Lesson 1, Getting Started

Preface

What you will learn

This is the first in a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

There is something for just about everyone here. Beginners start at the beginning, and experienced programmers jump in further on. You simply need to enter the series of lessons at the point that best fits your prior programming knowledge.

Beginners

If you don't know anything about programming, these lessons will take you from ground zero to the outer-space habitat of the modern object-oriented computer programmer.

Programmers

If you already know how to program, these lessons will teach you how to program using the Python scripting language.

Python programmers

If you already know how to program using Python, these lessons will teach you how to use the Python scripting language to take advantage of the ever expanding Java programming environment, without the requirement to learn how to program in Java.

Java programmers

If you already know how to program using Java, these lessons will teach you how to use Python for rapid prototyping of Java programs.

Overall

When you complete this series of tutorial lessons, you will have completed the equivalent of three or four semesters of computer programming studies at the community college level.

Prerequisites

There are only two prerequisites:

- Knowledge of how to use a computer on the web.

- A strong desire to learn how to write computer programs.

How to use a computer on the web

The fact that you are reading this page probably means that you already know how to use a computer on the web. I'm not talking about rocket science here. I am talking about knowing how to download and install software from specified web sites, and how to execute that software offline.

Windows would be handy

Although not absolutely necessary, it would also be very handy for you to have access to, and know how to use Microsoft Windows. The programming skills that I am going to teach you are applicable to a broad range of operating systems. However, I will use Microsoft Windows as my teaching platform, so you will need to be able to follow instructions explained in Windows jargon.

Strong desire to learn

This should be self-explanatory, but just in case it isn't, I will make a couple of comments in this regard.

Don't expect a miracle to happen

Simply reading these lessons won't cause you to experience a miraculous transformation that will cause recruiters to start waving money at you.

Practice, practice, and more practice

I am going to provide you with a lot of information, but you will also need to put your brain in gear and write a lot of programs to make the information stick in your brain. Very few people become expert musicians without a lot of practice. Similarly, very few people become expert programmers without a lot of practice.

So, if you want to learn to program, plan to spend a lot of time in front of your computer, not just reading, but programming as well.

Why Use Python?

With so many programming environments available, why did I select Python?

First of all, it's free. I like that. You won't need to invest an arm and a leg just to get started. All you will need to do is go to http://www.python.org/ and download the software. Yes, Virginia, there really is a Santa Claus, and he lives on the Web.

What is Python?

You might also want to visit http://www.python.org/doc/Summary.html, which is where I extracted the following:
"Python is an interpreted, interactive, object-oriented programming language...

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

The Python implementation is portable: it runs on many brands of UNIX, on Windows, DOS, OS/2, Mac, Amiga...

Python is copyrighted but freely usable and distributable, even for commercial use."

What is JPython?

Another very important reason that I chose Python is the availability of JPython.; JPython will become very important later in the course once we become involved in the use of the Java class libraries.

The following was extracted from http://www.jpython.org/docs/whatis.html. "JPython is an implementation of the high-level, dynamic, object-oriented language Python seamlessly integrated with the Java platform and certified as 100% Pure Java.

JPython is freely available for both commercial and non-commercial use and is distributed with source code.

JPython is complementary to Java and is especially suited for the following tasks: ..."

A pathway into Java

In other words, Python and JPython provide a relatively painless pathway into the exciting, and often complex world of Java.

To learn more about Java, visit my online Java tutorials at this URL.

More facts, less hype

So, beyond the hype, why did I really choose Python? Because it is an interpreted, interactive, object-oriented programming language that is relatively easy to learn.

Learn OOP without the details

With Python, you can learn object-oriented programming without the need to understand many of the complex details.

A beginner-friendly language

Python is a beginner-friendly language that takes care of many of the complex details for you behind the scenes. This makes it possible for you to concentrate on

the big picture of what your program is intended to accomplish without getting bogged down in detail.

Sneaking up on OOP

With Python, you can "sneak up" on object-oriented programming concepts. You don't have to deal with the object-oriented nature of the language on the first day, or for a long time, for that matter. (With Java, you can't write even the simplest program without encountering a need to understand object-oriented programming concepts.)

Python has an interactive mode

The interactive mode makes it easy to try new things without the burden of the *edit-compile-execute* cycle of other programming languages such as Java. Interactive mode is a very friendly environment for persons in the learning stages of a language.

Python has a non-interactive mode

Once you know that something works properly, it is very easy *to "put it in a can"* so to speak, and then execute it as a script. This lets you execute it without retyping it every time.

Python combines interactive and non-interactive modes

This combination gives you the ability to use previously written script files while working in the interactive mode. Using this approach, you are able to minimize your interactive typing effort while still writing and testing large Python programs in interactive mode.

Small core, large library

Like Java, Python has a small compact core and a large, extensible library. Thus, much of what you will need to do has already been written and tested for you. Your task will be to write the code to glue those library components together, and to write new capabilities on an as-needed basis.

Let's Write a Program

Download and install the software

The first step is to go to http://www.python.org/ with your web browser. Download, and install the Python software using the download link that you will find there.

I'm going to assume that either you already know how to do this, or you can get help from a friend. In other words, I'm not going to try to explain how to download and install the Python software.

Starting the programming environment

As I mentioned earlier, all of my instructions will be based on the use of Python running on my Windows NT operating system. I believe that those instructions will also be directly applicable to Windows 95 and Windows 98.

However, as mentioned earlier, you have many choices for using Python. If you are running on some other platform, you will need to translate my instructions from Windows jargon into the jargon of your platform.

Enough talk; let's begin

Find the *Start* button on the task bar on your Windows desktop. Select

Start/Programs/Python x.y

where x.y is the version of Python that you have downloaded and installed. At the time of this writing, I have version 1.5.2 installed on my computer, and the selection shows on the menu as **Python 1.5**.

What do you see?

When you make this selection, you should see a menu having at least the following four options

- IDLE (Python GUI)
- Python (command line)
- Python Manuals
- Uninstall Python

The python manuals

Don't be bashful about reading the manuals. There is a wealth of information contained there, including a short, to-the-point tutorial written by the author of Python, Guido van Rossum.

The library reference

A good way to get a feel for the breadth and power of Python is to select **Python Manuals** from the menu and then select the **Library Reference** link. (The Library Reference is also available online as of the time of this writing.)

Most of what you see there probably won't mean much to you at this point in time, but hopefully will be familiar territory after you complete this course.

IDLE (Python GUI)

When I select the first item on the menu on my Windows NT machine, I get an error message about a missing DLL file. But when I click OK, the message goes away and everything seems to work anyway.

This selection brings up a window entitled *"Python Shell."* This is one of the interactive programming environments available with Python on Microsoft Windows.

What does it look like?

When you bring up the Python Shell, you should see something like Figure 1 at the top of the window. (Note that I inserted some line breaks to force the text to fit in this narrow page format.)

What is a GUI?

GUI is an acronym for Graphical User Interface. This window is a GUI. It can be used for interactive Python programming.

To make it go away when you are finished, simply click the button in the upper-right corner that is labeled with an X.

The Python (command line) selection

When I select the second item in the menu mentioned above, I get essentially the same thing as the GUI, but in a "black screen" window commonly referred to as a DOS box, a console, a command-line window, or whatever you choose to call it.

The command-line window

This window can also be used for interactive Python programming in much the same way that the Python Shell can be used. Each has some practical advantages and disadvantages.

To make this screen go away, hold down the Ctrl key and press the Z key (Ctrl-z). Then press the Enter key. (You can also click the X in the upper right-hand corner.)

Your first Python program

You can use either of these windows to write and execute your first Python program.

Hello World in Python

As has become the custom in programming circles, we will make our first Python program one that displays "Hello World" on the computer screen. We will write and execute it interactively.

The Python prompt >>>

The three right-angle brackets that you see in both of the interactive screens (>>>) make up the Python interactive prompt. When the cursor is blinking to the right of that prompt, you can enter a Python programming statement interactively.

Let's do it

Type the following text to the right of the Python prompt and press the Enter key:
print "Hello World

If all goes well, your interactive Python screen should then look something like Figure 2.

Pay particular attention to the line that reads **Hello World** following your entry. That is the output from your program.

Congratulations

You have just written (and executed) your first Python program, and possibly your first computer program as well. Not only that, you only had to type one line of code to write and execute your program.

Note that your entire program, the output from your program, and a new prompt are all shown in Figure 2.

Python is ready for more

Python has provided a new prompt so that you can expand your program, or write another one.

The Java version of Hello World

In contrast, the simplest program that I know how to write in Java is shown in Figure 3.

This Java program also displays **"Hello World"** on the screen, but the program output is not shown in the box. This box shows only the program code.

The edit, compile, execute cycle

This Java program code must first be captured in a Java source file. Then it must be compiled. After it is compiled, it can be executed to produce "Hello World" on the computer screen.

More complex than Python

While not very complex by programming standards, this Java program is certainly more complex than the one-line Python program shown earlier. A great deal more programming knowledge is required to write, compile, and execute the Java program.

The following three steps were are required to write and execute this Java program:

- Use a text editor to create a file containing the program code.
- Compile the program using the Sun **javac** compiler program.
- Execute the program using the Sun java virtual machine program named **java**.

To my knowledge, Java programs cannot be written and executed interactively.

Where Do We Go From Here?

My plan is to publish a new lesson about once every other week. Over the course of time, these lessons will teach you how to program in Python, teach you how to program in JPython, and teach you how to use JPython to take advantage of many of the features of Java (without having to write Java code).

Review

1. These lessons are for beginners only, True or False?

Ans: False. These lessons will have something for just about everyone, depending on where the user enters the sequence of lessons. Beginners start at the beginning, experienced programmers jump in later on.

