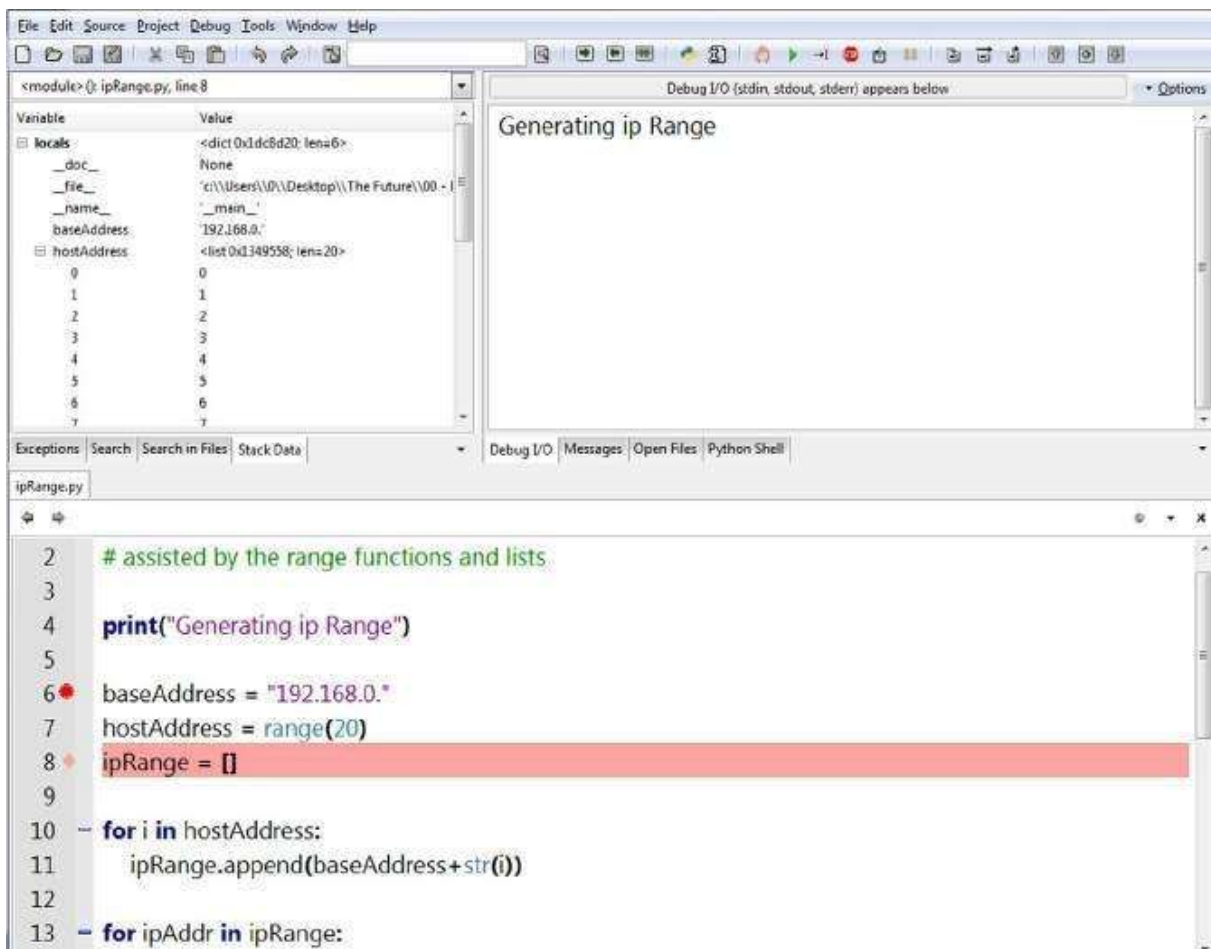


FIGURE 2.28
WingIDE in action.

At this point we can examine the variable section top left (in [Figure 2.28](#)) and we see that the variable `baseAddress` is equal to the string "192.168.0" and the variable `hostAddress` is a list with length 20 and has values 0-19 as expected. We name this variable `hostAddress` because in typical Class C address, the last

dotted decimal digit typically identifies the host. We consider naming conventions for variables, functions, methods, and attributes as we get into the Cookbook chapters, but I am sure you can already see the pattern.

Next, I am going to step over the initialization of the `ipRange` (again using [F6]) and also step into the “for loop” stopping before we execute the first `ipRange.append` method. As you can see in [Figure 2.29](#), the new variables `i` which equals 0, and is the first `hostAddress`, and `ipRange`, which is currently an empty list, have now appeared.

Now I am going to step through each of the for loop items, 20 iterations in this case and stop on the first print statement of the second loop. You notice in [Figure 2.30](#) in the variable section, I have expanded the contents of the `ipRange` list and you see the completed `ipRange` strings that now contain the added host addresses. All that is left is to print out the `ipRange` array line by line.

This should give you a good overview of the capabilities of WingIDE and how to use it to set breakpoints, step through the code, and examine variables. I will be using this IDE throughout the rest of the book introducing advanced features as we go.

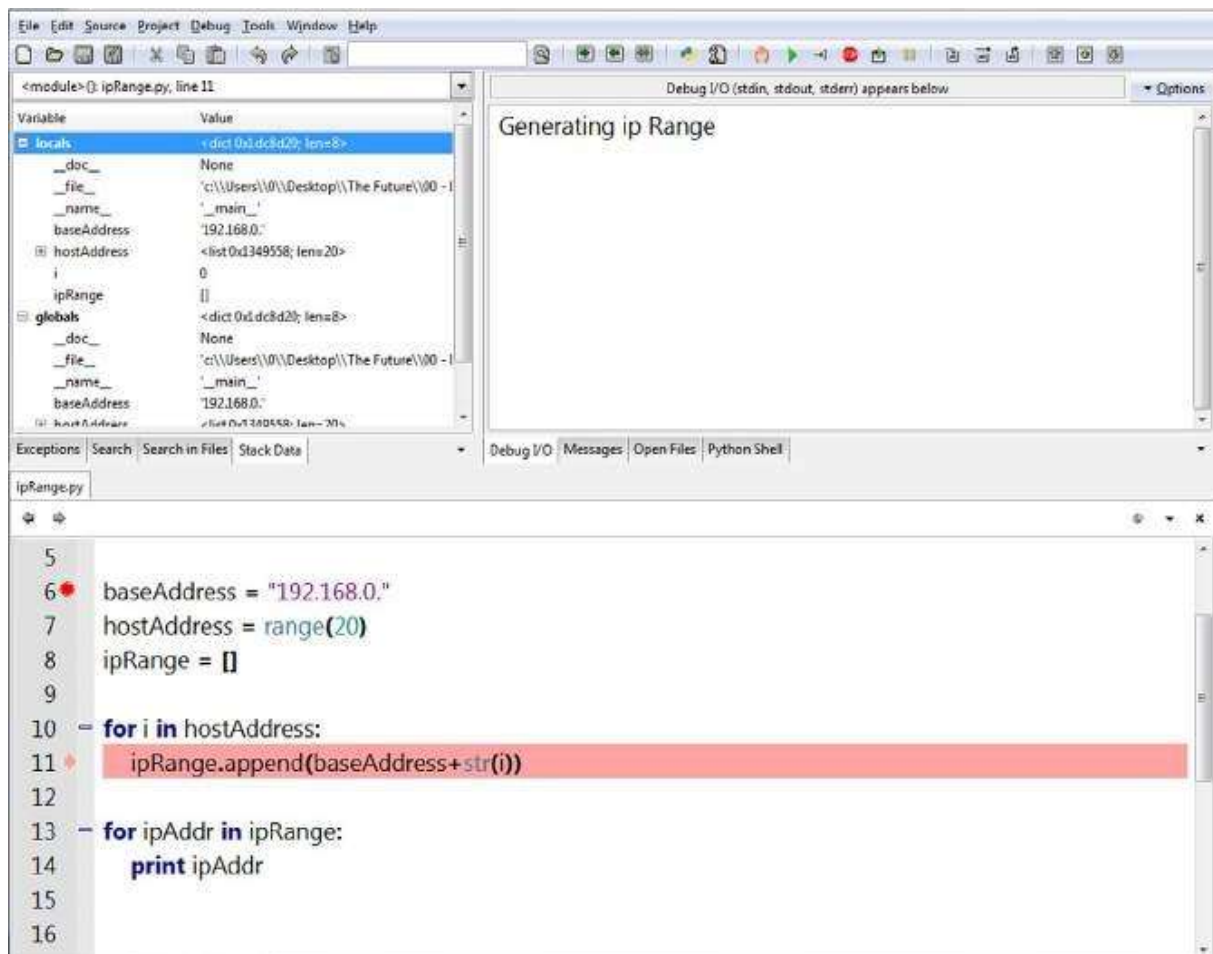


FIGURE 2.29
Using WingIDE to step through code.

One final feature that I use a great deal when writing code is auto-completion. In [Figure 2.31](#), I want to see what other methods and attributes are associated with the list `ipRange`. We have seen what the `append` method does, but you might be wondering what else we can do. By simply adding a line of code, `ipRange`, and entering a dot (a period), a list box automatically appears and displays all the attributes and methods that are available for `ipRange`. Note that since `ipRange` is a list, these are actually the attributes and methods that can be generally applied to any list. The dropdown you see is only a partial list of methods and attributes. As you can see, I have scrolled down to the `sort` method which would allow me to sort the list. The parameters that I pass to the `sort` method will determine how the list will actually be sorted.

Up to this point we have focused on developing and debugging on a Windows platform. In the next section, we will walk through an Ubuntu Linux installation

and quick session.

Python running on Ubuntu Linux

There are many investigation and forensic advantages to Linux-based tools. You can of course achieve greater performance, safely mount a wide range of media and image types, avoid the cost of the operating system, and have potentially greater flexibility. So let us setup Python running on Ubuntu. It really is easier than you might think.

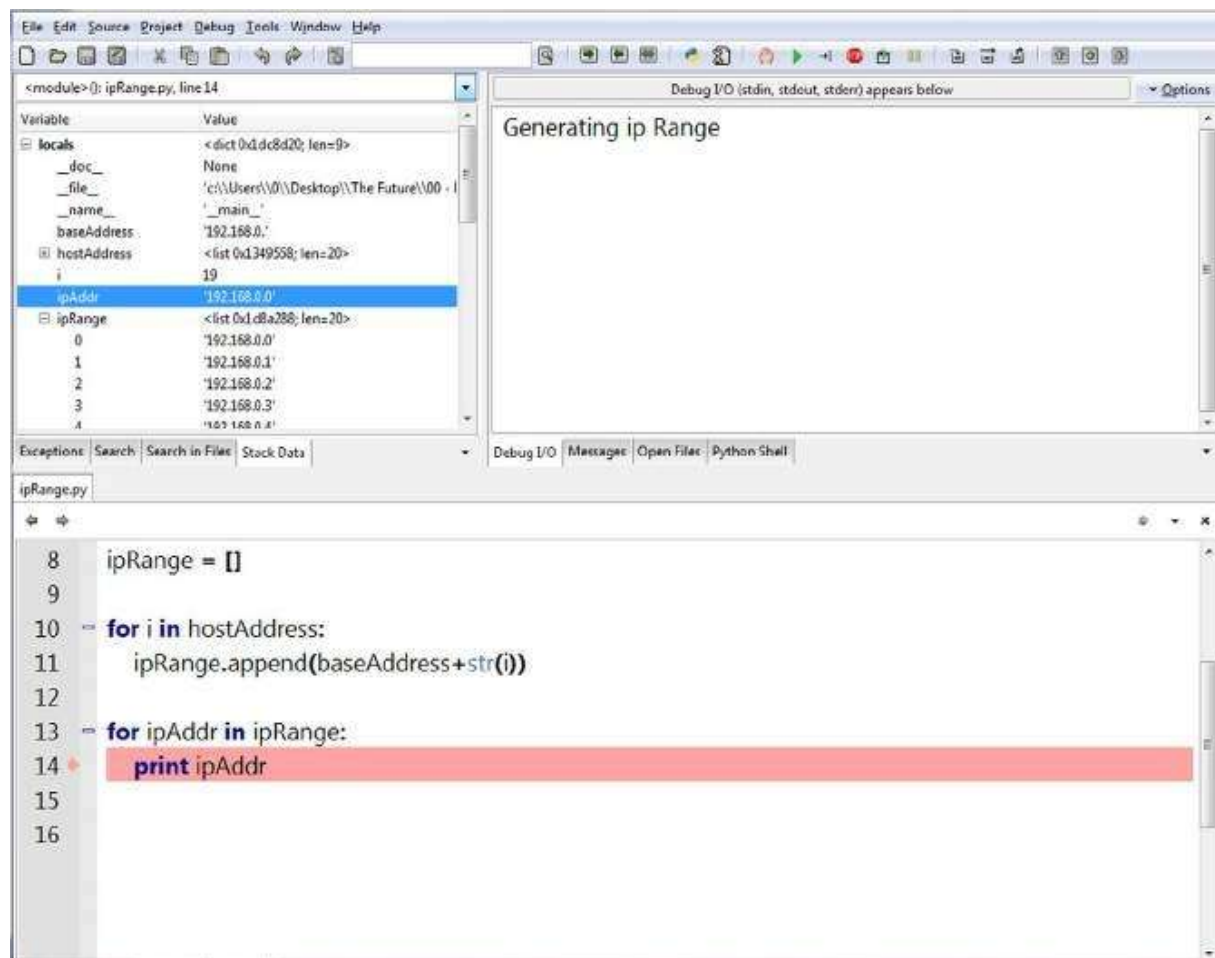


FIGURE 2.30

WingIDE examination of the completed list.

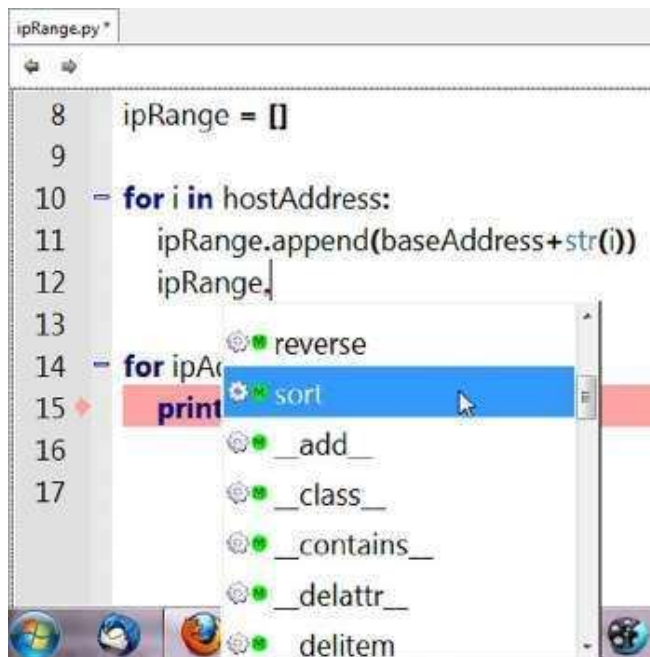


FIGURE 2.31
WingIDE auto complete feature.



FIGURE 2.32

Ubuntu download Web page 12.04 LTS.

As of this writing, I would make the following strong recommendation on Ubuntu installation and setup. In [Figure 2.32](#) you see a screenshot from the Ubuntu download page (<http://www.ubuntu.com/download/desktop>). Version 12.x LTS of Ubuntu is your best bet. This version is available in either 32 or 64 bit and is verified to operate on a wide range of standard desktop PCs. I am choosing version 12.x LTS because this is the long-term support version of the OS. Ubuntu is promising to provide support and security updates for this version through April 2017, giving you a stable well-tested platform for digital investigation and forensics.

As far as installing and running Python on the Ubuntu operating system, version 2.7.3 is already installed as part of the OS, thus installation of the base product is done for you. If you are not sure about your specific installation, simply launch a terminal window (as shown in [Figure 2.33](#)), and type Python at the prompt. If it is installed properly you will see a similar message as the one shown.

If this does not produce the desired results, you should reinstall Ubuntu 12.x LTS and this should repair the problem. (Make sure you backup your data before attempting any reinstall.)

If you have not used Linux in a while, or this is your first time installing Ubuntu, you will find that the days of struggling to install a Linux OS are over. As the saying goes “it is as easy as pie.” Once you have successfully installed the OS, the process of adding new capabilities is straightforward. For example, if you intend to install an IDE like WingIDE or IDLE in the Ubuntu environment, you can obtain the installers from the Ubuntu Software Center. The Ubuntu software center can be contacted by clicking on the taskbar item show in [Figure 2.34](#), and searching for your desired application.

```
ch@PythonForensics: ~  
ch@PythonForensics:~$  
ch@PythonForensics:~$ python  
Python 2.7.3 (default, Aug 1 2012, 05:16:07)  
[GCC 4.6.3] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

FIGURE 2.33
Ubuntu terminal window Python command.

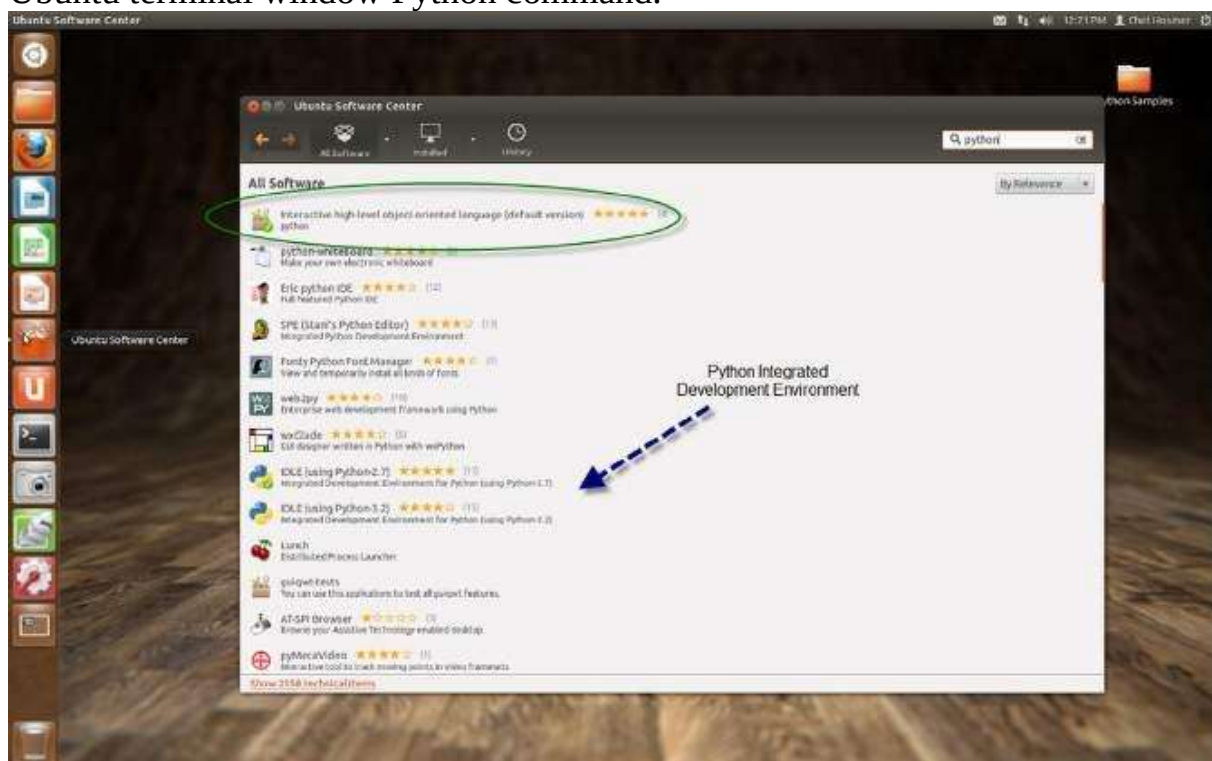


FIGURE 2.34
Ubuntu software center.

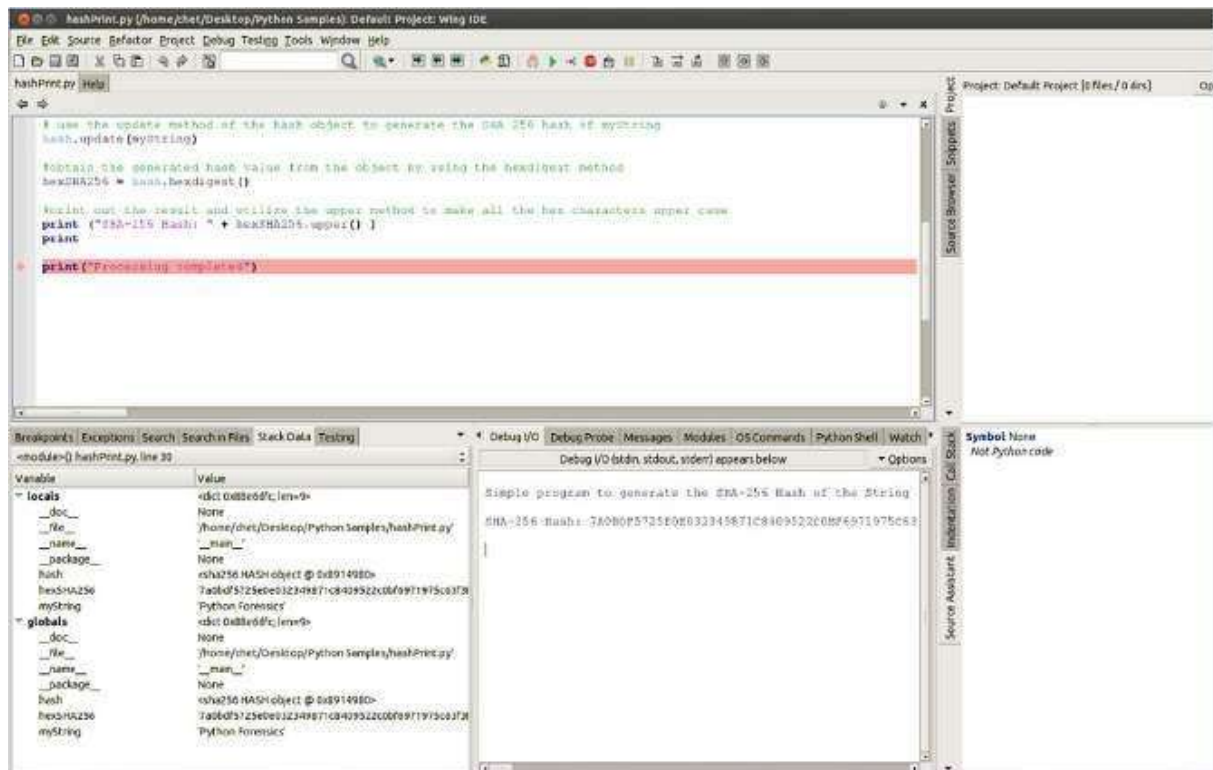


FIGURE 2.35
WingIDE running on Ubuntu 12.04 LTS.

As you see, I have searched for Python and the Software Center displays all the relevant applications and services for me. You notice the first item that I have circled for you is the actual Python environment. The Ubuntu Software Center shows that this application is already installed (notice the checkmark in front of the icon). Also, you see that two different versions of the IDLE IDE are also listed one supporting version 2.7 and the other supporting version 3.2.

I went ahead and installed the WingIDE environment for Ubuntu as this is the environment I have chosen. You can see the screenshot in [Figure 2.35](#) where WingIDE is running under Ubuntu 12.04 LTS, with the same feature set and operation as the Windows version that you saw earlier in this chapter.

PYTHON ON MOBILE DEVICES

Progress is currently being made to port versions of Python to smart mobile

devices (iOS, Windows 8 Phones, and of course Android). The apps that have been created have some limited functionality, but they are worth considering for experimentation or learning. I will take a look at a couple of these in the next section.

iOS Python app

Python for iOS is the version that I find most stable and supported for iPad. In [Figure 2.36](#), you can see an image of the Python 2.7 Shell running on iOS, just

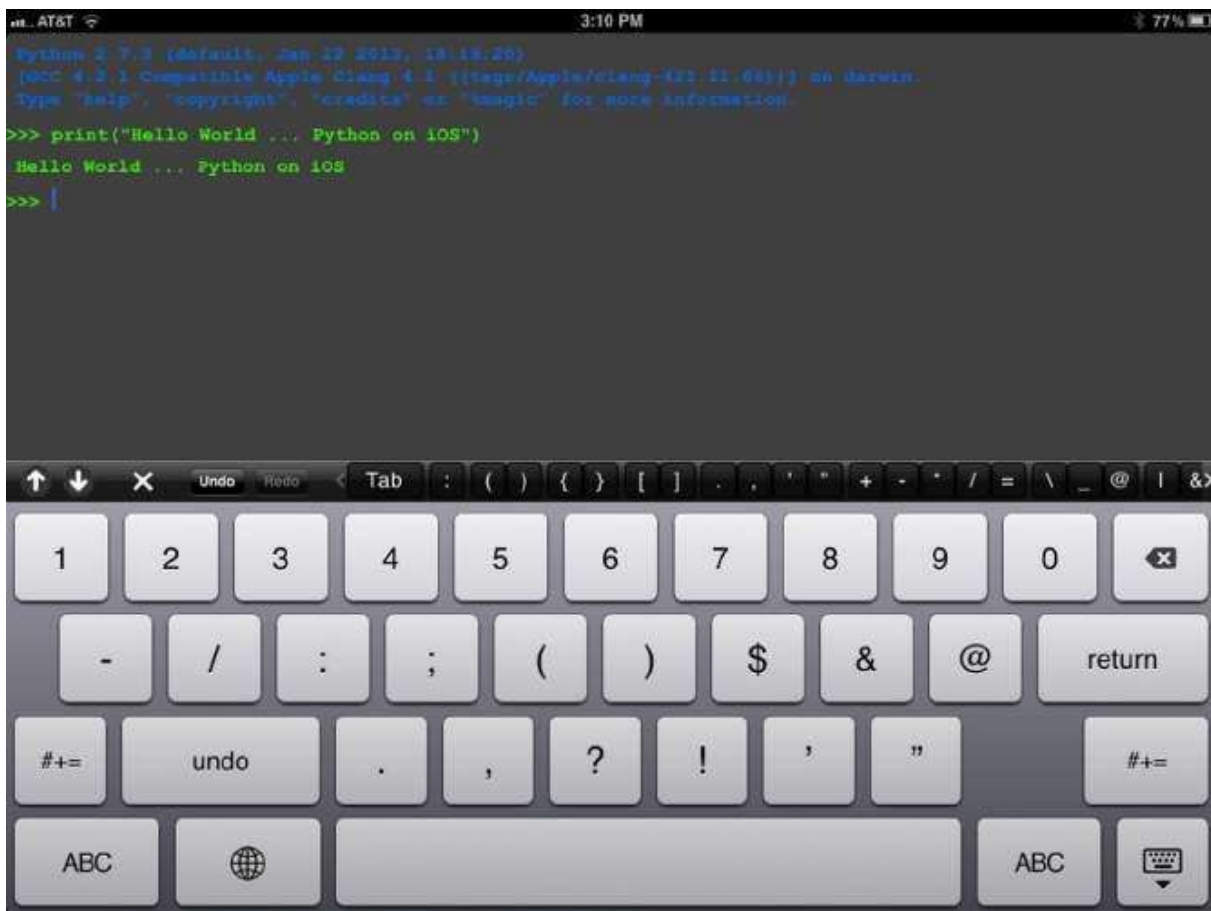


FIGURE 2.36

Python Shell running on iOS.

printing out the similar Hello World message. The Shell and editor have some great features that allow you to experiment with the Python language syntax and modules. The capability is actually quite a bit better than this simple example. I imported one of the earlier examples that performs a SHA256 Hash of a string. At the top of the short program I added a couple of new methods, one contained in the Standard Library module `sys` and one in the Standard Library module

platform. I imported the two modules and wrote the following code.

```
import sys
import platform
print("Platform: "+sys.platform)
print("Machine: "+platform.machine())
```

You can see the results of these new methods and the program in [Figure 2.37](#). You notice that Platform is defined as Darwin and the Machine is identified as an iPad version 1. Correct in both cases. Darwin is the platform name specified by Apple for Mac OS X and of course the machine is my old and still faithful iPad 1.

For those interested in checking out this iOS application from the Apple App Store, [Figure 2.38](#) is a screenshot from the Store with more details about Python for iOS.

The screenshot shows an iOS application interface. At the top, the status bar displays 'AT&T', signal strength, time '3:04 PM', and battery level '79%'. The app title 'hashTest' is centered. Below the title bar, there are three buttons: 'Run', 'Set script args' (with a checkbox), and a 'Menu' button. The main area is split into two panes. The left pane contains Python code with line numbers 00001 to 00037. The code imports 'sys' and 'platform', prints the platform and machine information, and then calculates a SHA-256 hash for the string 'Python Forensics'. The right pane shows the output of the program in a monospaced font. The output includes the platform 'darwin', machine 'iPad1,1', a message about generating a SHA-256 hash, the actual hash value, and a completion message.

```
00001 #
00002 # Python Forensics
00003 # Simple program to generate the SHA-256 one-way cryptographic has of a given string
00004 #
00005 # tell the interpreter to import the hashlib
00006 import hashlib
00007 import sys
00008 import platform
00009
00010 print("Platform: "+sys.platform)
00011 print("Machine: "+platform.machine())
00012
00013 # print a message to the users
00014 print
00015 print( "Simple program to generate the SHA-256 Hash of the String Python Forensics")
00016 print
00017
00018 # create a string with the desired text
00019 myString = "Python Forensics"
00020
00021 # create and object named hash which is of type sha256
00022 hash=hashlib.sha256()
00023
00024 # use the update method of the hash object to generate the SHA 256 hash of myString
00025 hash.update(myString)
00026
00027 #obtain the generated hash value from the object by using the hexdigest method
00028 hexSHA256 = hash.hexdigest()
00029
00030 #print out the result and utilize the upper method to make all the hex characters upper
00031 print ("SHA-256 Hash: " + hexSHA256.upper() )
00032 print
00033
00034 print("Processing completed")
00035
00036
00037
```

Platform: darwin
Machine: iPad1,1

Simple program to generate the SHA-256 Hash
of the String Python Forensics

SHA-256 Hash:
7A869F5125C3ED12345B71C9409532208F6971975C63
F2F6041E2522148B9CF7

Processing completed

FIGURE 2.37

iOS implementation of HashPrint.



FIGURE 2.38
Apple App Store page regarding Python for iOS.

Windows 8 phone

Finally, we can take a look at the Microsoft Windows 8 OS on a Nokia Lumia Phone. Yes—Python on a Phone. I tried several applications from the Microsoft store before I found one that worked pretty well. PyConsole (short for Python Console), shown in the launcher menu in [Figure 2.39](#), ran the sample hashPrint.py program unchanged.

As we progress through the book, we will come back to these devices and others to see how well the more advanced applications work on each of them.

When the app launches it brings up a simple Python Shell as shown in [Figure 2.40](#). I typed in a simple Hello Mobile World message, and you can see that I ran the script. The result appears in the display windows at the bottom of the screen. Turning to the simple SHA256 hashing example shown in [Figure 2.41](#), we see the code and execution of the program producing the same results as the other

platforms.

If you are interested in the Python Console Windows 8 Phone application, [Figure 2.42](#) depicts a screenshot from the Windows Store.



FIGURE 2.39

Windows 8 Phone screenshot of PyConsole launching



FIGURE 2.40
Python Console “Hello World” on a Windows 8 Phone.