2. Python is useful only as a compiled language, True or False?

Ans: False. Python can be used in an interactive, interpreted, or compiled mode.

3. JPython is part of the Java programming language, True or False?

Ans: False. JPython is an implementation of Python integrated with the Java platform from Sun and certified as 100% pure Java, but was not written by Sun, and not belonging to Sun.

4. Both Python and JPython are free, True or False.

Ans: True. As of this writing, both are freely usable and distributable, both for commercial and non-commercial use.

5. Python is an object-oriented language, True or False.

Ans: True.

6. Python has a large core and a small library, True or False?

Ans: False. Just the opposite is true, small core and large library.

7. What does the Python interactive prompt look like?

Ans: The Python prompt is >>>

Let's Do Numbers

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

There is something for just about everyone here. Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

Because this course is designed for beginning programmers, I am going to take it slow and easy for the first few lessons. My discussion will be informal, and will be designed to get you familiar and comfortable with the Python interactive programming environment. Later on, I will get a little more formal.

A programmable calculator

I am going to start out by showing you how to use Python as a programmable calculator. In the process, I will introduce you to some programming concepts, such as *operators* that I will explain in a more formal way in subsequent lessons.

Let's Program

Start the interactive Python environment

The first thing that you need to do is to start the interactive programming environment. If you have forgotten how to do that, see Lesson 1.

There are two ways to start the interactive Python environment from the Windows *Start* menu. You can either:

- Select IDLE (Python GUI), or
- Select Python (command line).

Either way, the interactive programming environment should look something like Figure 1 when it starts running:

The **>>>** that you see on the last line is the Python interactive prompt, which I will refer to simply as the prompt.

Program comments

Before we go any further, I need to introduce you to the concept of *program comments*.

In programming jargon, a comment is text that you insert into the program that is intended for human consumption only. Comments provide a quick and easy form of documentation. They are ignored by the computer and are intended to explain what you are doing.

How useful are comments?

Comments aren't terribly useful when doing interactive programming. Presumably you already know what you and doing and don't need to explain it to yourself using comments.

However, comments are very useful when you are writing scripts that you will store in files and use again later after you have forgotten how you did what you did.

In these lessons, I will use comments occasionally to explain what I am doing for your benefit, even in interactive mode.

What is the comment syntax?

According to The Python Reference Manual "A comment starts with a hash character (#) ... and ends at the end of the physical line.

Figure 2 shows a Python comment along with a new kind of prompt.

What is that ... thing?

The interactive mode actually uses two different kinds of prompts.

- One is **>>>**
- The other is **...**

I will explain the difference later. For now, just pretend like they both mean the same thing. That will suffice until we get into more complicated material.

How much is 2+5

Enter **2+5** at the prompt and press the Enter key. You should see something like Figure 3 on your screen:

Input and output

To begin with, we need to differentiate between input and output. Everything that appears on a line with one of the prompts is input to the Python interpreter. (You typed it, so you know that it is input.)

Everything that appears on a line without a prompt is output from the interpreter. (You didn't type it. The interpreter produced it, so it was output.)

What was the input in this case?

Your input was the expression **2+5**.

What was the output?

Python evaluated that expression and produced an output, which was the sum of 2 and 5, or 7.

Then Python presented you with a new prompt to allow you to provide more input.

Mixing comments and expressions

Now try entering the comment, followed by the expression that you see in Figure 4, and note the difference in the prompts.

My only reason for showing you this at this time is to give you some experience in mixing comments and expressions.

Command line versus GUI

The text in Figure 4 was produced using the Python (command line) window. I don't get exactly the same behavior (with respect to prompts) when I use the Python GUI Shell under Windows NT. Rather, what I get is shown in Figure 5. (Note the missing **...** prompt.)

I believe that the command-line version is the more correct of the two.

Let's get technical

But not too technical. Computer programs are made up of statements.

Statements are composed of expressions.

(Later we will learn that large Python programs are made up of smaller programs called modules, which are made up of statements, which are composed of expressions.)

Statement ends at the end of the line

In Python, a statement normally end at the end of the line that contains it (although there are exceptions to this rule).

For the time being, suffice it to say that expressions are made up of

- literal values,
- variables,

- operators, and
- parentheses

(I will defer a discussion of variables until a subsequent lesson.)

What are operators?

If you have ever used a hand calculator, you already know what operators are. The plus sign is an operator in the expression shown in Figure 6.

In programming jargon, operators are said to operate on *operands*.

What are operands?

In Figure 6, the 2 is the left operand and the 5 is the right operand of the plus operator.

Unary and binary operators

Normally, operators are said to be either *unary* or *binary*.

A unary operator has only one operand while a binary operator has two operands.

Some can be either

Some operators, such as the minus sign, can be either unary or binary operators.

In its unary mode with a single operand, a minus sign is usually a *sign changing* operator, while in its binary mode with two operands, a minus sign is usually a subtraction operator.

All of the operators discussed in this lesson are being used as binary operators.

Some arithmetic operators

Python has numerous operators. You can find a complete list of operators in the Python Reference Manual.

For the time being, we will concentrate on the follow arithmetic operators:

- The addition operator, +
- The subtraction operator, -
- The multiplication operator, *
- The division operator, /
- The modulus operator, %

Figure 7 shows some examples of using these operators that probably won't present any surprises to you.

If you add 2 and 5, you get 7. If you subtract 5 from 2, the answer is 3. If you multiply 2 by 5, you get 10

Integer division

The next result, shown in Figure 8, may surprise you until I explain it. In this case, we are dividing the integer (whole number) 2 by the integer 5 producing an integer result.

Normally, a hand calculator would tell you that the answer is 0.4, but that is not an integer result. Rather, it is a decimal fraction.

As you can see in Figure 8, Python tells you that the result of dividing the integer 2 by the integer 5 is 0.

Remember long division?

Think back to when your second grade teacher taught you how to do long division with whole numbers. She told you that if you divide 2 by 5, you get a quotient of 0 and a remainder of 5. Or, if you divide 23 by 4, you get a quotient of 5 and a remainder of 3. That is what we are talking about here.

The modulus operator

And that brings us to the modulus operator (%).

Try entering the expressions shown in Figure 9. (You don't need to enter the comments. They are there to explain what is going on. But it wouldn't hurt for you to enter a few, just for practice.)

What does the modulus do?

The purpose of the modulus operator is to produce the remainder resulting from an integer division.

As you can see from this example in Figure 9, the division operator produced the integer quotient of 5, and the modulus operator produced the remainder of 3 (just like your second-grade teacher told you it should).

Decimal division

What if you don't want to produce an integer quotient and a remainder? What if you really want to produce a decimal quotient the way hand calculators do.

This is easy to do. Just represent either the numerator or the denominator or both as a decimal value and Python will automatically convert to decimal arithmetic mode as shown in Figure 10.

Note that in the first expression, I changed the numerator from 2 to 2.0 (I could have left off the zero, but adding the zero makes the decimal point easier to spot).

This caused Python to perform decimal arithmetic instead of integer arithmetic, producing the decimal quotient value of 0.4.

I did essentially the same thing to the denominator in the second expression, producing the same result.

Grouping terms with parentheses

What is the result of evaluating the expression shown in Figure 11?

Try it on your hand calculator. (You will probably need to use an X instead of an * to indicate multiplication.) My hand calculator gives an answer of 32.

Now try it with Python and you should get the result shown in Figure 12.

Oops!

This answer doesn't match the answer given by my hand calculator, and I'll bet that it doesn't match your calculator either unless you are using a fancy scientific calculator.

The answer depends on the order in which you perform the various arithmetic operations. Ordinary hand calculators usually do the arithmetic in the order that the terms are fed into the keyboard.

Precedence

However, most computer programming systems, including Python, use a precedence system to decide which operations to perform first, which operations to perform second, etc.

I'm not going to go into the Python precedence system in detail. (If you are interested in the order of precedence of all the operators, you can find a precedence table in the Python Reference Manual.)

Rather, I am going to show you how to group terms using parentheses so that you can control the order of operations without worrying about the precedence system.

A sample grouping

The Python code fragment in Figure 13 shows how I can produce both results simply by grouping terms using parentheses.

The first expression produces 32 as produced by the hand calculator. The second expression produces 23 as produced by the earlier Python expression.

A Python expression is evaluated by first evaluating each of the sub-expressions inside the parentheses and then using those values to complete the evaluation.

Forcing addition to be performed first

In the first expression, I forced Python to perform the addition first by placing the addition inside the parentheses. This produced an intermediate value of 8 when the sub-expression inside the parentheses was evaluated. The remaining part of the overall expression was then evaluated by multiplying the intermediate value by 4, producing a result of 32.

Forcing multiplication to be performed first

In the second expression, I forced Python to perform the multiplication first (which it always does first anyway, but the parentheses make that more obvious). This produced an intermediate value of 20. The remaining part of the overall expression was then evaluated by adding the intermediate value to 3 producing an output of 23.

Hopefully, you get the picture. By using parentheses to group the terms in an expression, you have total control over the order in which the arithmetic operations are performed, without having to memorize a precedence table.

Nested parentheses

Parentheses can be, and often are nested to provide greater control over the order of the operations as shown in Figure 14.

In this case, Python evaluates the expressions inside the innermost parentheses first, and then works from the inside out evaluating each pair of parentheses along the way.

Negative integer division

When I learned to do long division in the second grade, I didn't know about positive and negative numbers yet, so I didn't learn about remainders when one of the operands is negative and the other is positive. I suspect that you didn't either.

Figure 15 shows how negative integer division and modulus works in Python.

The remainder may be surprising

The quotient shouldn't be a surprise, but the remainder may be surprising when the numerator and denominator have different algebraic signs.

An exercise for the student...