```
#
# Python Forensics
# Simple program to generate the SHA-25
#

# tell the interpreter to import the hashlib
import hashlib

# print a message to the users
print
print( "Simple program to generate the SH
print

# create a string with the desired text
myString = "Python Forensics"
```

run cls open save exit

Simple program to generate the SHA-256 Hash of the

SHA-256 Hash: 7A0BDF5725E0E032349871C8409522C

Processing completed

FIGURE 2.41
Windows 8 Phone HashPrint application execution.



FIGURE 2.42

Python Console Windows App Store page.

Summary Questions 51

A VIRTUAL MACHINE

Today, we have virtual machines for everything from complete server infrastructures to specialized applications, standardized development environments, databases, client services, desktops, and more. For those who want a really quick start, I have created a Ubuntu Python Environment that will be available in conjunction with the publication of this book. The environment includes a standard Ubuntu install, all the packages, modules, and Cookbook applications from [Chapters 3 to 11](#). I have also included any test data so you can start experimenting with the simplest or most advanced digital investigation or forensic applications found within. You can access the virtual machine by going

to www.python-forensics.org.

CHAPTER REVIEW

In this chapter we took a very broad, and in some cases deep, look at setting up a Python environment in the Microsoft Windows and Linux environment. We reviewed where to download the latest Python environment from the official source and walked through the steps of installing a Windows Python environment. We also made specific recommendation on the Ubuntu version of choice that include Python version 2.x already installed as part of the OS. We looked at the key differences between Python 2.x and 3.x. We also pulled back the covers a bit on the Python Standard Library and dialed in on some of the key built-in functions, data types, and modules. We even developed a few example applications that exercised the Standard Library and showed how a program written in Python could run without modification on Windows, Linux, iOS, or on a Windows 8 phone. MacOS is also supported and Python 2.x comes with the standard install of the latest versions.

We examined two specific third-party packages that address Natural Language and network applications, respectively. We also examined some of the capabilities that we will need included in an IDE, and took a step-by-step look at the features of WingIDE and briefly covered the IDLE IDE. Finally, we experimented with the Python Shell on all the computing platforms mentioned here.

SUMMARY QUESTIONS

1. What do you think would be the best Python development environment based on the following circumstances?
 - a. A student looking to experiment with Python forensics
 - b. A developer that plans to build and distribute Python investigative solutions
 - c. A laboratory that processes actual cases, but needs specialized tools and

capabilities that do not exist within standard forensic tools

52 CHAPTER 2 Setting up a Python Forensics Environment

2. What are some of the key features you would be looking for in an IDE if you are creating and utilizing Python forensic applications on actual cases or during real investigations?

3. What version of Ubuntu Linux would you choose for your platform of choice and why?
4. How might you utilize mobile platforms like iOS, Android, or Windows 8 devices for investigative purposes when a fully functional Python environment is available for them?
5. What are some of the key considerations for choosing Python 2.x vs. 3.x today?
6. What third-party modules or package would assist in Natural Language Processing?
7. What third-party modules or packages would assist in creating asynchronous network applications?

LOOKING AHEAD

We are now going to transition into the Cookbook chapters of the book, where in each chapter we tackle a specific forensic or digital investigation challenge. We will define the challenge, specify requirements, design an approach, code a solution, and finally test and validate the solution.

As we go forward, I will be slowly introducing new language constructs, packages and modules, debugging methodologies, and good coding practices in each chapter. So, I hope by this time you have setup your Python IDE and are ready to get started.

Additional Resources

Python Programming Language—Official Website, [Python.org](http://python.org).

<http://www.python.org>. The Python Standard Library.

<http://docs.python.org/2/library/>.

The Natural Language Toolkit. nltk.org.

Twisted Network Programming for Python. <http://twistedmatrix.com>.

CHAPTER

Our First Python Forensics App 3

CHAPTER CONTENTS

Introduction

54

Naming Conventions and Other Considerations

.....	55
Constants	55
Local Variable Name	55
Global Variable Name	55
Functions Name	55
Object Name	55
Module	55
Class Names	56
Our First Application “One-Way File System Hashing”	56
Background	56
One-way Hashing algorithms’ Basic Characteristics	56
Popular Cryptographic Hash Algorithms?	57
What are the Tradeoffs Between One-Way Hashing Algorithms?	57
What are the Best-use Cases for One-Way Hashing Algorithms in Forensics? ... 57	
Fundamental Requirements	58
Design Considerations	59
Program Structure	61
Main Function	62
ParseCommandLine	62
WalkPath Function	63
HashFile Function	63
CSVWriter (Class)	63
Logger	63
Writing the Code	63
Code Walk-Through	64
Examining Main—Code Walk-Through	64
ParseCommandLine()	66
ValidatingDirectoryWritable	69
WalkPath	69
HashFile	71
CSVWriter	74
Full Code Listing Pfish.py	75
Full Code Listing _Pfish.py	76
Results Presentation	

83 Chapter Review

.....	86 Summary
Questions	89
Looking Ahead	
.....	89 Additional
Resources	89

Python Forensics 53 © 2014 Elsevier Inc. All rights reserved.

INTRODUCTION

In 1998, I authored a paper entitled “Using SmartCards and Digital Signatures to Preserve Electronic Evidence” (Hosmer, 1998). The purpose of the paper was to advance the early work of Gene Kim, creator of the original Tripwire technology (Kim, 1993) as a graduate student at Purdue. I was interested in advancing the model of using one-way hashing technologies to protect digital evidence, and specifically I was applying the use of digital signatures bound to a SmartCard that provided twofactor authenticity of the signing (Figure 3.1).

Years later I added trusted timestamps to the equation adding provenance, or proof of the exact “when” of the signing.

Two-factor authentication combines a secure physical device such as a SmartCard with a password that unlocks the capability of the card’s. This yields “something held” and “something known.” In order to perform applications like signing, you must be in possession of the SmartCard and you must know the pin or password that unlocks the cards function.

Thus, my interest in applying one-way hashing methods, digital signature algo

Thus, my interest in applying one-way hashing methods, digital signature algo year journey ... so far. The application of these technologies to evidence preservation, evidence identification, authentication, access control decisions and network protocols continues today. Thus I want to make sure that you have a firm understanding of the underlying technologies and the many applications for digital investigation, and of course the use of Python forensics.



FIGURE 3.1

Cryptographic SmartCard.

Naming Conventions and Other Considerations 55

Before I dive right in and start writing code, as promised I want to set up some ground rules for using the Python programming language in forensic applications.

NAMING CONVENTIONS AND OTHER CONSIDERATIONS

During the development of Python forensics applications, I will define the rules and naming conventions that are being used throughout the cookbook chapters in the book. Part of this is to compensate for Python's lack of the enforcement of strongly typed variables and true constants. More importantly it is to define a style that will make the programs more readable, and easier to follow, understand, and modify or enhance.

Therefore, here are the naming conventions I will be using.

Constants

Rule: Uppercase with underscore separation Example: `HIGH_TEMPERATURE`

Local variable name

Rule: Lowercase with bumpy caps (underscores are optional) Example: `currentTemperature`

Global variable name

Rule: Prefix `gl` lowercase with bumpy caps (underscores are optional) Note: Globals should be contained to a single module Example: `gl_maximumRecordedTemperature`

Functions name

Rule: Uppercase with bumpy caps (underscores optional) with active voice Example: `ConvertFahrenheitToCentigrade(...)`

Object name

Rule: Prefix ob_ lowercase with bumpy caps Example: ob_myTempRecorder
Module

Rule: An underscore followed by lowercase with bumpy caps Example:
_tempRecorder

Class names

Rule: Prefix class_ then bumpy caps and keep brief

Example: class_TempSystem

You will see many of these naming conventions in action during this chapter.

OUR FIRST APPLICATION “ONE-WAY FILE SYSTEM HASHING”

The objective for our first Python Forensic Application is as follows:

1. Build a useful application and tool for forensic investigators.
2. Develop several modules along the way that are reusable throughout the book and for future applications.
3. Develop a solid methodology for building Python forensic applications.
4. Begin to introduce more advanced features of the language.

Background

Before we can build an application that performs one-way file system hashing I need to better define one-way hashing. Many of you reading this are probably saying, “I already know what a one-way hashing is, let’s move on.” However, this is such an important underpinning for computer forensics it is worthy of a good definition, possibly even a better one that you currently have.

One-way hashing algorithms’ basic characteristics

1. The one-way hashing algorithm takes a stream of binary data as input; this could be a password, a file, an image of a hard drive, an image of a solid state drive, a network packet, 1’s and 0’s from a digital recording, or basically any continuous digital input.
2. The algorithm produces a message digest which is a compact representation of the binary data that was received as input.
3. It is infeasible to determine the binary input that generated the digest with only the digest. In other words, it is not possible to reverse the process using the digest to recover the stream of binary data that created it.
4. It is infeasible to create a new binary input that will generate a given message digest.
5. Changing a single bit of the binary input data will generate a unique message

digest.

6. Finally, it is infeasible to find two unique arbitrary streams of binary data that produce the same digest.

Table 3.1 Popular One-Way Hashing Algorithms

Length	Algorithm	Creator	(Bits)	Related standard
--------	-----------	---------	--------	------------------

MD5	Ronald Rivest	128		
-----	---------------	-----	--	--

SHA-1	NSA and published by NIST	160		
-------	---------------------------	-----	--	--

SHA-2	NSA and published by NIST	224		
-------	---------------------------	-----	--	--

256				
-----	--	--	--	--

384				
-----	--	--	--	--

512				
-----	--	--	--	--

RIPEMD-160	Hans Dobbertin	160		
------------	----------------	-----	--	--

SHA-3	Guido Bertoni, Joan Daemen, 224, 256, Michael Peeters, and Gilles Van Assche			
-------	--	--	--	--

Popular cryptographic hash algorithms? There are a number of algorithms that produce message digests. [Table 3.1](#) provides background on some of the most popular algorithms.

RFC 1321

FIPS Pub 180 FIPS Pub 180-2 FIPS Pub 180-3 FIPS PUB 180-4

Open Academic Community

FIPS-180-5

What are the tradeoffs between one-way hashing algorithms? The MD5 algorithm is still in use today, and for many applications the speed, convenience, and interoperability have made it the algorithm of choice. Due to attacks on the MD5 algorithm and the increased likelihood of collisions, many organizations are moving to SHA-2 (256 and 512 bits are the most popular sizes). Many organizations have opted to skip SHA-1 as it suffers from some of the same weaknesses as MD5.

Considerations for moving to SHA-3 are still in the future, and it will be a couple of years before broader adoption is in play. SHA-3 is completely different and was designed to be easier to implement in hardware to improve performance

(speed and power consumption) for use in embedded or handheld devices. We will see how quickly the handheld devices' manufacturers adopt this newly established standard.

What are the best-use cases for one-way hashing algorithms in forensics?

Evidence preservation: When digital data are collected (for example, when imaging a mechanical or solid state drive), the entire contents—in other words every bit collected—are combined to create a unique one-way hashing value. Once completed the recalculation of the one-way hashing can be accomplished. If the new calculation matches the original, this can prove that the evidence has not been modified. This assumes of course that the original calculated hash value has been safeguarded against tampering since there is no held secret and the algorithms are available. Anyone could recalculate a hash, therefore the chain of custody of digital evidence, including the generated hash, must be maintained.

Search : One-way hashing values have been traditionally utilized to perform searches of known file objects. For example, if law enforcement has a collection of confirmed child-pornography files, the hashes could be calculated for each file. Then any suspect system could be scanned for the presence of this contraband by calculating the hash values of each file and comparing the resulting hashes to the known list of contraband hash values (those resulting from the child-pornography collection). If matches are found, then the files on the suspect system matching the hash values would be examined further.

Black Listing : Like the search example, it is possible to create a list of known bad hash files. These could represent contraband as with CP example, they could match known malicious code or cyber weapon files or the hashes of classified or proprietary documents. The discovery of hashes matching any of these Black Listed items would provide investigators with key evidence.

White Listing : By creating a list of known good or benign hashes (operating system or application executables, vendor supplied dynamic link libraries or known trustworthy application download files), investigators can use the lists to filter out files that they do not have to examine, because they were previously determined as a good file. Using this methodology you can dramatically reduce the number of files that require examination and then focus your attention on files that are not in the known good hash list.

Change detection : One popular defense against malicious changes to websites,

routers, firewall configuration, and even operating system installations is to hash a “known good” installation or configuration. Then periodically you can re-scan the installation or configuration to ensure no files have changed. In addition, you must of course make sure no files have been added or deleted from the “known good” set.

Fundamental requirements

Now that we have a better understanding of one-way hashing and its uses, what are the fundamental requirements of our one-way file system hash application?

When defining requirements for any program or application I want to define them as succinctly as possible, and with little jargon, so anyone familiar with the domain could understand them—even if they are not software developers. Also, each requirement should have an identifier such that could be traced from definition, through design, development, and validation. I like to give the designers and developers room to innovate, thus I try to focus on WHAT not HOW during requirements definition ([Table 3.2](#)).

Table 3.2 Basic Requirements Requirement Requirement number name

000 Overview

001 Portability

002 Key functions

003 Key results

004 Algorithm selection

005 Error handling Short description

The basic capability we are looking for is a forensic application that walks the file system at a defined starting point (for example, c:\ or /etc) and then generates a one-way hashing value for every file encountered

The application shall support Windows and Linux operating systems. As a general guideline, validation will be performed on Windows 7, Windows 8, and Ubuntu 12.04 LTS environments In addition to the one-way hashing generation, the application shall collect system metadata associated with each file that is hashed. For example, file attributes, file name, and file path at a minimum The application shall provide the results in a standard output file format that offers flexibility The application shall provide a wide diversity when specifying the one-way hashing algorithm(s) to be used

The application must support error handling and logging of all operations

performed. This will include a textual description and a timestamp

Design considerations Now that I have defined the basic requirements for the application I need to factor in the design considerations. First, I would like to leverage or utilize as many of the built-in functions of the Python Standard Library as possible. Taking stock of the core capabilities, I like to map the requirements definition to Modules and Functions that I intend to use. This will then expose any new modules either from third party modules or new modules that need to be developed ([Table 3.3](#)).

One of the important steps as a designer or at least one of the fun parts is to name the program. I have decided to name this first program p-fish short for Pythonfile system hashing.

Next, based on this review of Standard Library functions I must define what modules will be used in our first application:

1. argparse for user input
2. os for file system manipulation
3. hashlib for one-way hashing
4. csv for result output (other optional outputs could be added later)
5. logging for event and error logging
6. Along with useful miscellaneous modules like time, sys, and stat

Table 3.3 Standard Library Mapping Requirement Design considerations Library selection

User input (000, 003, 004)

Walk the file system
(000, 001)

Meta data collection (003)

File hashing (000)

Each of these requirements needs input from the user to accomplish the task. For example, 000 requires the user to specify the starting directory path. 003 requires that the user specify a suitable output format. 004 requires us to allow the user to specify the hash algorithm. Details of the exception handling or default settings need to be defined (if allowed)

This capability requires the program to traverse the directory structure starting at a specific starting point. Also, this must work on both Windows and Linux platforms

This requires us to collect the directory path, filename, owner, modified/access/created times, permissions, and attributes such as read only, hidden, system or archive

I must provide flexibility in the Hashing algorithms that the users could select. I have decided to support the most popular algorithms such as MD5 and several variations of SHA

Result output (003)

To meet this requirement I must be able to structure the program output to support a format that provides flexibility

For this first program I have decided to use the command line parameters to obtain input from the user. Based on this design decision I can leverage the argparse Python Standard Library module

The OS Module from the

Standard Library provides key methods that provide the ability to walk the file system, OS also provides abstraction which will provide cross platform compatibility. Finally, this module contains cross platform capabilities that provide access to metadata associated with files

The Standard Library module hashlib provides the ability to generate one-way hashing values. The library supports common hash algorithms such as “md5,” “sha1,” “sha224,” “sha256,” “sha384,” “sha512.” This should provide a sufficient set of selection for the user The Standard Library offers multiple options that I could leverage. For example, the csv module provides the ability to create comma separated value output, whereas the json module (Java Object Notation) provides encoder and decoders for JSON objects and finally the XML module could be leveraged to create XML output

Continued

Table 3.3 Standard Library Mapping—cont’d Requirement Design considerations

Logging and
error handling

I must expect that errors will occur during our walk of the file system, for example I might not have access to certain files, or certain files may be orphaned, or certain files maybe locked by the operating system or other applications. I need to handle these error conditions and log any notable events. For example, I should log information about the investigator, location, date and time, and information that pertains to the system that are walked

Program structure Next, I need to define the structure of our program, in other words how I intend to put the pieces together. This is critical, especially if our goal is to reuse components of this program in future applications. One way to compose the components is with a couple simple diagrams as shown in [Figures 3.2 and 3.3](#).

Library selection

The Python Standard Library includes a logging facility which I can leverage to report any events or errors that occur during processing

user



p-fish Report

Program Arguments

p-fish

Event and Error Log

p-fish context diagram FIGURE 3.2

Context diagram: Python-file system hashing (p-fish).

Program Arguments

rootPath

hashType reportName ReportName

Generate Report

hashType P-fish Report

HashFile Main resultRecords rootPath fileName

WalkPath eventValue

eventValue

logName LogEvents

Event

and

Error Log

p-fish internal structure

FIGURE 3.3 p-fish internal structure.

The context diagram is very straightforward and simply depicts the major inputs and outputs of the proposed program. A user specifies the program arguments, p-fish takes those inputs and processes (hashes, extracts metadata, etc.) the file system produces a report and any notable events or errors to the “p-fish report” and the “p-fish event and error log” files respectively.

Turning to the internal structure I have broken the program down into five major components. The Main program, ParseCommandLine function, WalkPath function, HashFile functions, CSVWriter class and logger (note logger is actually the Python logger module), that is utilized by the major functions of pfish. I briefly describe the operation of each below and during the code walk through section a more detailed line by line explanation of how each function operates is provided.

Main function

The purpose of the Main function is to control the overall flow of this program. For example, within Main I set up the Python logger, I display startup and

completion messages, and keep track of the time. In addition, Main invokes the command line parser and then launches the WalkPath function. Once WalkPath completes Main will log the completion and display termination messages to the user and the log.

ParseCommandLine

In order to provide smooth operation of p-fish, I leverage parseCommandLine to not only parse but also validate the user input. Once completed, information that is germane to program functions such as WalkPath, HashFile, and CSVWrite is available from parser-generated values. For example, since the hashType is specified by the user, this value must be available to HashFile. Likewise the CSVWriter needs the path where the resulting pfish report will be written, and WalkPath requires the starting or rootPath to start the walk.

WalkPath function

The WalkPath function must start at the root of the directory tree or path and traverse every directory and file. For each valid file encountered it will call the HashFile function to perform the one-way hashing operations. Once all the files have been processed WalkPath will return control back to Main with the number of files successfully processed.

HashFile function

The HashFile function will open, read, hash, and obtain metadata regarding the file in question. For each file, a row of data will be sent to the CSVWriter to be included in the p-fish report. Once the file has been processed, HashFile will return control back to WalkPath in order to fetch the next file.

CSVWriter (class)

In order to provide an introduction to class and object usage I decided to create CSVWriter as a class instead of a simple function. You will see more of this in upcoming cookbook chapters but CSVWriter sets up nicely for a class/object demonstration. The csv module within the Python Standard Library requires that the “writer” be initialized. For example, I want the resulting csv file to have a header row made up of a static set of columns. Then subsequent calls to writer will contain data that fills in each row. Finally, once the program has processed all the files the resulting csv report must be closed. Note that as I walk through the program code you may wonder why I did not leverage classes and objects more for this program. I certainly could have, but felt for the first application I would create a more function-oriented example.

Logger

The built-in Standard Library logger provides us with the ability to write messages to a log file associated with p-fish. The program can write information messages, warning messages, and error messages. Since this is intended to be a forensic application, logging operations of the program is vital. You can expand the program to log additional events in the code, they can be added to any of the `_pfish` functions.

Writing the code

I decided to create two files, mainly to show you how to create your own Python module and also to give you some background on how to separate capabilities. For this first simple application, I created (1) `pfish.py` and (2) `_pfish.py`. As you may recall, all modules that are created begin with an underscore and since `_pfish.py` contains all the support functions for `pfish` I simply named it `_pfish.py`. If you would like to split out the modules to better separate the functions you could create separate modules for the `HashFile` function, the `WalkPath` function, etc. This is a decision that is typically based on how tightly or loosely coupled the functions are, or better stated, whether you wish to reuse individual functions later that need to standalone. If that is the case, then you should separate them out.

In [Figure 3.4](#) you can see my IDE setup for the project `pfish`. You notice the project section to the far upper right that specifies the files associated with the project. I also have both files open—you can see the two tabs far left about half way down

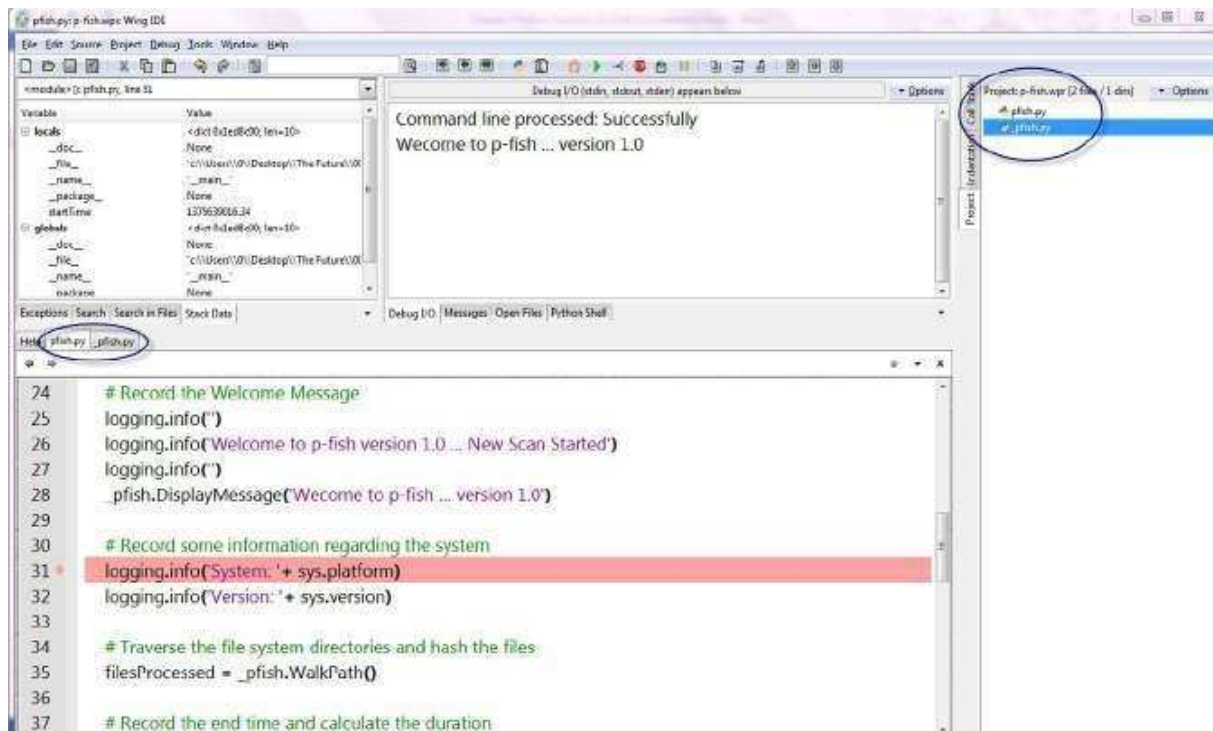


FIGURE 3.4
p-fish WingIDE setup.

where I can view the source code in each of the files. As you would expect in the upper left quadrant, you can see the program is running and the variables are available for inspection. Finally, in the upper center portion of the screen you can see the current display messages from the program reporting that the command line was processed successfully and the welcome message for pfish.

CODE WALK-THROUGH

I will be inserting dialog as I discuss each code section. The code walk-through will give you an in-depth look at all the code associated with the program. I will be first walking through each of the key functions and then will provide you with a complete listing of both files.

Examining main—code walk-through

The embedded commentary is represented in *italics*, while the code itself is represented in a fixed-sized font.

```
# p-fish : Python File System Hash Program
# Author: C. Hosmer
```

```
# July 2013
# Version 1.0
#
```

The main program code is quite straightforward. At the top as you would expect, you see the import statements that make the Python Standard Library modules available for our use. What you have not seen before is the import statement referencing our own module in this case `_pfish`. Since I will be calling functions from Main that exist in our module, the module must import our own module `_pfish`.

```
import logging import time
import sys
import _pfish # Python Library logging functions
# Python Library time manipulation functions # Python Library system specific
parameters # _pfish Support Function Module
```

```
if __name__ == '__main__': PFISH_VERSION = '1.0' # Turn on Logging
```

Next, you can see my initialization of the Python logging system. In this example I have hard-wired the log to be stored in the file aptly named `pFishLog.log`. I set the logging level to `DEBUG` and specified that I wanted the Time and Date recorded for each log event. By setting the level to `DEBUG` (which is the lowest level) this will ensure all messages sent to the logger will be visible.

```
logging.basicConfig(filename='pFishLog.log',level=logging.DEBUG,format='%
(asctime)s %(message)s')
```

Next, I pass control to process the command line arguments by calling the `_pfish.ParseCommandLine()` function. I must prefix the function with `_pfish`, since the function exists in the `_pfish` module. If the parse is successful, the function will return here, if not, it will post a message to the user and exit the program. I will take a deeper look at the operation of `ParseCommandLine()` in the next section.

```
# Process the Command Line Arguments _pfish.ParseCommandLine()
```

I need to record the current starting time of the application in order to calculate elapsed time for processing. I use the Standard Library function `time.time()` to

acquire the time elapsed in seconds since the epoch. Note, forensically this is the time of the system we are running on, therefore if the time is a critical element in your investigation you should sync your system clock accordingly.

```
# Record the Starting Time
startTime = time.time()
```

Next the program posts a message to the log reporting the start of the scan and display this on the user screen only if the verbose option was selected on the command line (more about this when I examine the ParseCommandLine function). Notice that I used a CONSTANT to hold the version number instead of just embedding a magic number. Now we can just modify the CONSTANT in the future. Then anywhere PFISH_VERSION is used it will display the proper version number. I also logged the system platform and version in case there is a question in the future about the system that was used to process these data. This would be a great place to add information about the organization, investigator name, case number, and other information that is relevant to the case.

```
# Post the Start Scan Message to the Log
logging.info('Welcome to p-fish version 1.0 ... New Scan Started')
_pfish.DisplayMessage('Welcome to p-fish ... version 1.0')
```

Note, since I created a constant PFISH_VERSION, we could use that to make the source code easier to maintain. That would look something like:

```
_pfish.DisplayMessage('Welcome to p-fish ... ' + PFISH_VERSION)
```

```
# Record some information regarding the system
logging.info('System:' + sys.platform)
logging.info('Version:' + sys.version)
```

Now the main program launches the WalkPath function within the _pfish module that will traverse the directory structure starting at the predefined root path. This function returns the number of files that were successfully processed by WalkPath and HashFile. As you can see I use this value along with the ending time to finalize the log entries. By subtracting the startTime from the endTime I can determine the number of seconds it took to perform the file system hashing operations. You could convert the seconds into days, hours, minutes, and seconds of course.

```

# Traverse the file system directories and hash the files filesProcessed¼
_pfish.WalkPath()

# Record the end time and calculate the duration
endTime¼ time.time()
duration¼ endTime - startTime

logging.info('Files Processed:'+ str(filesProcessed) ) logging.info('Elapsed
Time:'+ str(duration) +'seconds') logging.info('Program Terminated Normally')
_pfish.DisplayMessage("Program End")
ParseCommandLine()

```

In the design section I made a couple of decisions that drove the development:

I decided that this first application would be a command line program. 1.
 2. I decided to provide several options to the user to manipulate the behavior of the program. This has driven the design and implementation of the command line options.

Based on this I provided the following command line options to the program.

Option Description Notes

-v Verbose, if this option is specified then any calls to the DisplayMessage() function will be displayed to the standard output device, otherwise the program will run silently
 --MD5 Hash type selection, the user must
 --SHA256 specify the one-way hashing
 --SHA512 algorithm that would be utilized
 -d rootPath, this allows the user to specify the starting or root path for the walk
 -r reportPath, this allows the user to specify the directory where the resulting .csv file will be written The selection is mutually exclusive and at least one must be selected or the program will abort

The directory must exist and must be readable or the program will abort

The directory must exist and must be writable or the program will abort

Even though at first some of these requirements might seem difficult, the argparse Standard Library provides great flexibility in handling them. This allows us to catch any possible user errors prior to program execution and also provides us with a way to report problems to the user to handle the exceptions.

```
def ParseCommandLine():
```

The majority of the process of using argparse is knowing how to setup the parser. If you set the parser up correctly it will do all the hard work for you. I start by creating a new parser named “parser” and simply give it a description. Next, I add a new argument in this case `-v` or verbose. The option is `-v` and the resulting variable that is used is `verbose`. The help message associated with the argument is used by the help system to inform the user on how to use `pfish`. The `-h` option is built-in and requires no definition.

```
parser¼ argparse.ArgumentParser('Python file system hashing .. p-fish')
parser.add_argument('-v', '--verbose', help¼'allows progress messages to be
displayed', action¼'store_true')
```

The next section defines a mutually exclusive group of arguments for selecting the specific hash type the user would like to generate. If you wanted to add in another option, for example `sha384`, you would simply add another argument to the group and follow the same format. Since I specified under the `add_mutually_exclusive_group` the option `required¼ True`, argparse will make sure that the user has only specified one argument and at least one.

```
# setup a group where the selection is mutually exclusive # # and required.
group¼ parser.add_mutually_exclusive_group(required¼ True)
```

```
group.add_argument( '--md5', help¼'specifies MD5 algorithm',
action¼'store_true')
```

```
group.add_argument('--sha256', algorithm', action¼'store_true') help ¼'specifies
SHA256
```

```
group.add_argument('--sha512', algorithm', action¼'store_true') help ¼'specifies
SHA512
```

Next I need to specify the starting point of our walk, and where the report should be created. This works the same as the previous setup, except I have added the `type` option. This requires argparse to validate the type I have specified. In the case of the `-d` option, I want to make sure that the `rootPath` exists and is readable. For the `reportPath`, it must exist and be writable. Since argparse does not have builtin functions to validate a directory, I created the functions `ValidateDirectory()` and `ValidateDirectoryWritable()`. They are almost identical

and they use Standard Library operating system functions to validate the directories as defined.

```
parser.add_argument( '-d','--rootPath', type¼  
ValidateDirectory, required¼True, help¼"specify the root path for hashing)"
```

```
parser.add_argument( '-r','--reportPath', type¼  
ValidateDirectoryWritable, required¼True, help¼"specify the path for reports  
and logs will be written)"
```

```
# create a global object to hold the validated arguments, # # these will be  
available then to all the Functions within # the _pfish.py module
```

```
global gl_args  
global gl_hashType
```

Now the parser can be invoked. I want to store the resulting arguments (once validated) in a global variable so they can be accessible by the functions within the _pfish module. This would be a great opportunity to create a class to handle this which would avoid the use of the global variables. This is done in [Chapter 4](#).

```
gl_args ¼ parser.parse_args()
```

If the parser was successful (in other words argparse validated the command line parameters), I want to determine which hashing algorithm the user selected. I do that by examining each value associated with the hash types. If the user selected sha256 for example, the gl_args.sha256 would be True and md5 and sha512 would be false. Therefore, by using a simple if/elif language routine I can determine which was selected.

```
if gl_args.md5:  
    gl_hashType ¼'MD5'  
elif gl_args.sha256:  
    gl_hashType ¼'SHA256'  
elif gl_args.sha512:  
    gl_hashType ¼'SHA512'  
else:  
    gl_hashType ¼ "Unknown"  
logging.error("Unknown Hash Type Specified")
```

```
DisplayMessage("Command line processed: Successfully)" return
```

ValidatingDirectoryWritable

As mentioned above, I needed to create functions to validate the directories provided by the users for both the report and starting or root path of the Walk. I accomplish this by leveraging the Python Standard Library module `os`. I leverage both the `os.path.isdir` and `os.access` methods associated with this module.

```
def ValidateDirectoryWritable(theDir):
```

I first check to see if in fact the directory string that the user provided exists. If the test fails then I raise an error within `argparse` and provide the message “Directory does not exist.” This message would be provided to the user if the test fails.

```
# Validate the path is a directory
if not os.path.isdir(theDir):
    raise argparse.ArgumentTypeError('Directory does not exist')
```

Next I validate that write privilege is authorized to the directory and once again if the test fails I raise an exception and provide a message.