I'm not going to try to explain it. I just want to make you aware that the behavior of integer division and integer modulus is different when the operands have different signs. I will leave it as an exercise for the student to think about this and come to a mental reconciliation with the facts as presented here.

Complex Numbers

Python also provides a significant level of support for doing arithmetic with complex numbers. Since this is a very specialized area, which is probably of interest to only a

small percentage of potential Python users, I'm not going to provide any of the details. If this is something that interests you, see an example at this URL.

Programming Errors

Sometimes, you may make an error and enter an expression that can't be evaluated. In this case, you will get an error message. A typical error message is shown in Figure 16.

Briefly, this error message means that the Python interpreter doesn't know how to add the value **3** to the value **5a**. I will discuss error messages in more detail in a subsequent lesson. I just wanted to show you a programming error here at the beginning.

Mono spaced font

You may have noticed that the font that I used in Figure 16 is different from the font that I have been using in previous examples.

In the previous examples, I have used a font that allows me to put more information on each line in order to accommodate a narrow page format. However, that font does not allocate the same amount of space to each character. Narrow characters consume less space than wider characters. (That explains why I am able to get more text on each line.)

Note the pointer

In this case, I needed a font that allocates the same amount of space for each character so that the little pointer (^) below the **a** will be in the correct spot. This pointer is Python's way of giving you a hint as to where the error occurred.

Review

1. Python programming comments are ignored by the computer, True or False?

Ans: True. Programming comments are used for program documentation and are intended for human consumption.

2. Just like in C, C++, and Java, a Python comment begins with two slash characters (//) and continues to the end of the line, True or False?

Ans: False. In Python, a comment starts with the hash character (#) and ends at the end of the line.

3. The only prompt used by the Python interactive system is the one consisting of three right-angle brackets (**>>>**), True or False?

Ans: False. The Python interactive system also uses a secondary prompt consisting of three periods (**...**).

4. The output produced by the Python interactive system appears on a line without either of the prompts mentioned above, True or False?

Ans: True.

5. If you enter an expression at the prompt and press the **Enter** key, the result of evaluating the expression will be displayed on the next line without a prompt, True or False?

Ans: True, unless the expression can't be evaluated, in which case an error message will appear.

6. Computer programs are composed of expressions, which are made up of statements, True or False?

Ans: False. Just the reverse is true. Programs are made up of statements, which are composed of expressions.

7. In the following expression, **2+5**, what is the common jargon for the plus sign, and what is the common jargon for the 2 and the 5?

Ans: The plus sign is commonly called the **operator**. The 2 and the 5 are commonly called **operands**. More specifically, the 2 is the left operand and the 5 is the right operand.

8. List the operators discussed in this lesson, and describe the purpose of each.

Ans:

- The addition operator, +
- The subtraction operator, -
- The multiplication operator, *
- The division operator, /
- The modulus operator, %

9. Integer division produces a decimal result, True or False?

Ans: False. Integer division produces an integer result.

10. Describe how to force Python division to produce a decimal result.

Ans: Represent either the numerator, the denominator, or both as a decimal value, appending a decimal point and a 0 if necessary to cause it to be represented as a decimal value (actually, the decimal point alone is sufficient, but the zero makes the decimal point easier to detect visually).

11. The modulus operator is used to produce the quotient in division, True or False?

Ans: False, the modulus operator is used to produce the remainder.

12. Describe the use of parentheses in expressions.

Ans: Parentheses can be used to group terms in an expression in order to provide control over the order in which the operations are performed.

13. Describe how Python evaluates an expression containing parentheses.

Ans: A Python expression is evaluated by first evaluating each of the sub-expressions in the parentheses, and then using those values to complete the evaluation. If the expression contains nested parentheses, the evaluation is performed by evaluating the innermost parentheses first and working outwards from there.

Variables and Identifiers

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

There is something for just about everyone here. Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

I am taking it slow and easy for the first few lessons. My informal discussion is designed to familiarize you with the Python interactive programming environment

This lesson provides an introduction to the use of variables, and the required syntax of the identifiers used to represent variables.

What Is A Variable

As the name implies, a variable is something whose value changes over time.

A pigeonhole in memory

As a practical matter, a variable is a pigeonhole in memory, which has a nickname, where you can store values.

You can later retrieve the values that you have stored there by referring to the pigeonhole by its nickname (*identifier*). You can also store a different value in the pigeonhole later if you desire.

Is Python strongly typed?

One of the main differences between Python and programming languages such as Java is the concept of *type*.

In *strongly-typed* languages like Java, variables not only have a name, they have a type. The type determines the kind of data that you can store in the pigeonhole.

It is probably more correct to say that the type determines the values that you can store there and the operations (addition, subtraction, etc.) that you can perform on those values.

Python is not strongly typed

One of the characteristics that makes Python easier to use than Java is the fact that, with Python, you don't have to be concerned about the type of a variable. Python takes care of type issues for you behind the scenes.

Declaration of variables

Another difference between Python and Java is that with Java, you must *declare* variables before you can use them. Declaration of variables is not required with Python.

With Python, if you need a variable, you simply come up with a name and start using it as a variable.

Dangerous curves ahead!

With this convenience comes some danger. You can only have one variable with the same name within the same scope (I will discuss scope in a subsequent lesson).

Don't use the same name for two variables

With Python, if you unintentionally use the same name for two variables, the first will be overwritten by the second. This can lead to program bugs that are difficult to find and fix.

A more subtle danger

A more subtle danger is that you create a variable that you intend to use more than once and you spell it incorrectly in one of those uses. This can be an extremely difficult problem to find and fix. I will illustrate what I mean by this later with a sample program.

Rules for Identifiers

The name for a variable must follow the naming rules for identifiers that you will find in the *Python Language Reference* at this URL.

Give me the rules in plain English

The notation used in the *Python Language Reference* to define the naming rules is a little complicated, so I will try to interpret it for you.

I believe that the Language Reference is saying that identifiers must begin with either a letter or an underscore character. Following that, you can use an unlimited sequence of letters, numbers, or underscore characters.

Case is significant

The letters can be uppercase or lowercase, and case is significant. In other words, the identifier **Ax** is not the same as the identifier **aX**.

Any old digit will do

Numbers can be any of the digit characters between and including 0 and 9.

Watch out for underscore characters

I recommend that you not use the underscore character unless you know exactly why you are using it. In some situations, the use of the underscore character has a special meaning. I will discuss some of those situations in subsequent lessons.

Let's Program

 **Start the interactive Python environment**

The first thing that you need to do is to start the interactive programming environment. If you have forgotten how to do that, see Learn to Program using Python: Lesson 1, Getting Started .

Create and use some variables

The interactive fragment shown in Figure 1:

```
>>> x=6     # create and populate x
>>> y=5     # create and populate y
>>> x+y     # add x to y and display the sum
11
>>>
```

Figure 1

- Creates two variables named **x** and **y**,
- Populates them by assigning values of 6 and 5 to them respectively
- Adds their values together to produce the sum value of 11.

Back to the pigeonholes

Using the informal jargon from an earlier paragraph, two pigeonholes are established in memory and are given nicknames of **x** and **y**.

The assignment operator

Integer values of **6** and **5** are stored in the two pigeonholes using the assignment operator (**=**).

The use of the assignment operator in this fashion causes the value of its right operand to be stored in the pigeonhole identified by its left operand.

What is an operand?

(If you don't recognize the use of the term operand, see an earlier lesson for an explanation.)

In this case, the right operands of the two variables are *literal numeric values*.

The left operands of the two variables are the nicknames identifying the two memory locations that constitute the variables named **x** and **y**.

Addition of variables

Then, in the third line of code, the values are retrieved from each pigeonhole and added together. The result of the addition (11) is displayed as output from the expression **x+y**.

Assigning the same value to several variables

Python allows you to assign the same value to several variables, causing them to come into existence (begin to occupy memory) at the same time if necessary.

Consider the interactive fragment shown in Figure 2.

```
>>> a=b=c=10    # assign 10 to several
variables
>>> a+b+c       # add them together
30
>>> a=b=c=20    # assign 20 to same
variables
>>> a+b+c       # add them together
60
>>>
```

Figure 2

Create three variables

The first statement creates three variables named **a**, **b**, and **c**, and assigns a value of 10 to each of them.

They are then added together, in the second line of code, to produce an output value of 30.

Assign different values

The boldface statement in the fourth line assigns the value of 20 to the same three variables, replacing what was previously there with the new value. Again they are added together, this time producing an output value of 60.

Type considerations

In most modern programming systems, values having fractional parts, such as 3.14159 are commonly referred to as *floating point* types. (This terminology comes from the fact that the decimal point can float back and forth from left to right.)

Similarly, whole number values are commonly referred to as *integer* types. (These are values with no decimal parts, such as, "I have **3** whole apples.")

Advantages and disadvantages

Each type has advantages and disadvantages relative to the other.

The range of values

For example, in some systems, the total range of values for an integer type is restricted to the set of whole numbers between -32768 and +32767. Anything outside that range cannot be handled as a whole number.

Although the range of an integer type will be different on different systems, it will almost always be less than the range of a floating point type on the same system.

Speed

However, on some systems integer arithmetic is performed much faster than floating point arithmetic. On those systems, if speed is important, using integers may be more attractive than using floating point types.

Floating point provides greater range

On most systems, the floating point type provides a much greater range in terms of the values that can be maintained and used for arithmetic. For example, a particular system might be capable of representing the following two values as well as millions of values in between:

0.000000000033333
333330000000000.0

Sometimes range is important, and sometimes not

Sometimes range is important, and sometimes it isn't. However, as I mentioned above, in many cases this greater range is obtained at some sacrifice in arithmetic speed relative to integer types.

Approximate results

Also, as I will explain in the *Review* section, floating point arithmetic often produces approximate results instead of exact results.

While approximate results might he OK for scientific calculations, they might not be OK for financial calculations.