```
# Validate the path is writable
if os.access(theDir, os.W_OK):
    return theDir
else:
    raise argparse.ArgumentTypeError('Directory is not writable')
```

Now that I have completed the implementation of the `ParseCommandLine` function, let us examine a few examples of how the function rejects improper command line arguments. In [Figure 3.5](#), I created four improperly formed command lines:

- (1) I mistyped the root directory as `TEST_DIR` instead of simply `TESTDIR`
- (2) I mistyped the `–sha512` parameter as `–sha521`
- (3) I specified two hash types `–sha512` and `–md5`
- (4) Finally, I did not specify any hash type

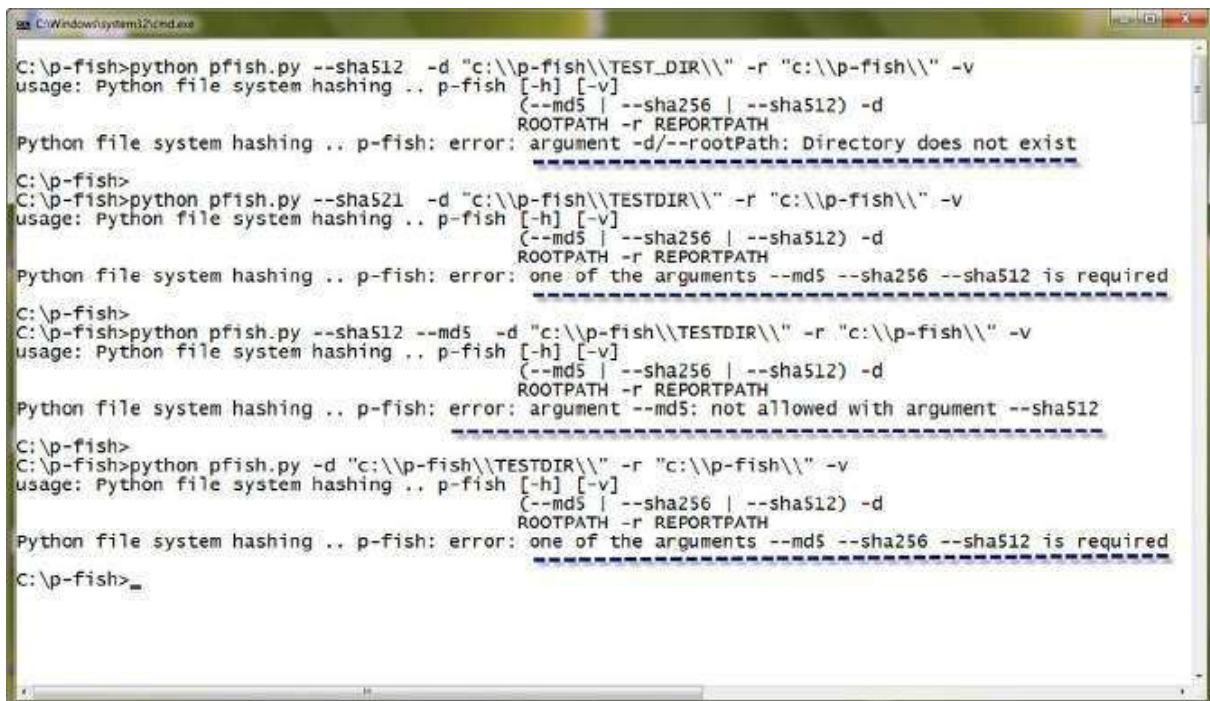
As you can see in each case, `ParseCommandLine` rejected the command.

In order to get the user back on track they simply have to utilize the `–h` or `help`

option as shown in [Figure 3.6](#) to obtain the proper command line argument instructions.

WalkPath

Now let us walk through the WalkPath function that will traverse the directory structure, and for each file will call the HashFile function. I think you will be pleasantly surprised how simple this is.



```
C:\p-fish>python pfish.py --sha512 -d "c:\\p-fish\\TEST_DIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
          (--md5 | --sha256 | --sha512) -d
          ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: argument -d/--rootPath: Directory does not exist

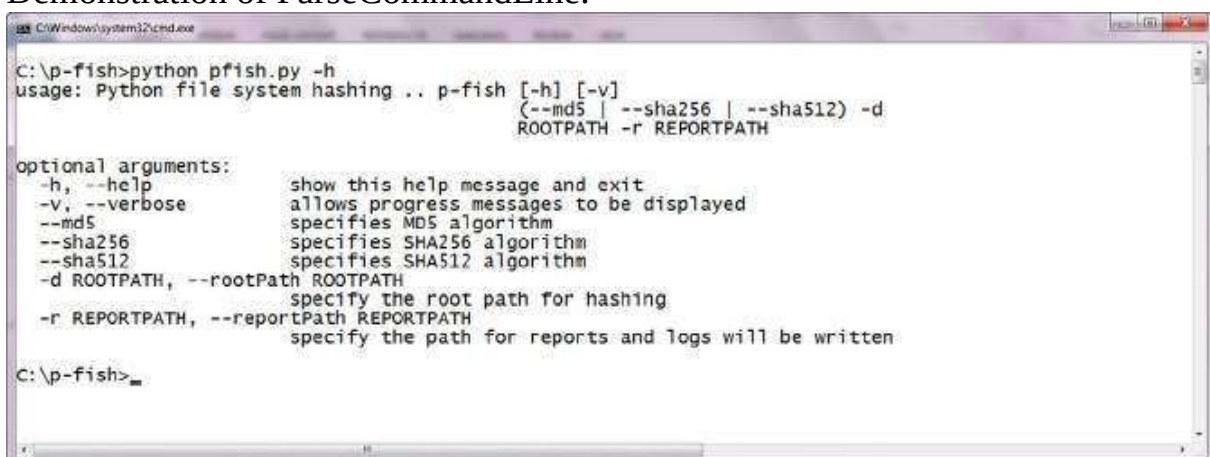
C:\p-fish>
C:\p-fish>python pfish.py --sha512 -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
          (--md5 | --sha256 | --sha512) -d
          ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: one of the arguments --md5 --sha256 --sha512 is required

C:\p-fish>
C:\p-fish>python pfish.py --sha512 --md5 -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
          (--md5 | --sha256 | --sha512) -d
          ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: argument --md5: not allowed with argument --sha512

C:\p-fish>
C:\p-fish>python pfish.py -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
usage: Python file system hashing .. p-fish [-h] [-v]
          (--md5 | --sha256 | --sha512) -d
          ROOTPATH -r REPORTPATH
Python file system hashing .. p-fish: error: one of the arguments --md5 --sha256 --sha512 is required

C:\p-fish>
```

FIGURE 3.5
Demonstration of ParseCommandLine.



```
C:\p-fish>python pfish.py -h
usage: Python file system hashing .. p-fish [-h] [-v]
          (--md5 | --sha256 | --sha512) -d
          ROOTPATH -r REPORTPATH

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose          allows progress messages to be displayed
  --md5                 specifies MD5 algorithm
  --sha256              specifies SHA256 algorithm
  --sha512              specifies SHA512 algorithm
  -d ROOTPATH, --rootPath ROOTPATH
                        specify the root path for hashing
  -r REPORTPATH, --reportPath REPORTPATH
                        specify the path for reports and logs will be written

C:\p-fish>
```

FIGURE 3.6
pfish -h command.

```
def WalkPath():
```

I first initialize the variable processCount in order to count the number of successfully processed files and I post a message to the log file to document the root path value.

```
processCount = 0
errorCount = 0
log.info('Root Path:' + gl_args.rootPath)
```

Next I initialize the CSVWriter with the reportPath provided on the command line by the user. I also provide the hashType selected by the user so it can be included in the Header line of the CSV file. I will cover the CSVWriter class later in this chapter.

```
oCVS = CSVWriter(gl_args.reportPath+'fileSystemReport.csv', gl_hashType)
```

```
# Create a loop that process all the files starting
# at the rootPath, all sub-directories will also be
# processed
```

Next I create a loop using the os.walk method and the rootpath specified by the user. This will create a list of file names that is processed in the next loop. This is done for each directory found within the path.

```
for root, dirs, files in os.walk(gl_args.rootPath):
    # for each file obtain the filename and call the # HashFile Function
```

The next loop processes each file in the list of files and calls the function HashFile with the file name joined with the path, along with the simple file name for use by HashFile. The call also passes HashFile with access to the CVS writer so that the results of the hashing operations can be written to the CVS file.

```
for file in files:
    fname = os.path.join(root, file)
    result = HashFile(fname, file, oCVS)
```

```
# if successful then increment ProcessCount
The process and error counts are incremented accordingly
```

```
if result is True:
    processCount += 1
# if not successful, the increment the ErrorCount
else:
    errorCount += 1
```

Once all the directories and files have been processed the CVSWriter is closed and the function returns to the main program with the number of successfully processed files.

```
oCVS.writerClose()
return(processCount)
```

HashFile

Below is the code for the HashFile function, it is clearly the longest for this program, but also quite simple and straightforward. Let us walk through the process.

```
def HashFile(theFile, simpleName, o_result):
```

For each file several items require validation before we attempt to hash the file.

- (1) Does the path exist
- (2) Is the path a link instead of an actual file
- (3) Is the file real (making sure it is not orphaned)

For each of these tests there is a corresponding log error that is posted to the log file if failure occurs. If the file is bypassed the program will simply return to WalkFile and process the next file.

```
# Verify that the path is valid
if os.path.exists(theFile):
#Verify that the path is not a symbolic link
if not os.path.islink(theFile):
#Verify that the file is real
if os.path.isfile(theFile):
```

The next part is a little tricky. Even through our best efforts to determine the existence of the file, there may be cases where the file cannot be opened or read. This could be caused by permission issues, the file is locked or possibly corrupted. Therefore, I utilize the try methods while attempting to open and then read from the files. Note that I'm careful to open the file as read-only the "rb"

option. Once again if an error occurs a report is generated and logged and the program moves on to the next file.

```
try:
#Attempt to open the file
f = open(theFile,'rb')

except IOError:
#if open fails report the error log.warning('Open Failed:'+ theFile) return

else:
try:
# Attempt to read the file
rd = f.read()

except IOError:
# if read fails, then close the file and # report error
f.close()
log.warning('Read Failed:'+ theFile) return

else:
#success the file is open and we can #read from it
#lets query the file stats
```

Once the file has been successfully opened and verified that reading from the file is allowed, I extract the attributes associated with the file. These include owner, group, size, MAC times, and mode. I will include these in the record that is posted to the CSV file.

```
theFileStats = os.stat(theFile)
(mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime) = os.stat(theFile)

# Display progress to the user DisplayMessage("Processing File: " + theFile)

# convert the file size to a string fileSize = str(size)
# convert the MAC Times to strings

modifiedTime = time.ctime(mtime) accessTime = time.ctime(atime) createdTime
= time.ctime(ctime)
```

```
# convert the owner, group and file mode
```

```
ownerID ¼ str(uid) groupID ¼ str(gid) fileMode ¼ bin(mode)
```

Now that the file attributes have been collected the actual hashing of the file occurs. I need to hash the file as specified by the user (i.e., which one-way hashing algorithm should be utilized). I'm using the Python Standard Library module hashlib as we experimented with in [Chapter 2](#).

```
#process the file hashes
```

```
if gl_args.md5:
```

```
#Calcuation the MD5
```

```
hash ¼ hashlib.md5()
```

```
hash.update(rd)
```

```
hexMD5 ¼ hash.hexdigest()
```

```
hashValue ¼ hexMD5.upper()
```

```
elif gl_args.sha256:
```

```
#Calculate the SHA256
```

```
hash¼hashlib.sha256()
```

```
hash.update(rd)
```

```
hexSHA256 ¼ hash.hexdigest() hashValue ¼ hexSHA256.upper()
```

```
elif gl_args.sha512:
```

```
#Calculate the SHA512
```

```
hash¼hashlib.sha512()
```

```
hash.update(rd)
```

```
hexSHA512 ¼ hash.hexdigest() hashValue ¼ hexSHA512.upper()
```

```
else:
```

```
log.error('Hash not Selected')
```

```
#File processing completed
```

```
#Close the Active File
```

Now that processing of the file is complete the file must be closed. Next I use the CSV class to write out the record to the report file and return successfully to the caller in this case WalkPath.

```
f.close()
# write one row to the output file
```

```
o_result.writeCSVRow(simpleName, theFile, fileSize, modifiedTime,
accessTime, createTime, hashValue, ownerID, groupID, mode)
return True
```

This section posts the warning messages to the log file relating to problems encountered processing the file.

```
else:
log.warning('[ '+ repr(simpleName) +', Skipped NOT a File'+']') return False
```

```
else:
log.warning('[ '+ repr(simpleName) +', Skipped Link NOT a File'+']')
return False
```

```
else:
log.warning('[ '+ repr(simpleName) +', Path does NOT exist'+']') return False
```

CSVWriter

The final code walk-through section I will cover in this chapter is the CSVWriter. As I mentioned earlier, I created this code as a class instead of a function to make this more useful and to introduce you to the concept of classes in Python. The class only has three methods, the constructor or init, writeCSVRow, and writerClose. Let us examine each one.

```
class _CSVWriter:
```

The constructor or init method accomplishes three basic initializations: (1)

Opens the output csvFile

(2) Initializes the csv.writer

(3) Writes the header row with the names of each column

If any failure occurs during the initialization an exception is thrown and a log entry is generated.

```
def __init__(self, fileName, hashType):
```

```
try:
```

```
# create a writer object and write the header row self.csvFile ¼
```

```

open(fileName,'wb')
self.writer = csv.writer(self.csvFile,

delimiter = ',', quoting=csv.QUOTE_ALL) self.writer.writerow(
('File','Path','Size',
'Modified Time','Access Time','Created Time', hashType,'Owner','Group','Mode')
)
except:
log.error('CSV File Failure')

```

The second method writeCSVRow receives a record from HashFile upon successful completion of each file hash. The method then uses the csv writer to actually place the record in the report file.

```

def writeCSVRow(self, fileName, filePath, fileSize, mTime, aTime, cTime,
hashVal, own, grp, mod):

```

```

self.writer.writerow( (fileName, filePath, fileSize, mTime, aTime, cTime,
hashVal, own, grp, mod))

```

Finally, the writeClose method, as you expect, simply closes the csvFile.

```

def writeClose(self):
self.csvFile.close()

```

Full code listing pfish.py

```

#
# p-fish : Python File System Hash Program # Author: C. Hosmer # July 2013
# Version 1.0
#
import logging
import time
import sys
import _pfish
# Python Standard Library Logger
# Python Standard Library time functions # Python Library system specific
parameters # _pfish Support Function Module

if __name__ == '__main__':
PFISH_VERSION = '1.0'

```

```

# Turn on Logging
logging.basicConfig(filename='pFishLog.log',level=logging.DEBUG,
format'%(asctime)s %(message)s')

# Process the Command Line Arguments _pfish.ParseCommandLine()

# Record the Starting Time startTime = time.time()

# Record the Welcome Message
logging.info('')
logging.info('Welcome to p-fish version'+PFISH_VERSION +'...New Scan
Started')
logging.info('')
_pfish.DisplayMessage('Welcome to p-fish ... version'+
PFISH_VERSION)

# Record some information regarding the system logging.info('System:'+
sys.platform)
logging.info('Version:'+ sys.version)

# Traverse the file system directories and hash the files filesProcessed = 0
_pfish.WalkPath()

# Record the end time and calculate the duration endTime = time.time()
duration = endTime - startTime
logging.info('Files Processed:'+ str(filesProcessed)) logging.info('Elapsed
Time:'+ str(duration)+'seconds') logging.info('')
logging.info('Program Terminated Normally')
logging.info('')

_pfish.DisplayMessage("Program End")
Full code listing _pfish.py

#
# pfish support functions, where all the real work gets done #

# Display Message() # HashFile()
# ValidateDirectory() #

import os

```



```
import stat
import time
import hashlib
import argparse
```

```
import csv ParseCommandLine() WalkPath() class _CVSWriter
ValidateDirectoryWritable()
```

```
#Python Standard Library - Miscellaneous operating system interfaces
#Python Standard Library - functions for interpreting os results
#Python Standard Library - Time access and conversions functions
#Python Standard Library - Secure hashes and message digests
#Python Standard Library - Parser for commandline options, arguments
#Python Standard Library - reader and writer for csv files
import logging #Python Standard Library – logging facility
```

```
log ¼ logging.getLogger('main._pfish')
```

```
#
# Name: ParseCommand() Function
#
# Desc: Process and Validate the command line arguments # use Python
Standard Library module argparse #
# Input: none
#
# Actions:
#
#
```

```
# # Uses the standard library argparse to process the command line
establishes a global variable gl_args where any of the functions can
obtain argument information
```

```
def ParseCommandLine(): parser¼ argparse.ArgumentParser('Python file system
hashing .. p-fish')
parser.add_argument('- v','—verbose', help¼'allows progress messages to be
displayed', action¼'store_true')
```

```
# setup a group where the selection is mutually exclusive and required.
```



```

# Desc: Processes a single file which includes performing a hash of the
file
# and the extraction of metadata regarding the file processed # use Python
Standard Library modules hashlib, os, and sys #
# Input: theFile ¼ the full path of the file
# simpleName ¼ just the filename itself
#
# Actions:
# Attempts to hash the file and extract metadata # Call GenerateReport for
successful hashed files #
def HashFile(theFile, simpleName, o_result):