Automatic type handling in Python

In *strongly-typed* languages such as Java, it is the responsibility of the programmer to make certain that types are handled correctly. For example, it is often not possible to store a floating point value into a variable previously declared to be for the storage of integer values. There is a very strong possibility that it simply won't fit.

Python takes care of the routine type issues for us automatically.

Consider the interactive code fragment shown in Figure 3.

```
>>> x=5
>>> y=6
>>> x+y
11
```

```
>>> x=5.55555
>>> y=6.66666
>>> x+y
12.22221
>>>
```

Figure 3

The variables **x** and **y** are originally created to store integers and are populated with the values 5 and 6 respectively. The variables are added and the correct sum is displayed as output from the interpreter.

Next assign some floating point values

Then the floating point values 5.55555 and 6.66666 are assigned to the same two variables named **x** and **y**. The two variables are successfully added and the correct result is displayed, demonstrating that the two floating point values were successfully stored in the variables originally created for integers.

How is this accomplished?

I don't know how this is accomplished. As Python programmers, we don't really care. We are simply happy that it works without the requirement for us to deal with the details of type.

One caution

As I mentioned in an earlier lesson, you might want to be very careful when doing division. If both operands are integers, you will not get the floating point result that you might be hoping for. The quotient will be truncated down to the next (algebraically) lower integer. To get a floating point result from a division, one of the operands must be a floating point value.

The magic continuation variable

In interactive mode, Python automatically provides a variable whose name is simply the underscore character (_).

This variable makes it easy to do continuation arithmetic in interactive mode. (This variable is intended for read only purposes, so don't assign a value to it explicitly.)

At any point in time, this variable will contain the most recent output value displayed by the interpreter.

How does it work?

Consider the interactive code fragment shown in Figure 4.

```
>>> 5+6
11
>>> _+22   # add 22 to the cont variable
33
>>>
```

Figure 4

This fragment starts out just like previous examples, causing the sum of 5 and 6 to be calculated and displayed.

Sum is saved in the continuation variable

As mentioned above, the sum value of 11 is automatically saved in the continuation variable whose name is simply the underscore.

The contents of the continuation variable (11) are then added to 22 producing a result of 33 (note the use of the underscore as a variable name in the boldface expression).

Tell me why again

The primary purpose of this automatic variable named _ is to make it easier for you to string calculations together in interactive mode and to display the intermediate results as you go.

Illegal variable names

The interactive fragment in Figure 5 shows the result of attempting to use an illegal variable name.

```
>>> 1x=6
  File "<stdin>", line 1
    1x=6
     ^
SyntaxError: invalid syntax
>>>
```

Figure 5

(As I indicated earlier, variable names cannot begin with a digit. They must begin with either a letter or an underscore character)

The result shown in Figure 5 is generally self-explanatory. The little pointer points to the **x** in the intended variable name, **1x**, reporting that this is a syntax error.

Variable name spelling errors

The interpreter assumes that you know what you are doing, and won't help you avoid spelling errors in variable names (unless the spelling error produces an illegal variable name).

A serious potential programming problem

Now I will illustrate a very subtle and serious potential problem. Consider the interactive code fragment in Figure 6. The programmer expected to get a final answer of 16+5 = 21, but instead the final answer was 11.

```
>>> xypdq = 6
>>> pzmbw = 5
>>> xypdq + pzmbw
11
>>> xyppq = 16 # accidental misspelling
>>> xypdq + pzmbw # correct spelling
11
>>>
```

Figure 6

Why did this happen?

The problem arose in the line with the boldface highlight. In this line, the programmer intended to assign a value of 16 to the existing variable named **xypdq**.

However, because of a spelling error, the programmer created a new variable named **xyppq** and assigned the new value of 16 to the new variable instead of assigning it to the existing variable.

As a result, the value stored in the original variable wasn't changed, and when that variable was used later in an expression, the result did not meet the programmer's expectations.

Spelling errors can be dangerous

This is one of the greatest dangers of using a programming language that doesn't require the declaration of variables. This type of spelling error is easy to make (as a result of a simple typing error), and can be extremely difficult to find and fix.

Defending against spelling errors

The best defense against this kind of error is to make all of your variable names meaningful. Then if you make a typing error (that results in a spelling error), you might have a better chance of finding it later.

Meaningful variable names

Some meaningful variable names follow. Note the judicious use of upper and lower case to visually break up the variable name into separate words. This is a naming convention that has become very popular, particular among Java programmers.

- myUpperLimit
- yourUpperLimit
- theOverheadRate
- theFinalPrice

Remember, case is significant in variable names

The variable named **MyUpperLimit** is not the same variable as the one named **myUpperLimit**.

As a practical matter, it is poor programming practice do distinguish between two variable names simply by using subtle differences in case. This will almost surely lead to spelling errors later.

Review

 1. A variable is the same as a constant, True or False?

Ans: False. The value of a variable is intended to change during the execution of the program. The value of a constant (which I haven't discussed yet) is not intended to change.

2. Python is a *strongly typed* language, True or False.

Ans: False. Python is not a strongly typed language. From a pure technical viewpoint, the Python programmer rarely needs to be concerned about type.

3. Python programmers must declare all variables, True or False?

Ans: False. Variable declarations are not required in Python. All that is required to cause a variable to come into existence is to invent a new name for a variable and assign a value to it.

4. Explain the dangers of using a language that does not require variables to be declared.

Ans: Several different kinds of problems can result from making typing errors that result in misspelling the names of variables. These errors usually result in programs that produce incorrect results without warning.

5. What is the best defense against spelling errors in variables names?

Ans: Use meaningful variable names for which the spelling is obvious, such as **theOverheadRate**.

6. Variable names can begin with the digit characters, True or False?

Ans: False. Variable names must begin with a letter or underscore character.

7. The underscore character should be used liberally in variable names, True or False?

Ans: False. You should use the underscore character in a variable name only when you know exactly why you are using it. Otherwise, you may create conflicts with special system variables whose names contain underscore characters.

8. Write a simple program that illustrates case sensitivity in the names of variables.

Ans: An example of such a program is shown in Figure 7.

```
>>> aX=10      # this is one variable
>>> Ax=20      # this is a different variable
>>> aX+Ax
30
>>>
```

Figure 7

Note that the names of the two variables have the same letters, but different case. The fact that the two variables are different variables is illustrated by the fact that each is assigned a different value. The sum of the two variables demonstrates that the two variables contain different, and correct, values.

Contrast the above result with the program in Figure 8 where a variable whose name contains the same letters and the same case is used.

```
>>> ax=10
>>> ax=20
>>> ax+ax
40
>>>
```

Figure 8

All this program accomplishes is the assignment of two different values to the same variable. It then adds the variable to itself using its current value of 20 producing a result of 40 (instead of 30 as in the previous example).

9. Explain the use of the assignment operator.

Ans: The assignment operator causes the value of its right operand to be stored in the memory location identified by its left operand.

10. Which type usually provides the greater range for storage of numeric values, integer or floating point?

Ans: Floating point usually provides the greater range for storage of numeric values.

11. Should you just always use floating point instead of integer to be safe?

Ans: Probably not. Floating point arithmetic often suffers from speed penalties. In addition, integer arithmetic produces exact results while floating point arithmetic usually produces approximate results (although the approximations may be very close to being exact in many cases).

12. Write a simple program that illustrates the approximation nature of floating point arithmetic.

Ans: See the sample program in Figure 9. We all know that the true result of this expression is an unending string of nines, as in 9.999999999999999...

```
>>> 10/3. + 20/3.
10.0
>>>
```

Figure 9

However, in this case, Python returned an answer of 10.0, indicating that the answer is accurate to three significant figures. This is not really the correct answer, but as a practical matter, it may be the *best* answer. I will leave that for you to decide.

13. Explain the purpose of the automatic continuation variable whose name is simply the underscore character.

Ans: The primary purpose of the automatic variable named _ is to make it easier for you to string calculations together in interactive mode and to display the intermediate results as you go.

Learn to Program using Python: Strings, Part I

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

I am taking it slow and easy for the first few lessons. My informal discussion is designed to familiarize you with the Python interactive programming environment while teaching you some important programming concepts at the same time.

This lesson provides an introduction to the use of strings.

What Is A String

 The common interpretation of the word string in computer programming jargon is that a string is a sequence of characters that is treated as a unit. For example, a person's first and last names are often treated as two different strings.

A person's first name usually consists of several characters, and these characters are treated as a unit to produce a name.

What is a literal?

Perhaps the best way to describe a literal is to describe what it is not.

A literal is not a variable. In other words, the value of a literal doesn't change with time as the program executes. You might say that it is taken at face value.

An expression using variables

For example, the following expression describes the sum of two variables named **var1** and **var2**:

```
sum = var1 + var2
```

The result of this expression can vary depending on the values stored in var1 and var2 at the instant in time that the expression is evaluated.

An expression using literals

On the other hand, the following expression describes the sum of two literal numeric values:

```
sum = 6 + 8
```

No matter when this expression is evaluated, it will always produce a sum of 14.

String literals

Literal values can also be used for strings.

```
>>> "Dick Baldwin"
'Dick Baldwin'
>>> 'Dick Baldwin'
'Dick Baldwin'
>>> Dick Baldwin
  File "<stdin>", line 1
    Dick Baldwin
               ^
SyntaxError: invalid syntax
>>>
```

Figure 1

For example, the interactive code fragment in Figure 1 shows

- My name entered three times, in three different ways, on the interactive command line (highlighted in boldface)
- The output from the interpreter for each entry.

Oops!

The first two entries are valid string literals. As you can see, in the first two cases, the interpreter displays my name in the output.

Note that in the first two cases, my name is surrounded by either quotes (sometimes called double quotes) or apostrophes (sometimes called single quotes).

A syntax error

However, the third entry is not a valid string literal, and the interactive interpreter produced a syntax error message. In the third case, my name is not surrounded by either double quotes or single quotes, and that is what produced the error.

So, what is a valid string literal?

According to the Python Reference Manual, string literals can be enclosed in matching single quotes (') or double quotes (").

This explains why the first two input lines in the above interactive code fragment were accepted and the third line produced an error.

Proper syntax

In the first line, my name was surrounded by matching double quotes. In the second input line, my name was surrounded by matching single quotes.

Bad syntax

However, in the third input line, my name was not surrounded by quotes of either type and this produced a syntax error.

More examples

Figure 2 shows two more examples of valid string literals with the input value highlighted in boldface.

```
>>> """Dick Baldwin"""
'Dick Baldwin'
>>> """Dick
... Baldwin"""
'Dick\012Baldwin'
>>>
Figure 2
```

(Note that I purposely colored the "**012**" in red to make it stand out. It was not that color in the original interpreter output. I will explain what it means later.)

What does """..."""" mean?

This syntax is explained by the following excerpt from the Python Reference Manual: "Strings can also be enclosed in matching groups of three single or double quotes (these are generally referred to as triple-quoted strings)." The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.
...
In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A ``quote'' is the character used to open the string, i.e. either ' or ".)

Use of triple quoted strings

One of the main advantages of using triple-quoted strings is that this makes it possible to

- Deal with strings that occupy more than one line
- Deal with all of the lines that make up the string as a unit
- Preserve newline characters that separate the lines in the process.

This is illustrated in Figure 3, which shows my name, surrounded by matching triple quotes and split onto two consecutive lines of input.

```
>>>"""Dick
... Baldwin"""
'Dick\012Baldwin'
>>>
```

Figure 3

The newline (\012) character

When this triple quoted, multiple-line input was displayed, by the interpreter, the display included "**\012**".

This is a numeric representation of the *newline* character. (I will show you another representation later.) It appeared in the output at the point representing the end of the first line of input. This indicates that the interpreter knows and remembers that the input string was split across two lines.

Why "represent" the newline character?

As the name implies, a newline character is a character that means, "Go to the beginning of the next line."

The newline character is sort of like the wind. You can't see the wind, but you can see the result of the wind blowing through a tree.

Similarly, you can't see a newline character, but you can see what it does. Therefore, we must represent it by something else, like **\012** if we want to be able to see where it appears within a string.

An escape sequence

The **\012** is what we call an escape sequence. I will discuss escape sequences in detail a little later.

One more syntax option

The Python Reference Manual describes one more syntax option for strings as shown below. I am going to let this one lie for the time being. I will come back and address it in a subsequent lesson if I have the time. I am including it here simply for completeness. "String literals may optionally be prefixed with a letter `r' or `R'; such strings are called raw strings and use different rules for backslash escape sequences...Unless an `r' or `R' prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C."

What Are Escape Sequences?

Escape sequences are special sequences of characters used to represent other characters that

- Cannot be entered directly into a string, or
- Would cause a problem if entered directly into a string.

The newline character

An example of the first category is the newline character. Except when using triple quoted strings, you cannot enter the newline character directly into a string.

Why? Because when you press the *Enter* key in an attempt to enter a newline, that simply terminates your input for that line. It doesn't enter the newline character into the string.

Using the newline character

The interactive code fragment in Figure 4 illustrates the use of an escape sequence to enter the newline character into a string. Note the **\012** between my first and last names.

```
>>> print "Dick\012Baldwin"
Dick
Baldwin
>>>
```

Figure 4

What does *print* mean?

This fragment uses a **print** statement. I haven't explained that statement to you before, but you can probably guess what it means.

When **print** is used interactively, it is a request to have its right operand (the expression to its right) printed on the next line. In this case, it is a request to have my name printed on the next line.

Including the newline character

In this fragment, I entered the newline escape sequence between my first and last names when I constructed the string. Then, when the string was printed, the cursor advanced to a new line following my first name and printed my last name on the new line. That is what escape sequences are all about.

print renders according to meaning

Note also that the **print** statement rendered the newline character according to its meaning.

What I mean by this is that the **print** statement did not print something that represented the newline character (**\012**) as we have seen before. Rather, it actually did what a newline character is supposed to do -- go to the beginning of the next line.

Escaping the quote character

Suppose that you are constructing a string that is surrounded by double quotes, and you want to use a pair of double quotes inside the string. If you were to simply enter the double quote when you construct the string, that quote would terminate the string.

The interactive code fragment in Figure 5 shows how to escape the double quote character -- precede it with a backslash character.

```
>>> print "Richard \"Dick\" Baldwin"
Richard "Dick" Baldwin
>>>
```

Figure 5

What I mean by this is that if you want to include a double quote inside a string that is surrounded by double quotes, you must enter the double quote inside the string as follows: \"

Avoiding the quote problem

Because this is such a common problem, and because the *escape solution* is so ugly and difficult to read, Python gives us another way to deal with quotes inside of quotes. This solution, shown in Figure 6, is the use of single and double quotes in combination.

```
>>> print 'Richard "Dick" Baldwin'
Richard "Dick" Baldwin
>>>
```

Figure 6

In Python, double quotes can be included directly in strings that are surrounded by single quotes, and single quotes can be included directly in strings that are surrounded by double quotes. This is much easier to read than the solution that requires you to place a lot of backslash characters inside your string.

List of escape sequences

A complete list of the escape sequences supported by Python is available in the Python Reference Manual.

More Ways to Span Lines

Just when you thought that you had seen it all, I am going to show you three more ways to span multiple lines with strings. One of them is shown in Figure 7.

```
>>> print "Richard \
... Baldwin"
Richard Baldwin
>>>
```

Figure 7

End the line with a backslash

As shown in Figure 7, the use of a backslash at the end of the line makes it possible to continue the string on a new line. However, the backslash is not included in the output, and there is no newline character in the output.

Not restricted to strings

Actually, the backslash can be used at the end of a line to cause that line to be continued on the next line whether inside a string or not. This is illustrated in the review section.

A form of concatenation

When used in this way with a string, the backslash at the end of the line becomes a form of string concatenation. The portions of the strings on each of the input lines are concatenated to produce a single line containing both parts of the string in the output.

I will have more to say about string concatenation later in this lesson.

Use the \n escape sequence

As shown in Figure 8, the inclusion of "**\n**" inside the string produces the same result as the inclusion of the numeric representation of the newline character, "**\012**" shown earlier.

```
>>> print "Richard \nBaldwin"
Richard
Baldwin
>>>
```

Figure 8

This is the common form of the newline escape sequence typically used in C, C++, and Java.

Combine backslash and \n

The code in Figure 9 shows how to combine the backslash at the end of the line with a newline character placed there to cause the output to closely resemble the input.

```
>>> print "Richard \n\
... Baldwin"
Richard
Baldwin
>>>
```

Figure 9

String Concatenation

To concatenate two strings means to hook them together end-to-end, thus producing a new string that is the combination of the two.

Literal string concatenation

You can cause literal strings to be concatenated just by writing one adjacent to the other as shown in Figure 10.

```
>>> print "Dick"'Baldwin'
DickBaldwin
>>> print 'Joe'          "Smith"
JoeSmith
>>>
```

Figure 10

Note that you can mix the different quote types and it doesn't matter if there is whitespace in between.

Creating whitespace

However, if you want any space between the substrings in the output, you must include that space inside the quotes that delimit the individual strings as shown in Figure 11.

```
>>> x = "Richard "
>>> y = " Baldwin"
>>> print x + "G." + y
Richard G. Baldwin
>>>
```

Figure 11

Using + for concatenation

The plus operator (+) can be used to concatenate strings as illustrated in Figure 11.

This fragment assigns string literal values to two variables, and then uses the plus operator to concatenate the contents of those variables with another string literal.

Of course, it could also have been used to concatenate the contents of the two variables without the string literal in between.

Whitespace is included in the quotes

Note that the string literals contain space characters. There is a space after the **d** in my first name and before the **B** in my last name. That is what I meant earlier when I said that if you want any space between the substrings in the output, you must include that space inside the quotes.

More on Strings

I will have more to say about strings in a future lesson. Before that, however, we need to learn how to create and execute script files, and we also need to learn a little more about Python syntax.

Review

1. Describe the common meaning of the word *string* in your own words, and give some examples.

Ans: The common interpretation of the word string in computer programming jargon is that a string is a sequence of characters that is treated as a unit. For example, a person's first and last names are often treated as two different strings.

2. Describe the common meaning of the word *literal* in your own words.

Ans: Perhaps one way to describe the meaning of the word literal would be that the literal item is taken at face value, and its value is not subject to change as the program executes.

3. Describe three different ways to format string literals (without spanning lines) and show examples.

Ans: Surround with matching pairs of single quotes, double quotes, or triple quotes as shown in Figure 12.

```
>>> print 'Dick Baldwin'
Dick Baldwin
>>> print "Tom Jones"
Tom Jones
>>> print """Mary Smith"""
Mary Smith
>>>
```

Figure 12

4. What is one of the advantages of using triple quoted strings? Show an example.

Ans: The use of triple quoted strings, as shown in Figure 13, makes it possible for you to continue a string on a new line, and to preserve the line break in the string.

```
>>> print """Dick
... Baldwin"""
Dick
Baldwin
>>>
```

Figure 13

5. Show two different representations of the *newline* character.

Ans: \012 and **\n** as shown in Figure 14. Of the two, the latter is probably the most commonly used, perhaps because it is easiest to remember.

```
>>> print 'Richard\012G.\nBaldwin'
Richard
G.
Baldwin
>>>
```

Figure 14

6. Describe, in your own words, the purpose of an escape sequence. Show two examples.

Ans: Escape sequences are special sequences of characters used to represent other characters that either

- Cannot be entered directly into a string, or
- Would cause a problem if entered directly into a string.