# Verify that the path is valid if os.path.exists(theFile): #Verify that the path is
not a symbolic link if not os.path.islink(theFile):
#Verify that the file is real if os.path.isfile(theFile):

try :
#Attempt to open the file
f ¼ open(theFile, 'r b')

except IOError :
#if open fails report the error log.warning('Open Failed:'+ theFile) return

else:
try:
# Attempt to read the file
rd ¼ f.read()
except IOError:

# if read fails, then close the file and report error
f.close()
log.warning('Read Failed:'+ theFile) return

else :
#success the file is open and we can read from it #lets query the file stats

theFileStats ¼ os.stat(theFile)
(mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime) ¼ os.stat(theFile)

```

[illegible]

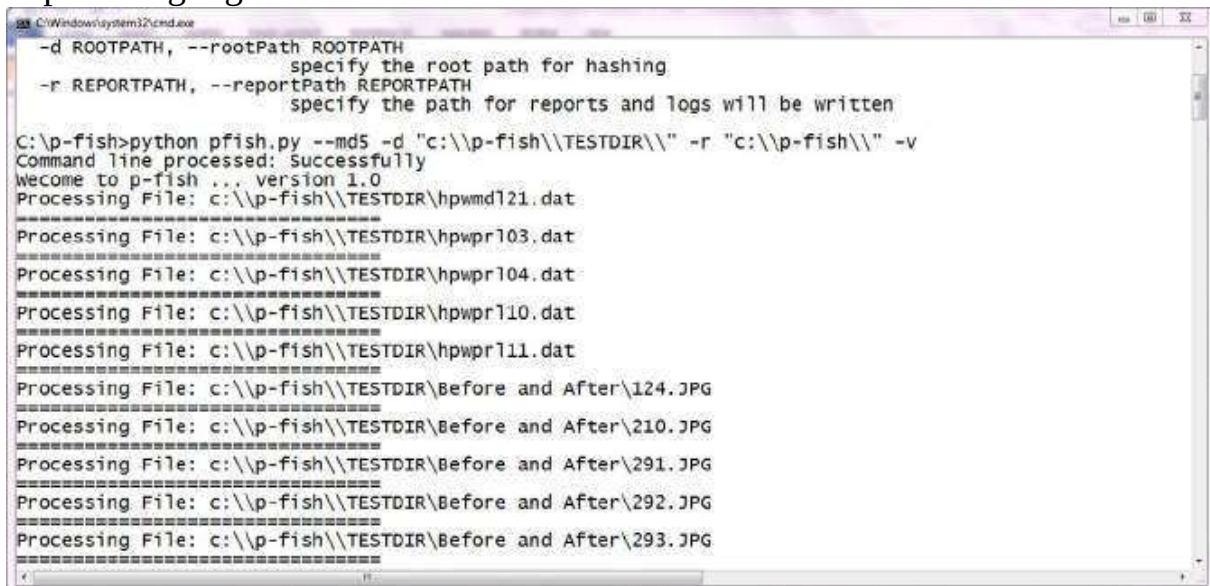
[illegible]

The -v or verbose option was selected and the program displayed information regarding every file processed was selected as expected.

In Figure 3.8, I examine the c:\p-fish directory and discover that two files were created there, which are the two resulting files for the pfish.py.

fileSystemReport.csv1.

2. pFishLog.log



```
C:\Windows\system32\cmd.exe
-d ROOTPATH, --rootPath ROOTPATH
    specify the root path for hashing
-r REPORTPATH, --reportPath REPORTPATH
    specify the path for reports and logs will be written

C:\p-fish>python pfish.py --md5 -d "c:\\p-fish\\TESTDIR\\" -r "c:\\p-fish\\" -v
Command line processed: successfully
Welcome to p-fish ... version 1.0
Processing File: c:\\p-fish\\TESTDIR\\hpwmd121.dat
=====
Processing File: c:\\p-fish\\TESTDIR\\hpwpr103.dat
=====
Processing File: c:\\p-fish\\TESTDIR\\hpwpr104.dat
=====
Processing File: c:\\p-fish\\TESTDIR\\hpwpr110.dat
=====
Processing File: c:\\p-fish\\TESTDIR\\hpwpr111.dat
=====
Processing File: c:\\p-fish\\TESTDIR\\Before and After\\124.JPG
=====
Processing File: c:\\p-fish\\TESTDIR\\Before and After\\210.JPG
=====
Processing File: c:\\p-fish\\TESTDIR\\Before and After\\291.JPG
=====
Processing File: c:\\p-fish\\TESTDIR\\Before and After\\292.JPG
=====
Processing File: c:\\p-fish\\TESTDIR\\Before and After\\293.JPG
=====
```

FIGURE 3.7

Test run of pfish.py.

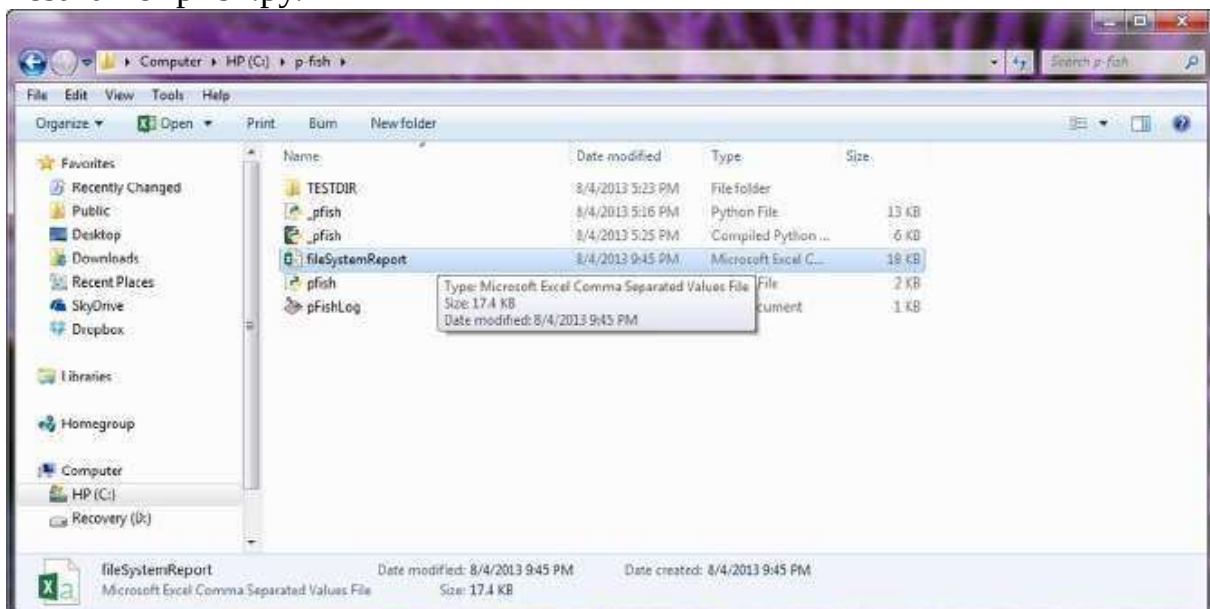


FIGURE 3.8

Result directory after pfish execution.

By choosing to leverage the Python csv module to create the report file Windows already recognizes it as a file that Microsoft Excel can view. Opening the file we see the results in [Figure 3.9](#), a nicely formatted column report that can now manipulate with Excel (sort columns, search for specific values, arrange in date order, and examine each of the results). You notice that the hash value is in a column named MD5 that is labeled as such because I passed the appropriate heading value during the initialization of the csv.

File	Path	Name	Modified Time	Access Time	Created Time	MD5	Owner
hpvr101.dat	c:\p-fish\TESTDIR\hpvr101.dat	574	Wed Jul 26 17:28:00 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	48007F93085190C8BAF8727B3F5E7E	0
hpvr102.dat	c:\p-fish\TESTDIR\hpvr102.dat	462	Tue Feb 16 22:58:59 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	A60140082874F58A1896464642D82894	0
hpvr104.dat	c:\p-fish\TESTDIR\hpvr104.dat	897	Wed Jul 26 17:28:04 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	40B2043D3D1611C48946C6243D94AD7	0
hpvr110.dat	c:\p-fish\TESTDIR\hpvr110.dat	210	Tue Feb 16 23:30:16 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	8C2B57E2D0C1D3C42DA4F4904C481877E	0
hpvr111.dat	c:\p-fish\TESTDIR\hpvr111.dat	189	Wed Jul 26 17:28:04 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	182DF0C64C8AC0C718B71E17129B842	0
114.jpg	c:\p-fish\TESTDIR\Before and After\114.jpg	50874	Sun Apr 04 12:39:04 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	7548324685321004C2B8B719C9A3	0
210.jpg	c:\p-fish\TESTDIR\Before and After\210.jpg	1360810	Wed Aug 18 20:18:04 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	1711142B944088E4C2B4C121781F6262A	0
291.jpg	c:\p-fish\TESTDIR\Before and After\291.jpg	1434599	Tue Dec 26 17:38:04 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	0A3F5B8C1D9816C45C1AC74157C3429	0
392.jpg	c:\p-fish\TESTDIR\Before and After\392.jpg	2458849	Tue Dec 26 17:38:04 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	713E3CFE0D9F2D048D94C4B082F8C021	0
1199.jpg	c:\p-fish\TESTDIR\Before and After\1199.jpg	1824754	Tue Dec 26 17:38:21 2010	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	00F8F5AC0D48277E4D0F3D8C4F6A7A	0
Appra1ea1.docx	c:\p-fish\TESTDIR\Before and After\Appra1ea1.docx	1057774	Sun Apr 05 18:12:51 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	04800F8E13017D8F8F17C0C48C2D834	0
Appra1ea2.docx	c:\p-fish\TESTDIR\Before and After\Appra1ea2.docx	1055448	Sun Apr 05 18:12:42 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	831E034A4C70728F8B8784999999	0
DECT0401.jpg	c:\p-fish\TESTDIR\Before and After\DECT0401.jpg	1714334	Thu Nov 13 12:51:47 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	17043997DFA3482C3275E8B4C2A8A4	0
DECT0402.jpg	c:\p-fish\TESTDIR\Before and After\DECT0402.jpg	1462238	Thu Nov 13 12:51:38 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	4AAG208187D94642408FAC17484EE2	0
DECT0403.jpg	c:\p-fish\TESTDIR\Before and After\DECT0403.jpg	1894864	Thu Nov 13 12:51:11 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	181D6AC3D8C7804A8A1984015C78A1	0
DECT0404.jpg	c:\p-fish\TESTDIR\Before and After\DECT0404.jpg	2646144	Thu Nov 13 12:51:06 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	6B8C8C1D997D88AFA8A240C1D01E	0
DECT0405.jpg	c:\p-fish\TESTDIR\Before and After\DECT0405.jpg	5944310	Thu Nov 13 12:48:40 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	ACD84324E3D1C884C1C81D8C753D	0
DECT0406.jpg	c:\p-fish\TESTDIR\Before and After\DECT0406.jpg	1888942	Thu Nov 13 12:48:35 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	34C09D8B8A34C7AC4C40DFA3F8E8	0
DECT0407.jpg	c:\p-fish\TESTDIR\Before and After\DECT0407.jpg	1448477	Thu Nov 13 12:47:09 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	18A4C824239F8F8F7C138A1F0B881	0
DECT0418.jpg	c:\p-fish\TESTDIR\Before and After\DECT0418.jpg	1679516	Thu Nov 12 12:42:57 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	784C97E7349386558C7488D2B3CA	0
DECT0419.jpg	c:\p-fish\TESTDIR\Before and After\DECT0419.jpg	1807891	Thu Nov 12 12:42:54 2008	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	1A02BAC99D0430671870478F1894	0
114.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\114.jpg	1236	Sat Feb 26 10:42:06 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	8E113A8A283F0C4C2D468A8A9774	0
210.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\210.jpg	1780	Sat Feb 26 10:42:19 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	4E8B318E87734621D9848D3C1E46976	0
291.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\291.jpg	1479	Sat Feb 26 10:44:40 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	0D277C491180B87875B877768829	0
392.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\392.jpg	1830	Sat Feb 26 10:45:48 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	8E8698988F7F22F2C3D8A4948C799	0
1199.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\1199.jpg	1820	Sat Feb 26 10:46:01 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	8DCCD9A8A0C7790DA8E18F73D2880	0
Appra1ea1.docx	c:\p-fish\TESTDIR\Mayan Glyphs\Appra1ea1.docx	1369	Sat Feb 26 10:46:20 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	7158B29D0F04B0F99F9C946039F0D2	0
Appra1ea2.docx	c:\p-fish\TESTDIR\Mayan Glyphs\Appra1ea2.docx	1597	Sat Feb 26 10:47:54 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	06442A8A3C20F872B8F08E8A0F379C1	0
DECT0401.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0401.jpg	1810	Sat Feb 26 10:48:13 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	5932C8A1C2C03A2D087458A058648	0
DECT0402.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0402.jpg	1802	Sat Feb 26 10:48:25 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	49683111898212F3F8E8C4842447E7	0
DECT0403.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0403.jpg	1763	Sat Feb 26 10:48:40 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	4760C0C0C08070C0A8B070303031E22	0
DECT0404.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0404.jpg	1886	Sat Feb 26 10:49:03 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	2182C753F9777847C2182988A8A386	0
DECT0405.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0405.jpg	1797	Sat Feb 26 10:50:00 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	70D8A80A76E10E13748D17101A216	0
DECT0406.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0406.jpg	1905	Sat Feb 26 10:50:39 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	3A4935844E187328295678A6E83048	0
DECT0407.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0407.jpg	1707	Sat Feb 26 10:50:49 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	864D3AC3C1B40477707D040C03DA3	0
DECT0418.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0418.jpg	1787	Sat Feb 26 10:51:57 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	28A7B8C30438A8A3A42402E2C1A439	0
DECT0419.jpg	c:\p-fish\TESTDIR\Mayan Glyphs\DECT0419.jpg	1911	Sat Feb 26 10:51:17 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	38E84474FC93F018A4F78A8D094FED7	0
Appra1ea1.docx	c:\p-fish\TESTDIR\Mayan Glyphs\Appra1ea1.docx	1754	Sat Feb 26 10:51:29 2011	Sun Aug 04 17:28:39 2013	Sun Aug 04 17:28:39 2013	8F2E8A62A0D48B118E8C851E60E	0

FIGURE 3.9
Examining the Result File with Microsoft Excel.