Examples are shown in Figure 15.

```
>>> print\
... "She said, \"He \nwon't go\""
She said, "He
won't go"
>>>
```

Figure 15

7. Show two different ways to include a double quote character in a string.

Ans: Surround with single quotes, or use an escape character as shown in Figure 16.

```
>>> print 'Richard "Dick" Baldwin'
Richard "Dick" Baldwin
>>> print "Richard \"Dick\" Baldwin"
Richard "Dick" Baldwin
>>>
```

Figure 16

8. Show the escape sequence for the tab character.

Ans: The escape sequence for the tab character is **\t** as shown in Figure 17.

```
>>> print "\tTom\n\tDick, and\n\tHarry"
        Tom
        Dick, and
        Harry
>>>
```

Figure 17

Learn to Program using Python: Writing and Using Scripts

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

I have been taking it slow and easy for the first few lessons. My informal discussion has been designed to familiarize you with the Python interactive programming environment while teaching you some important programming concepts at the same time.

It's time to learn about scripts

This lesson provides an introduction to the use of scripts, and of necessity will depart from the interactive mode used in previous lessons.

Some system stuff

Unfortunately, it will be necessary for me to get into some *system stuff* in this lesson.

Don't panic! I still plan to write this lesson at a level appropriate for beginning programmers.

It will be necessary for me to get into some *system stuff* that really has nothing in particular to do with Python programming. Rather, it will involve getting your computer set up for using scripts with Python.

What Is A Script?

Up to this point, I have concentrated on the interactive programming capability of Python. This is a very useful capability that allows you to type in a program and to have it executed immediately in an interactive mode.

But, interactive can be burdensome

By now you may have realized that you sometimes find yourself typing the same thing over and over. That is where scripts are useful.

Scripts are reusable

Basically, a script is a text file containing the statements that comprise a Python program. Once you have created the script, you can execute it over and over without having to retype it each time.

Scripts are editable

Perhaps, more importantly, you can make different versions of the script by modifying the statements from one file to the next using a text editor. Then you can execute each of the individual versions. In this way, it is easy to create different programs with a minimum amount of typing.

You will need a text editor

Just about any text editor will suffice for creating Python script files.

You can use *Microsoft NotePad, Microsoft WordPad, Microsoft Word,* or just about any word processor if you want to.

The Programmer's File Editor

I'm fond of an editor named *The Programmer's File Editor*. You can learn about it at the following URL: http://www.lancs.ac.uk/people/cpaap/pfe/

Arachnophilia and NoteTab

Some other editors that I like are *Arachnophilia*, which is available at http://www.arachnoid.com/arachnophilia/ and *NoteTab*, which is available at http://www.notetab.com/.

Must be a plain text editor

Whichever editor you choose, make certain that it produces plain ASCII text in its output (no bold, no underline, no italics, etc.).

Combining scripts with interactive mode

It is also possible to combine script files with interactive mode to incorporate pre-written scripts into interactive programs.

Getting Started

In order to use script files, you must prepare you computer to use them. This doesn't amount to much in the way of effort, but it is critical.

Where is Python located?

When you first installed your Python software, a directory should have been created somewhere on your hard drive containing a file named **python.exe**.

On a Windows system, unless you forced the program to be installed somewhere else, it was probably installed somewhere on your C-drive. I forced the program to be installed on my D-drive.

On my machine, the file named **python.exe** is in a directory named **Python**, which in turn is contained in a directory named **Program Files** on my D-drive.

My directory listing

To help you get oriented, here is a list of files appearing in my **Python** directory running under the WinNT 4.0 Workstation operating system. The file mentioned above is highlighted in boldface.

- INSTALL.LOG
- py.ico
- pyc.ico
- pycon.ico
- **python.exe**
- pythonw.exe
- UNWISE.EXE

What about your files?

If you are running under Windows, I expect that your **Python** directory will contain pretty much the same set of files.

However, if you are using some other operating system, your list of files may be different, but hopefully there will still be a file named **python.exe** (or the equivalent executable file for your system).

Setting the path

You will need to cause the directory containing the file named **python.exe** to be listed in your system environment variable named **path**.

Do you know how to set the path?

If you already know how to set the path, go ahead and do it. If you don't already know how, you may need to get some help.

I'm not going to try to tell you how to do it, because the procedure varies from one operating system to the next, and if you don't do it correctly, you may cause problems that are difficult to recover from. (I don't want to be responsible for that.)

Get help if you need it

So, if you need help in setting the path, get it, but make sure that the person helping you knows what he is doing.

Create a script file

Once you have the path variable properly set, use any plain text editor and create a file named **junk.py** that contains the Python programming statements shown in Figure 1. (Note that the figure number at the bottom is not part of the program.)

```
a=2
b=3
a=2*a
b=3*b
c=a+b
print c
```

Figure 1

Where do I put the script file?

Store this file in any directory on your hard drive. You may want to create a new directory for the sole purpose of storing Python script files.

A command prompt window

Then open a *command prompt* window and make the directory containing your new script file become the current directory.

How do I open a command prompt window?

With MS Windows, you can open a command prompt window by pulling up the *Start* menu, selecting *Programs*, and then selecting *Command Prompt*. At least this is how it is done on WinNT 4.0 Workstation. Win95 and Win98 should be similar.

The procedure will be different if you are using some other operating system.

What is the current directory?

If you don't know how to cause a particular directory to become your *current directory*, you will need to get someone who knows how to navigate the directory structure to show you. It isn't rocket science, but it will be different for different operating systems, so you will need to know how to do it on your operating system.

Running your script file

Once you have accomplished all of the above, you should be able to enter the following command at the command prompt and you should see the result of executing your script appear on the screen.

python junk.py

Figure 2 shows what my screen look like when I do this. (Again, as a reminder, the figure number at the bottom is not part of the program output.)

```
D:\Baldwin\AA-School\PyProg>python junk.py
13
D:\Baldwin\AA-School\PyProg>
```

Figure 2

I highlighted my command

I highlighted the command that I entered in boldface (in Figure 2) to separate it from the command prompt and the output produced by the program.

Is this the correct output?

If you go back to the script file and do the arithmetic, you will see that the output value of 13 produced by the program is correct.

Congratulations are in order

If you got an output value of 13, -- congratulations -- you have just written and executed your first Python script.

What's Next

Obviously, there is a lot more that you will need to learn before you can write that "killer script" that takes the world by storm, but at this point, you have the tools to experiment with some simple scripts.

Practice, practice

I recommend that you look back into the earlier lessons and convert some of the interactive programs listed there into scripts, execute them, and confirm that the scripts behave as expected.

In the future...

In future lessons, I will switch back and forth between scripts and interactive mode, depending on which seems to be the most appropriate at the time.

I like cut-and-paste programming

However, I am a strong advocate of cut-and-paste programming.

Cut-and-paste programming works well with scripts, and not so well with interactive mode.

Therefore, any time there is very much typing involved, you can usually expect to see me using scripts instead of interactive mode.

Review

1. In your own words, what is a script?

Ans: A script is a text file containing the programming statements that comprise a Python program.

2. A script is a one-time affair, True or False?

Ans: False. Script files are reusable.

3. Convert the interactive program shown in Figure 3 into a script, execute it, and confirm that you get the correct result.

```
>>> x=6     # create and populate x
>>> y=5     # create and populate y
>>> x+y     # add x to y and display the sum
11
>>>
```

Figure 3

In case you didn't get any output, note that unlike in interactive mode, in order to cause a script to produce an output, you will need to use a statement something like the following:

print x+y

4. Convert the interactive program shown in Figure 4 into a script and execute it.

```
>>> a=b=c=10    # assign 10 to several variables
>>> a+b+c       # add them together
30
>>> a=b=c=20    # assign 20 to same variables
>>> a+b+c       # add them together
60
>>>
```

Figure 4

5. Convert the interactive program shown in Figure 5 into a script and execute it.

```
>>> x=5
>>> y=6
>>> x+y
11
>>> x=5.55555
>>> y=6.66666
>>> x+y
12.22221
>>>
```

Figure 5

6. Convert the interactive program shown in Figure 6 into a script and execute it.

```
>>> 5+6
11
>>> _+22 # add 22 to the continuation variable
33
>>>
```

Figure 6

If you were unable to get this to work, don't be dismayed. The special variable whose name is the underscore character is available only in interactive mode. Therefore, you can't use that variable in a script, and you will need to develop a workaround.

7. Convert the interactive program shown in Figure 7 into a script and execute it.

```
>>> print "Dick\012Baldwin"
Dick
Baldwin
>>>
```

Figure 7

Learn to Program Using Python: Program Construction

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Program Construction

Conceptually, programs are composed of statements, and statements are composed of expressions. In practice, you need to know how to construct statements from a physical viewpoint.

Line structure

A Python program is divided into a number of *logical lines*. A logical line is constructed from one or more *physical lines.*

What is a physical line?

A physical line ends with the character(s) used by your platform for terminating lines.

On Unix, this is the *linefeed* character. On DOS/Windows, it is the *carriage return* character followed by the *linefeed* character. On Macintosh, it is the *carriage return* character.

We refer to this in programming jargon as the *newline* character (even though it is actually two characters on DOS/Windows systems).

Please be more specific

Basically, a physical line is what you get when you type some characters into your text editor and press the *Enter* key, the *Return* key, or whatever you call that key on your keyboard. (Actually you don't even need to type characters before pressing the *Enter* key, in which case you get blank lines.)

This is the key that causes the cursor to go down to the next line and return to the left side of the editing window.

For example, on my Dell laptop, I pressed the **Enter** key immediately following the --> in the two lines shown in Figure 1.

```
I will end the line here -->
and end the next line here -->
```
Figure 1

This produced a *newline* in each case. I also pressed the *Enter* key a couple more times following that as well.

A side trip into HTML

Just in case you are interested in HTML, the Netscape Composer program that I am using to create this document inserts a
 tag into the HTML code each time I press the *Enter* key (but that has nothing to do with Python, or computer programming either for that matter).

What is a logical line?

A *logical line* is constructed from one or more *physical lines.*

Line joining rules (described later) can be used to construct a logical line from two or more physical lines.

Statements and logical lines

A statement cannot cross logical line boundaries except where the syntax allows for the *newline* character, such as in *compound statements*. I will show you an example of a compound statement later in this lesson.

Comments

We learned about comments in an earlier lesson. To summarize, a comment starts with a hash character (#) that is not part of a string literal (we also learned about string literals in an earlier lesson). The comment ends at the end of the physical line.

Explicit line joining

You can join two or more *physical lines* to produce a *logical line*, using the backslash character as shown in Figure 2.

```
a = 3
b = 4
# construct c=a+b
c \
= \
a \
+ \
b
print c

# the output is 7
```

Figure 2

The backslash character

When a *physical line* ends in a backslash that is not part of a string literal or comment, that line is joined with the following *physical line* forming a single *logical line*.

The backslash and the following end-of-line character are deleted (or ignored by the compiler) and do not become part of the logical line.

In Figure 2, the expression **c=a+b** is created by joining five consecutive physical lines to create one logical line.

For illustration only

Obviously, this is not how you would want to write a long program, but it is not unusual to break long expressions into two or more physical lines to make them fit onto a prescribed page width.

No spaces or comments allowed

Be careful to make certain that no space characters follow the backslash.

You may not place a comment following the backslash, and a backslash does not continue a comment.

Otherwise illegal

A backslash is illegal elsewhere on a line except inside a string literal.

Implicit line joining

Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line without using backslashes as shown in Figure 3.

```
a = 3
b = 4
c = (a # continue on next line
     + b)
print c

# the output is 7
```

Figure 3

You can also place a comment on a line that is being continued implicitly as I did in the third line of Figure 3.

Also, you can indent the continuation line however you want. This is very useful for making the code more readable.

What about blank lines?

Lines containing only spaces, tabs, formfeeds, and comments are ignored in scripts, but the behavior may be different in interactive mode, depending on the implementation.

An example of some spaces and blank lines is shown in Figure 4.

```
a = 3
  # spaces and comment
b = 4
 # tab and comment
c = (a # continue on next line
     + b)
# a blank line follows

print c

# the output is 7
```

Figure 4

Indentation

In every programming environment that I have worked with in the past, indentation is used strictly for cosmetic or readability purposes.

Not just for cosmetics in Python

However, indentation is such an important topic in Python that I have separated it into a major section in this tutorial lesson.

Used to determine grouping of statements

Unlike most other programming environments, physical indentation is used in Python to determine the grouping of statements.

A blessing and a curse

This can be both a blessing and a curse. The blessing is that it forces you to use proper indentation, which usually leads to more readable code.

The curse is that if you don't use proper indentation, your program probably won't behave properly.

No safety nets

There are no safety nets in Python (such as the matching curly braces used in C, C++, and Java) to protect you from indentation errors.

To make matters worse, such errors often turn up as logical errors (meaning that the program simply doesn't work correctly) rather than compiler errors (meaning that the compiler or interpreter will tell you about the error).

An example of correct indentation

Although I haven't introduced you to the **if** statement yet, you probably have a fairly good idea what it is used for. I am going to use it to illustrate the proper indentation of a *group of statements*.

An **if** statement means that if some expression evaluates to true, do something specific. Otherwise, don't do it.

Don't be too concerned if the logic of this program escapes you at this point in time. I will explain the use of the **if** statement in detail in a subsequent lesson. The important thing here is to understand the grouping of statements.

Compare A with B and take appropriate action

The sample program in Figure 5 compares the value of the variable **A** with the value of the variable **B** (**if B > A:**).

```
A = 3
B = 4

if B > A:
  print A # begin group
  print B
  print (A + B) # end group
A = 6 # not part of above group
print A
```

Figure 5

The program logic

If the value of **B** is greater than the value of **A** (which it is in this case), the three (red) statements are executed. Otherwise, that group of three statements is bypassed.

(Note that I made the statements red here to make them stand out. You would not make them red when actually writing the program.)

A compound statement

The three statements shown in red are either all executed, or they are all bypassed. Hence, they behave as a group.

A group of statements like this is sometimes referred to as a *compound statement* (a statement made up of two or more individual statements).

Group behavior

When the three statements are executed as a group, the values 3, 4, and 7 are printed on consecutive output lines by the three print statements in the group of statements, as shown in Figure 6.

```
D:\Baldwin\AA-School\PyProg>python junk.py
3
4
7
6

D:\Baldwin\AA-School\PyProg>
```

Figure 6

Last statement is not part of the group

Then the value 6 is printed by the last statement in the program, *which is not part of the group*.

Another sample program

Now, let's make a change. The program shown in Figure 7 is identical to the one above except that I switched the values of **A** and **B** to cause the group of red statements to be bypassed (**B** is no longer greater than **A**).

```
A = 4
B = 3

if B > A:
  print A # begin group
  print B
  print (A + B) # end group
A = 6 # not part of above group
print A
```

Figure 7

The output

In this case, the output screen looks something like Figure 8. Only one value (6) is printed because the three print statements in the group were bypassed as a group.

```
D:\Baldwin\AA-School\PyProg>python junk.py
6

D:\Baldwin\AA-School\PyProg>
```

Figure 8

What makes them into a group?

The important point here is that the three red statements constitute a group ***because of their common indentation level***.

Emphasis added

(Note that I used boldface and red in the above programs to emphasize certain parts of the program. The original script did not contain boldface and did not contain any red color. Python scripts must be written in plain text, and control codes, such as bold or color, are not allowed.)

An opinion

I personally don't like the idea of using indentation to create grouping. Although it sounds nice in theory, it can be very labor intensive in practice. Once you have written a script, one simple change can often require you to go back and modify the indentation level of almost every statement in the script.

I guess the good news is that this will certainly encourage you to write your program as a series of short, concise independent modules rather than as a single long rambling program.

Details, details

Now let's talk about the details of indentation.

Leading whitespace

Leading whitespace (spaces and tabs) at the beginning of a *logical line* is used to compute the indentation of the line. This, in turn, is used to determine the grouping of statements.

What about tabs?

My advice is to avoid the use of tabs altogether. Use spaces instead, and use the same number of spaces for each statement in the group.

However, if you must use tabs, there are a few things that you should know. Here is what the Python Reference Manual has to say on the subject.

> "First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation.
>
> Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.
>
> **Cross-platform compatibility note:** because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file."

The bottom line on indentation

Use either spaces, or tabs, but not both to cause the indentation level of all statements in a group of statements to be indented to the same level.

Review

1. Logical lines are constructed from one or more _____ (fill in the blank).

Ans: Logical lines are constructed from one or more physical lines.

2. What constitutes a physical line?

Ans: A physical line is what you get when you type some characters into your text editor and press the *Enter* key, the *Return* key, or whatever it is called on your keyboard.

3. Can statements cross logical line boundaries?

Ans: Yes, but only in those cases where the syntax allows for the *newline* character, such as in *compound statements*.

4. What character is used for explicit line joining?

Ans: The backslash character can be used to join two physical lines into one logical line.

5. Explain implicit line joining in your own words.

Ans: Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line without using backslashes.

6. What are the indentation rules for lines that have been implicitly joined?

Ans: You can indent the continuation line however you want.

7. Indentation in Python is used for cosmetic purposes only. True or False?

Ans: False. Indentation is used in Python to determine the grouping of statements, which causes it to be critical to the success of a program.

Strings, Part II

Preface

 This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Lesson 1 provides an overall description of this online programming course.

Introduction

What you have learned

You have learned how to write some simple programs and execute them interactively.

You have learned how to capture simple programs in script files and to execute those script files.

You have learned how to construct programs, including the indentation concepts involved in Python.

You have also learned some of the fundamental concepts involving strings.

What you will learn

This lesson will expand your knowledge of strings, and in addition will introduce you to some concepts that will be useful with other data types as well: *indexing* and *slicing*.

What is indexing?

According to a definition that I found on the web, *"... an* ordinal number *is an adjective which describes the numerical position of an object, e.g., first, second, third, etc."*

A practical example

Many years ago when I did a tour of duty as an enlisted man in the U.S. Air Force, they had a habit of lining us up and requiring us to *"count off."*

What this meant was that the first person in the line called out the number *one*, the person behind him called out the number *two*, the person behind him called out the number *three*, etc. (Since learning about computer programming, I now wonder if the first person should have called out *zero*.)

Assigning an ordinal index

I'm sure they didn't realize that what they were doing was *assigning an ordinal index value* to each person in the line (and neither did I at the time).

Using an ordinal index

Even though they didn't know the technical details of ordinal indices, they didn't have any difficulty saying, *"Number six, wash dishes, number fourteen, peel potatoes, number twenty-two, carry out the garbage, etc."*

This is indexing

That is what indexing is all about.

> In the context of this lesson, indexing is the process of assigning an ordinal index value to each data item contained in some sort of a container.

In other words, we assign an ordinal number to each item, which describes the numerical position of the item in the container.

> For example, if you were very careful, you could use a felt tip pen to assign an ordinal index to each of the twelve eggs contained in a carton containing a dozen eggs. (Should you start with *zero* or *one*?)
> Then you could extract the egg whose index value is 9 from the container and eat it for breakfast.