```

2013-08-04 21:45:11.042
2013-08-04 21:45:11.042 welcome to p-fish version 1.0 ... New Scan Started
2013-08-04 21:45:11.059
2013-08-04 21:45:11.059 system: win32
2013-08-04 21:45:11.059 Version: 2.7.5 (default, May 15 2013, 22:43:36) [Msc v.1500 32 bit (Intel)]
2013-08-04 21:45:11.059 Root Path: c:\p-fish\TESTDIR\
2013-08-04 21:45:12.042 Files Processed: 82
2013-08-04 21:45:12.042 Elapsed Time: 0.999000072479 seconds
2013-08-04 21:45:12.042
2013-08-04 21:45:12.042 Program Terminated Normally
2013-08-04 21:45:12.042

```

FIGURE 3.10
Contents of the pFishLog file.

The generated pFishLog.log file results are depicted in [Figure 3.10](#). As expected we find the welcome message, the details regarding the Windows environment, the root path that was specified by the user, the number of files processed, and the elapsed time of just under 1 s. In this example no errors were encountered and the program terminated normally.

Moving to a Linux platform for execution only requires us to copy two Python Files.

1. pfish.py
2. _pfish.py

Execution under Linux (Ubuntu version 12.04 LTS in this example) works without changing any Python code and produces the following results shown in [Figures 3.11–3.13](#).

You notice that the pFishLog file under Linux has a number of warnings; this is due to lack of read access to many of the files within the /etc directory at the user privilege level I was running and due to some of the files being locked because they were in use.

CHAPTER REVIEW

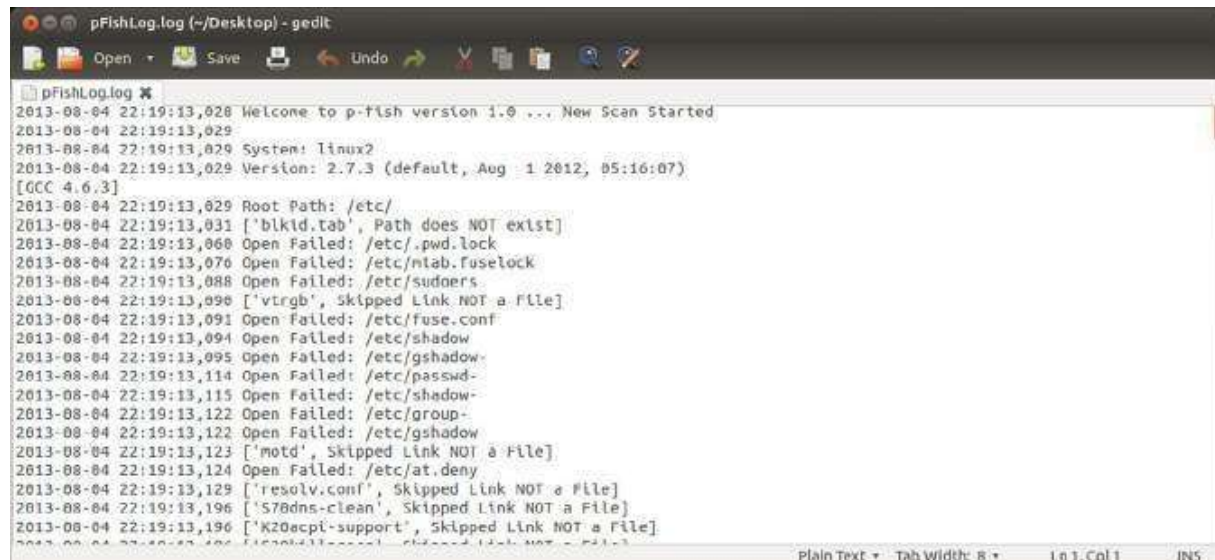
In this chapter I created our first useable Python forensics application. The pfish.py program executes on both Windows and Linux platforms and through some ingenuity I only used Python Standard Library modules to accomplish this along with our own code. I also scratched the surface with argparse allowing us to not only parse the command line but also validate command line parameters before they were used by the application.

I also enabled the Python logger and reported events and errors to the logging system to provide a forensic record of our actions. I provided the user with the capability of selecting among the most popular one-way hashing algorithms and the program extracted key attributes of each file that was processed. I also leveraged the cvs module to create a nicely formatted output file that can be opened and processed by standard applications on both Windows and Linux systems. Finally, I implemented our first class in Python with many more to come.

FIGURE 3.12

Linux Execution Results pfish Result File.

Additional Resources 89



```
pFishLog.log (~/Desktop) - gedit
2013-08-04 22:19:13,028 Welcome to p-fish version 1.0 ... New Scan Started
2013-08-04 22:19:13,029
2013-08-04 22:19:13,029 System: linux2
2013-08-04 22:19:13,029 Version: 2.7.3 (default, Aug  1 2012, 05:16:07)
[GCC 4.6.3]
2013-08-04 22:19:13,029 Root Path: /etc/
2013-08-04 22:19:13,031 ['blkid.tab', Path does NOT exist]
2013-08-04 22:19:13,060 Open Failed: /etc/.pwd.lock
2013-08-04 22:19:13,070 Open Failed: /etc/ntab.fuse.lock
2013-08-04 22:19:13,088 Open Failed: /etc/sudoers
2013-08-04 22:19:13,090 ['vtrgb', Skipped Link NOT a File]
2013-08-04 22:19:13,091 Open Failed: /etc/fuse.conf
2013-08-04 22:19:13,094 Open Failed: /etc/shadow
2013-08-04 22:19:13,095 Open Failed: /etc/gshadow
2013-08-04 22:19:13,114 Open Failed: /etc/passwd
2013-08-04 22:19:13,115 Open Failed: /etc/shadow
2013-08-04 22:19:13,122 Open Failed: /etc/group
2013-08-04 22:19:13,122 Open Failed: /etc/gshadow
2013-08-04 22:19:13,123 ['motd', Skipped Link NOT a File]
2013-08-04 22:19:13,124 Open Failed: /etc/at.deny
2013-08-04 22:19:13,129 ['resolv.conf', Skipped Link NOT a File]
2013-08-04 22:19:13,196 ['S78dns-clean', Skipped Link NOT a File]
2013-08-04 22:19:13,196 ['K20acpi-support', Skipped Link NOT a File]
```

FIGURE 3.13

Linux Execution Results pFishLog File.

SUMMARY QUESTIONS

1. If you wanted to add additional one-way hashing algorithms, which functions would you need to modify? Also, by using just the Python Standard Library what other one-way hashing algorithms are readily available.
2. If you wanted to eliminate the need of the two global variables how could you easily accomplish this by using classes? What function would you convert to a class and what methods would you need to create?
3. What other events or elements do you think should be logged? How would you go about doing that?
4. What additional columns would you like to see in the report and how would you obtain the additional information?
5. What additional information (such as Investigator name or case number) should be included in the log. How would you obtain that information?

LOOKING AHEAD

In [Chapter 4](#), I will be continuing with the cookbook section by tackling searching and indexing of forensic data.

Additional Resources

Hosmer C. Using SmartCards and digital signatures to preserve electronic evidence. In: SPIE proceedings, vol. 3576. Forensic Technologies for Crime Scene and the Laboratory I. The paper was initially presented at the investigation and forensic science technologies symposium; 1998. Boston, MA, <http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid%974141> [01.11.1998].

Kim G. The design and implementation of tripwire: a file system integrity checker. Purdue ePubs computer science technical reports, 1993. <http://docs.lib.purdue.edu/cstech/1084/>. CHAPTER

Forensic Searching and Indexing Using Python 4

CHAPTER CONTENTS

Introduction	
91	
Keyword Context Search	93
How can This be Accomplished Easily in Python?	94
Fundamental Requirements	96
Design Considerations	96
Main Function	98
ParseCommandLine	98
SearchWords Function	98
PrintBuffer Functions	99
Logger	99
Writing the Code	99
Code Walk-Through	
100	
Examining Main—Code Walk-Through	100
Examining _p-Search Functions—Code Walk-Through	101
Examining ParseCommandLine	101
Examining ValidateFileRead(theFile)	103
Examining the SearchWords Function	103
Examining the PrintBuffer Function	106
Results Presentation	

107	
Indexing	
109	
Coding isWordProbable	
112	
p-Search Complete Code Listings	
.....	114
p-Search.py	114
_p-Search.py	116
Chapter Review	
122	
Summary Questions	
123	
Additional Resources	
123	

INTRODUCTION

Searching is certainly one of the pillars of forensic investigation and today a search is only as good as the investigator running it. Knowing what to search for, where to search for it (in a file, in deleted files, slack space, e-mail files, a database, or application data), and then interpreting the search results requires both experience and knowledge of criminal behavior. Over the past several years we have seen the emergence of indexing of evidence and we have also seen frustration with the performance of indexing methods. The Python programming language has several built-in language mechanisms along with Standard Library modules that will assist in both searching and indexing.

Python Forensics 91 © 2014 Elsevier Inc. All rights reserved.

There are many reasons for searching and indexing, and I want to make sure we examine more than just simple keywords, names, hashes, etc. So what is the underlying reason or rationale why we search? Referencing fictional works or characters is always a risk when writing a book like this, but in this case I believe it works. Most of us have read some if not all of the works of Sir Arthur Conan Doyle's Sherlock Holmes novels and short stories. In these fictional accounts, Holmes uses both deductive and inductive reasoning to create theories about the crime and profiles of the suspects, victims, and even bystanders.

Deductive reasoning takes no risk, in other words if you have verifiable facts along with a valid argument the conclusion must be true. For example:

1. All men are mortal
2. Socrates was a man

Therefore, Socrates was mortal.

Inductive reasoning on the other hand takes risks and uses probabilities. For example:

1. John always leaves for the office at 6:00 AM
2. He punches in between 7:30 and 7:45 AM each day
3. John left home today promptly at 6:00 AM verified by a surveillance camera and punched in at work at 7:42 AM.

Therefore, John could not have committed a murder at 7:05 AM that occurred over 3 miles from the office.

The basis of these hypotheses whether based on deductive or inductive reasoning is facts. Therefore, I believe the reason to search and index is to discover facts that we then use to create theories and hypothesis. I know this may seem brutally obvious, but then again we sometimes narrow in on a theory before we have any trustworthy facts. This is especially true in the digital world where we can be quick to judge as the saying goes “because I saw it on the Internet” when in fact the Internet or digital data have been manipulated to skew the facts.

Seasoned investigators understand this and are typically very thorough in their investigation of digital crime scenes. I have had the privilege of working with many such investigators and they are tenacious in validation of digital evidence. I cannot tell you how many times I have been asked “are you sure about that?” or “can you prove that” or even “how else could this have happened?” Finally, we need to be conscious in our searching to discover both inculpatory and exculpatory evidence and make sure both types of facts are recorded before we create theories or hypothesis.

Inculpatory evidence supports a defined hypothesis

Exculpatory evidence is contradictory to a defined hypothesis

The question then becomes what facts do we wish to discover? Fundamentally, the answer is who, what, where, when, and how or to be slightly more specific, here are small set of examples.

You might be wondering about the “Why” of the search. We tend not to think about why as in many cases this is generated by the investigator, through experience, intuition, and deep understanding regarding the circumstances that surround the case.

- What documents exist and what is their content or relevance? When were they created, modified, last accessed, when and how many times were they printed? Were the documents stored or written to a flash device or stored in the cloud? Where did they originate?
- What multimedia files exist and where did they originate? For example, were they downloaded from the Internet or recorded or snapped by the suspect or victim. If so, what camera or recording device was used?
- Who is the suspect(s) we are investigating and what do we know about them? Do we have a photograph, phone numbers, e-mail addresses, home address, where do they work, when and where have they traveled, do they own a vehicle(s), if so what make, model, year, and plate number? Do they have an alias (physical world or cyber world)?
- Who are their known associates, what do we know about them?

I think you get the idea, the questions that we might ask during a search are vast with some being very specific and others more general. In this chapter, we certainly cannot solve all of these issues, but we can scratch the surface and create Python programs that will assist us now and be able to be evolved in the future.

KEYWORD CONTEXT SEARCH

Keyword searching with context is the ability to perform a search of a data object like a disk image or a memory snapshot, and discover keywords or phrases that relate to a specific context or category. For example, I might want to perform a search of a disk image for each occurrence of the following list of street names for the drug cocaine:

blow, C, candy, coke, do a line, freeze, girl, happy dust, mama coca, mojo, monster, nose, pimp, shot, smoking gun, snow, sugar, sweet stuff, white powder, base, beat, blast, casper, chalk, devil drug, gravel, hardball, hell, kryptonite, love, moonrocks, rock, scrabble, stones, and tornado.

How can this be accomplished easily in Python?

In order to tackle keyword searching with Python we must first address several simple issues. First, how should we store the search words or phrases? Python has several built-in data types that are ideally suited to handle such lists each with their own unique capabilities and rules. The basic types are sets, lists, and dictionaries. My preference would be to create a set to hold the search words and phrases. Sets in

Python provide a nice way of dealing with this type of data mainly due to the fact that they automatically eliminate duplicates, they are trivial to implement, and they can easily be tested for members.

The code below demonstrates the basics. I start by initializing an empty set named searchWords. Next, the code opens and then reads the file narc.txt that contains narcotic-related words and phrases. I of course include exception handling for the file operations to ensure we successfully opened and then can read each line from the file. Next, I print out the list of search words and phrases that were loaded line by line (notice one word or phrase per line) and then search the list for the word “kryptonite” and either print “Found word” if the word exists in the search words list or “not found” if the word does not exist. One fine point, I specify line.strip() when adding the words to the set searchWords. This strips the newline characters that exist at the end of each line.

```
import sys
searchWords = set()
try:

    fileWords = open('narc.txt')
    for line in fileWords:
        searchWords.add(line.strip())

except:
    print("file handling error")
    sys.exit()

print(searchWords)
if ('kryptonite' in searchWords):
    print("Found word")
else:
    print("not found")
```

Executing the code produces the following output:

Starting with the contents of the set:

```
{ 'shot','devil drug' 'do a line' 'scrabble' 'casper' 'hell' 'kryptonite', 'mojo' 'blow' 'stones  
tornado' 'white powder' 'smoking gun' 'happy dust' 'gravel' 'hardball'  
'moonrocks' 'monster' 'beat' 'snow sugar' 'coke' 'rock' 'base' 'blast' 'pimp' 'sweet  
stuff' 'candy' 'chalk' 'nose' 'mama coca' 'freeze girl' }
```

Then printing out the results of the search for kryptonite produces Found word
This simple code then solves two fundamental challenges:

(1) Reading a list into a set from a file

(2) Searching a set to determine if an entry exists in a set. Next, we have to solve the problem of extracting words and phrases from the target of a search. Almost always this requires that a binary file or stream be parsed. These binary data could be a memory snapshot, a disk image or a snapshot of network data. The key is the data would be either unformatted or formatted based on one of hundreds of standards, some open and some proprietary with text and binary data intertwined. Therefore, if we would like to develop a simple search tool that could extract and compare text strings without regard to the format or content, how might we accomplish this?

The first challenge would be to define what Python data object we would use to hold these binary data. For this I have chosen the Python object bytearray as this fits our criteria exactly. As the name implies bytearrays are simply a series of bytes (a byte being an unsigned 8-bit value) starting at the zero array element and extending to the end of the array. I am going to take advantage of a couple of Python specialties, for example, loading a file into a bytearray, the core code to do so is only two lines.

The first line opens a file for reading in binary mode. Whereas the second line reads the contents of the file into a bytearray named baTarget. You notice that I prefix the object Target with “ba” this simply makes the code more readable and anyone will immediately recognize that the object is of type bytearray. Also, note this code is just an illustration. In the real program, we have to check the return values from the function calls and make sure there are no errors, I will illustrate that technique when we get to the actual program.

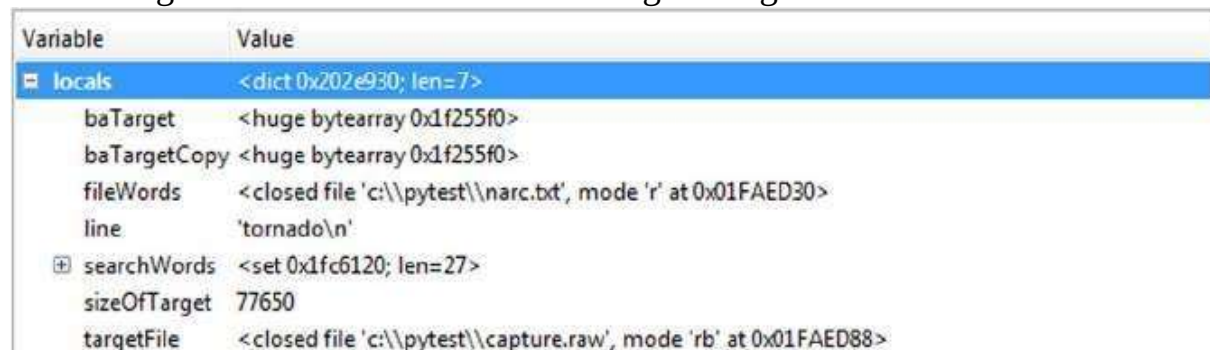
```
targetFile = open('file.bin','rb')
```

```
baTarget¼ bytearray(targetFile.read())
```

One of the immediate questions you might have is, “how do I know how many bytes were loaded into the bytearray?” This is handled just like any other Python object by using the len function.

```
sizeOfTarget¼len(baTarget)
```

In [Figure 4.1](#), we can see the object baTarget that was created and also the sizeOfTarget that was determined of baTarget using the len function.



Variable	Value
locals	<dict 0x202e930; len=7>
baTarget	<huge bytearray 0x1f255f0>
baTargetCopy	<huge bytearray 0x1f255f0>
fileWords	<closed file 'c:\\pytest\\narc.txt', mode 'r' at 0x01FAED30>
line	'tornado\n'
searchWords	<set 0x1fc6120; len=27>
sizeOfTarget	77650
targetFile	<closed file 'c:\\pytest\\capture.raw', mode 'rb' at 0x01FAED88>

FIGURE 4.1

Snapshot of stackdata displaying the baTarget object and the size in bytes of the baTarget bytearray.

Fundamental requirements

Now that I have defined the new language and data elements we intend to use for our simple text search, I want to define the basic requirements of the program as shown in [Table 4.1](#).

Design considerations

Now that I have defined the basic requirements for the application, I need to factor in the design considerations. For the search program, I will be using Python Standard Library modules and the specialized code we developed for parsing the target file. [Table 4.2](#) covers the basic mapping of requirements to modules and functions that will be utilized.

Next, I want to define the overall design of the search capability. As with the p-fish program we developed in [Chapter 3](#), I will be using the command line and command line arguments to specify the inputs to the program. For the search results, I will be directing the output to standard output using some specialized methods developed to render the data for easy interpretation. Finally, I will be

using the built-in logging functions to complete the forensic nature of the search (Figure 4.2).

Turning to the internal structure, I have broken the program down into four main components. The Main program, ParseCommandLine function, the actual SearchWords function, PrintBuffer functions for outputting the results, and logger (note logger is actually the Python logger module), that is utilized by the major functions of p-search. I briefly describe the operation of each below and during the code

Table 4.1 Basic Requirements Requirement Requirement number name
SRCH-001 Command Line Arguments

SRCH-002 Log

SRCH-003 Performance SRCH-004 Output

SRCH-005 Error Handling Short description

Allow the user to specify a file containing keywords (ASCII in this example)

Allow the user to specify a file containing a binary file to search

Allow the user to specify verbose output regarding the progress of search

The program must produce a forensic audit log The program should perform the search efficiently The program should identify any of the keywords found in the binary file and provide a Hex/ASCII printout, the offset where the keyword was found along with the surrounding values in order to provide context

The application must support error handling and logging of all operations performed. This will include a textual description and a timestamp

Table 4.2 Standard Library Mapping
Requirement Design considerations

User Input (001) User input for keyword and target file

Performance (003) Selecting the right language and library objects to process the data is important when designing a search method

Output (004) The search results must be provided in a meaningful manner. In other words, mapping the identified keywords to the offset in the file along with pre- and postdata that surround the identified keywords

Logging and error Errors may occur during the handling (002 and processing of the keyword 005) and target file. Therefore,

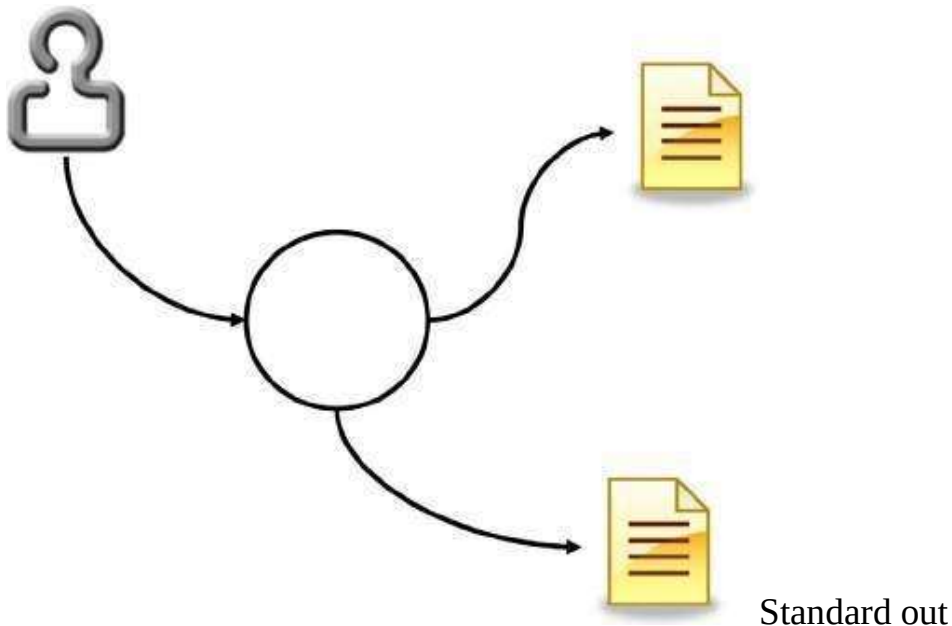
handling these potential errors must be rigorous Library selection

I will be using argparse from the Standard Library module to obtain input from the user

I will be using Python sets and bytearrays to handle the data related to searching which will increase performance and make the search process extensible I will be using the Standard Library print function to accomplish this, but instead of writing the result to a file directly, I am going to write the data to standard out, the user can then pipe the output if they wish to a file or into other programs or functions

The Python Standard Library includes a logging facility which I can leverage to report any events or errors that occur during processing

User



Program Arguments
p-search

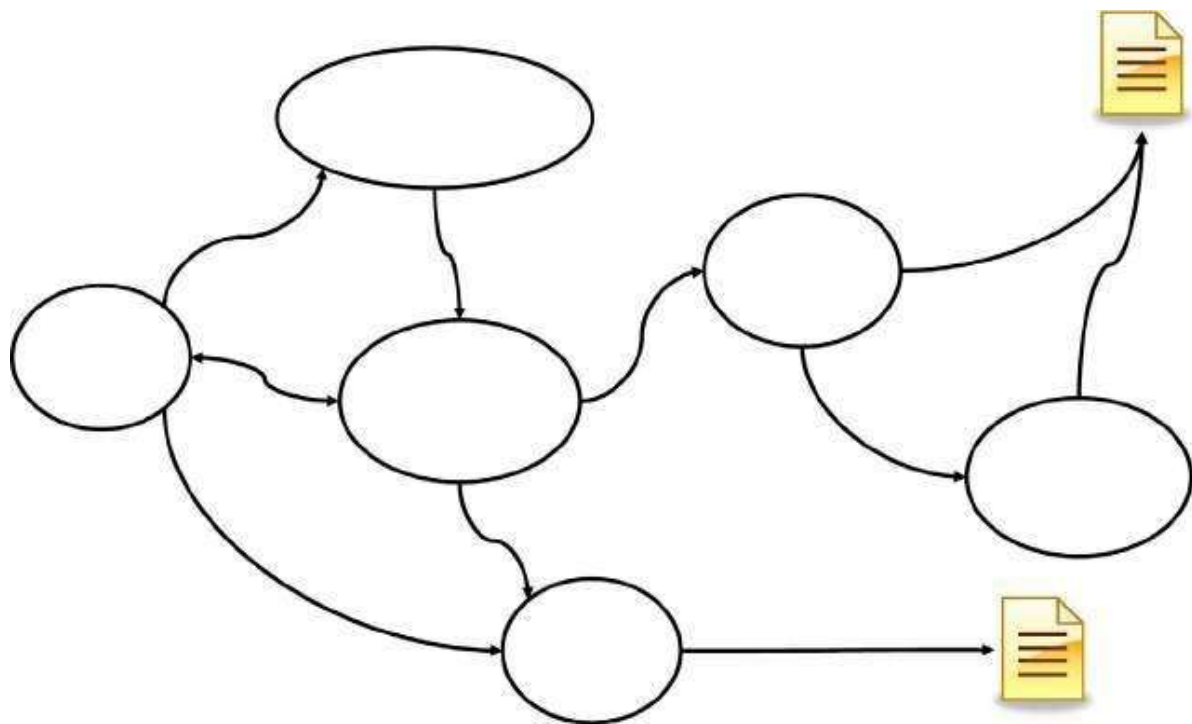
Event
and
error log

p-search context diagram.

p-search internal structure Standard out

Program Arguments Keywords

Main ASCII

SearchWords^{foundWords}

Offset buffer

Events

FIGURE 4.3

p-search internal structure.

PrintHeading

Event
and
error log

walk-through section a more detailed line by line explanation of how each function operates is provided ([Figure 4.3](#)).

Main function

The purpose of the Main function is to control the overall flow of this program. For example, within Main, I setup the Python logger, I display startup and completion messages, and keep track of the time. In addition, Main invokes the command line parser and then launches the SearchWords function. Once SearchWords completes, Main will log the completion and display termination messages to the user and the log.

ParseCommandLine

In order to provide smooth operation of p-search, I leverage the ParseCommandLine to not only parse but also validate the user input. Once completed, information that is needed by the core search method will be available via the parser.

SearchWords function

SearchWords is the core of the p-search program. The objective is to make this function as quick and accurate as possible. In this example, we are only searching for ASCII text strings or words. This provides a huge advantage and we can set the search up in a way that is quick and effective. The algorithm performs the search in two passes over the baTarget bytearray. The first pass converts any byte that is not an alpha character to a zero. The second pass collects sequences of alpha characters. If the consecutive alpha character sequence is greater than or equal to MIN_WORD and less than or equal to MAX_WORD before encountering a zero, the characters are collected and then compared to the set of keywords. Note MIN_WORD and MAX_WORD will be defined as constants. In the future, they could be command line arguments to the search.

PrintBuffer functions

The PrintBuffer function is called by SearchWords whenever a keyword match is discovered. Then the offset within the file and Hex/ASCII display are sent to standard out.

logger

The built-in Standard Library logger provides us with the ability to write messages to a log file associated with p-search. The program can write information messages, warning messages, and error messages. Since this is intended to be a forensic application, logging operations of the program are vital. You can expand the program to log additional events in the code; they can be added to any of the `_p-search` functions.

Writing the code

Once again, I broke the code into source files `p-search.py` and `_p-search.py`, where `_p-search.py` provides the support and heavy lifting functions of p-search. In [Figure 4.4](#) you can see the WingIDE environment along with the p-search program running. The upper left corner displays the local and global variables. In the upper

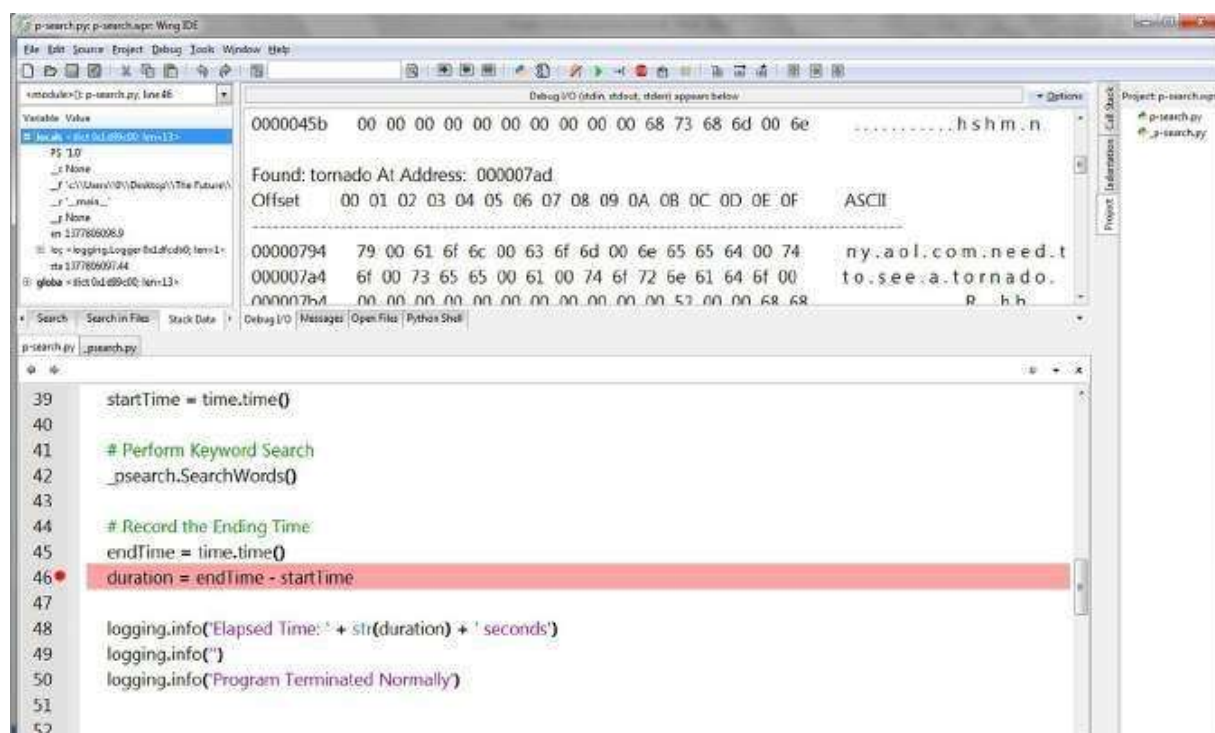


FIGURE 4.4
WingIDE p-search execution.

center panel you get a glimpse of the program output showing the match of the word “tornado” along with the detailed Hex/ASCII representation of the output. On the far right, the files associated with the program are listed. Finally, in the bottom portion of the screen you can see the source code for the program. In the

following section, we will walk-through the important code elements to disclose the detailed method and approach I have taken.

CODE WALK-THROUGH

I will be inserting dialog as I discuss each code section. The code walk-through will give you an in-depth look at all the codes associated with the program. I will be first walking through each of the key functions and then will provide you with a complete listing of both p-search.py and _p-search.py.

Examining Main—code walk-through

```
import logging
import time
import _psearch
```

For the main program we need to import logging to record forensic events and time to calculate the duration of program execution both from the Python Standard Library. We also import our _psearch module that contains the supporting and core functions of p-search

```
if __name__ == '__main__':
    P-SEARCH_VERSION = '1.0'
    # Turn on Logging
    logging.basicConfig(filename='pSearchLog.log', level=logging.DEBUG,
                        format='%(asctime)s %(message)s')
    Next I invoke command line parser to obtain the program arguments passed in
    by the user.
    # Process the Command Line Arguments
    _psearch.ParseCommandLine()
```

I then setup logging to occur and send the startup events to the logger. log

```
logging.getLogger('main._psearch')
log.info("p-search started")
```

I record the start time of execution in order calculate duration when the search is complete.

```
# Record the Starting Time
startTime = time.time()
```

Next, I invoke the SearchWords function. Note that this functions is contained in the _psearch module thus I need to prefix the call with the module name _psearch.

```
# Perform Keyword Search
```

```
_psearch.SearchWords()
```

I record the end time and calculate the duration and make the final entries in the log.

```
# Record the Ending Time
```

```
endTime¼ time.time()
```

```
duration¼ endTime - startTime
```

```
logging.info('Elapsed Time:'+ str(duration) +'seconds') logging.info("")
```

```
logging.info('Program Terminated Normally')
```

Examining _p-search functions—code walk-through

The remaining functions are contained in the _p-search.py file. The file starts by importing the needed modules. These include `argparse` for processing command line arguments. The `os` module is for handling file I/O operations. Finally, `logging` is used for forensic logging capabilities.

```
import argparse
```

```
import os
```

```
import logging
```

Examining ParseCommandLine

For the p-search program, we only require two parameters from the user—the full path name of the file containing the keywords to search along with the full path name of the file we wish to search. The verbose option is optional and when it is not provided the program messages will be suppressed ([Table 4.3](#)).

As you can see the ParseCommandLine prepares for three arguments verbose, keywords and srchTarget. Keywords and search target are required parameters and they are both validated by the function ValidateFileName().

```
def ParseCommandLine():
```

```
parser¼ argparse.ArgumentParser('Python Search')
```

```
parser.add_argument('-v','--verbose', help¼"enables printing of program messages", action¼'store_true')
```

```

parser.add_argument('-k', '--keyWords', type¼ ValidateFileRead, required¼True,
help¼"specify the file containing search words)"
parser.add_argument('-t', '--
srchTarget', type¼ ValidateFileRead, required¼True, help¼"specify the target
file to search)"

```

Table 4.3 p-search command line argument definition

Option	Description	Notes
-v	Verbose, if this option is specified then any calls to the DisplayMessage() function will be displayed to the standard output device, otherwise the program will run silently	
-k	Keyword, this allows the user to specify the path of the file containing the keywords	
-t	Target, this allows the user to specify the path of the file to be searched The file must exist and must be readable or else the program will abort	

-v Verbose, if this option is specified then any calls to the DisplayMessage() function will be displayed to the standard output device, otherwise the program will run silently

-k Keyword, this allows the user to specify the path of the file containing the keywords

-t Target, this allows the user to specify the path of the file to be searched The file must exist and must be readable or else the program will abort

The file must exist and must be readable or else the program will abort

Next we establish a global variable to hold results of parse_args operations. Any invalid entries will automatically abort processing.

```

global gl_args
gl_args ¼ parser.parse_args()
DisplayMessage("Command line processed: Successfully)"
return

```

One of the nice features of argparse is the ability to include a help message with each entry. Then whenever the user specifies -h on the command line argparse automatically assembles the appropriate response. This allows the developer to provide as much information as possible to assist the user in providing the correct response, so be verbose. [Figure 4.5](#) depicts the operation of the -h option for p-search.