Having assigned the index, we can use that index to access the data item corresponding to that index, as in *"Number six, wash dishes."*

(Note that this process is also referred to as a *subscription* in the Python Reference Manual.)

Index values automatically assigned

In this lesson, we will be using the index values that are automatically assigned to the characters in a string for the purpose of accessing those characters, both individually, and in groups.

What is slicing?

Here is what Magnus Lie Hetland has to say on the topic of slicing (and indexing as well.) Although this quotation was taken from a discussion of lists, it applies equally well to strings.

> "One of the nice things about lists is that you can access their

elements separately or in groups, through indexing and slicing.

Indexing is done (as in many other languages) by appending the index in brackets to the list. (Note that the first element has index 0). **(This is the answer to the question about the first egg -- Baldwin)**
...
Slicing is almost like indexing, except that you indicate both the start and stop index of the result, with a colon (":") separating them:
..
Notice that the end is **non-inclusive**. If one of the indices is dropped, it is assumed that you want everything in that direction. i.e. list[:3] means "every element from the beginning of list up to element 3, non-inclusive."
...
list[3:] would, on the other hand, mean "every element from list, starting at element 3 (inclusive) up to, and including, the last one."

For really interesting results, you can use negative numbers too: list[-3] is the third element from the end of the list..."

Some material deleted for brevity

I added the boldface and the red comment for emphasis. I also deleted some of the material from this quotation for brevity, but I will cover that material later in conjunction with my discussion of indexing and slicing strings.

A Sample Program

I will illustrate indexing and slicing of strings using a sample program contained in a script file named **String01.py**.

The program listing

A complete listing of the program, and the output produced by the program, are provided at the end of the lesson.

Will discuss in fragments

I will discuss the program in fragments, illustrating particular aspects of indexing and slicing in each fragment. This is a scheme that I will use frequently in this set of tutorial lessons.

Interesting Code Fragments

A single character can be extracted from a string by referring to the string and indicating the index of the character in square brackets, as shown in the code fragment in Figure 1.

```
aStr = "This is a string"
```

```
print aStr[0] #print T
print aStr[3] #print s
```
Figure 1

(Note that this is a fragment from a script file, not from an interactive Python program.)

First create a string to work with

The fragment in Figure 1 creates a string (highlighted in boldface) and assigns it to a variable named **aStr**. From this point on, the contents of the string can be accessed by referring to the variable.

Index values always begin with zero

Unlike eggs and Air Force enlisted men, the first character in a string is always located **at index 0**, as in **aStr[0]**.

Thus, the second statement in the fragment extracts and prints the **T**, which is the first character in the word **This**.

Character at index 3, display yourself

Similarly, the last statement in the fragment extracts and prints the **s** from index position 3, as in **aStr[3]**.

The character at this index position is the **s** that ends the word **This**.

The last statement is equivalent to the following request, "Will the character at index position 3 please display yourself on the screen."

Is this the fourth character?

(You would probably refer to this as the fourth character, and you would be correct if you did. The character at index 3 is the fourth character in the string. The character at index 0 is the first character in the string. First **does not** equate to index 1. You need to think about this, because this can be a confusing topic for new programmers.)

An important, and potentially confusing point

At the risk of becoming boring, there is an important point here that you might as well get used to right now.

The **s** is located at index value 3. However, according to the way you are probably accustomed to counting, this is actually the fourth character in the string. You might be inclined to refer to this character as character number 4.

This is because index values always begin with zero, while you are probably accustomed to counting things beginning with *one*, not *zero*.

Not like eggs

If you access the egg at index value 4 in the container and eat it for breakfast, it cannot be accessed again (because it will be gone).

However, if you access the character at index value 4 in the string and use it for some purpose, what you really use is a copy of the character. It is still there and can be accessed again.

(Some data containers do allow for the removal of data elements in much the same sense that we can remove an egg from its container. However, a string is not such a container.)

A simple slice

For convenience, here is another copy of the fragment that created the string.

```
aStr = "This is a string"
```

The fragment in Figure 2 cuts a couple of slices out of that string and displays them on the screen.

```
print aStr[0:4] #print This
print aStr[10:16] #print string
Figure 2
```

Slice Notation

The slice notation uses two index values separated by a colon, as shown in boldface in Figure 2.

The end is non-inclusive

As was indicated in the earlier quotation, "*... the end is non-inclusive.*" This means that the character whose index value is the number following the colon is *not included in the slice*.

Extract the first word in the string

Thus, the first statement containing the reference **aStr[0:4]** extracts and prints the character sequence beginning with index value **0** and ending with index value **3** (not 4). This causes the word **This** to be extracted and printed.

Extract the last word in the string

Similarly, the second statement in the above fragment (**aStr[10:16]**) extracts and prints the characters having index values from 10 through 15, inclusive (not 16). This causes the word **string** to be extracted and printed.

Omitting the first index

If you omit the first index value, as shown in Figure 3, it defaults to the value zero.

```
print aStr[:4] #print This
Figure 3
```

Therefore, the statement in Figure 3 extracts and prints the first word in the string, which is **This**.

Omitting the second index

If you omit the second index, as shown in Figure 4, it defaults to a value that includes the last character in the string.

```
print aStr[10:] #print string
Figure 4
```

Thus, the statement in Figure 4 extracts and prints the last word in the string, which is **string**.

Print the entire string

Figure 5 shows two different ways to extract and print the entire string. I won't comment on this, but will leave the analysis as an exercise for the student.

```
print aStr[:5] + aStr[5:]
print aStr[:100]
Figure 5
```

(Hint: Remember that the plus sign when used with strings is the string concatenation operator.)

Print an empty string

There are several ways that you can specify the index values that will produce an empty string. One of those ways is shown following the plus sign in Figure 6.

```
print "Empty: " + aStr[16:100]
Figure 6
```

In Figure 6, both index values are outside the bounds of the index values of the characters in the string, which range from 0 through 15 inclusive.

Negative indices

Although it can be a little confusing, negative index values can be used to count from the right, as shown in Figure 7.

```
print aStr[-5:-2] #print tri
```

Figure 7

This fragment extracts and prints the characters **tri** from the word **string**, which is the last word in the string.

Eliminating confusion

Once you allow negative indices for slicing, thing can become very confusing. The following explanation of how indices work with slicing is attributed to Guido van Rossum.

In this example, Mr. van Rossum is referring to a five-character string with a value of "**HelpA**".

```
The best way to remember how slices
work is to think of the indices as
pointing between characters, with the
left edge of the first character
numbered 0. Then the right edge of the
last character of a string of n
characters has index n, for example:

     +---+---+---+---+---+
     | H | e | l | p | A |
     +---+---+---+---+---+
     0   1   2   3   4   5
    -5  -4  -3  -2  -1

The first row of numbers gives the
position of the indices 0...5 in the
string; the second row gives the
corresponding negative indices.

The slice from i to j consists of all
characters between the edges labeled i
and j, respectively.

For nonnegative indices, the length of
a slice is the difference of the
indices, if both are within bounds,
e.g., the length of word[1:3] is 2.
```

Hopefully, this explanation will help you to understand and to remember how index values are used for the extraction of substrings from strings using slicing.

Getting the length of a string

And finally, a built-in function named **len()** can be used to determine the number of characters actually contained in a string as shown in Figure 8.

```
print len(aStr)
```

Figure 8

For the example string used in this lesson, Figure 8 gets and prints the length of the string as 16.

If you count the characters in the string (beginning with 1), you will conclude that there are 16 characters in the string.

Note the difference between the number of characters and the maximum index value

For a string containing 16 characters, the valid index values range from 0 through 15 inclusive.

The complete output

This Python script file produces the output shown in Figure 9 on my computer (boldface added for emphasis).

```
D:\Baldwin\AA-School\PyProg>python strings01.py
T
s
This
string
This
string
This is a string
This is a string
Empty:
tri
16

D:\Baldwin\AA-School\PyProg>
```

Figure 9

A String Is Immutable

There is one more point that needs to be made here. Although you can use indexing and slicing to access the characters in a string, you cannot use indexing and slicing to assign new character values to those characters.

This is because a Python string is *immutable*. In other words, after it is created, it cannot be modified.

Complete Program Listing

A complete listing of the program follows is shown in Figure 10.

```
# File String01.py
```

```
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates indexing and
# slicing strings
#
#------------------------------

aStr = "This is a string"
print aStr[0] #print T
print aStr[3] #print s
print aStr[0:4] #print This
print aStr[10:16] #print string
print aStr[:4] #print This
print aStr[10:] #print string

#print the entire string
print aStr[:5] + aStr[5:]
print aStr[:100]

#print an empty string
print "Empty: " + aStr[16:100]

#count from the right
print aStr[-5:-2] #print tri

#get the length of the string
print len(aStr)
```

Figure 10

Review

1. What is an ordinal number?

Ans: An ordinal number is an adjective, which describes the numerical position of an object, e.g., first, second, third, etc.

2. What is indexing?

Ans: Indexing is the process of assigning an ordinal index value to each data item contained in some sort of a container.

3. We assign index values to the characters in a string, True or False?

Ans: False. Index values are automatically assigned to the characters in a string.

4. What is the syntax for accessing a character at a particular index in a string?

Ans: Refer to the string, and include the index value in square brackets, as in **aStr[3]**.

5. What is the syntax for accessing a substring from a string using slicing?

Ans: Include both the start and stop index in square brackets, separated by a colon, as in **aStr[10:16]**.

6. The second index of a slice is inclusive, True or False?

Ans: False. The character whose index value matches the second index of a slice is not included in the slice.

7. What is the default value for the first index if you omit the first index value in a slice?

Ans: The default value is zero (0).

8. What is the default value for the second index if you omit the second index value in a slice?

Ans: The default value is the length of the string (the actual number of characters in the string).

9. What is the purpose of negative slice indices?

Ans: Negative indices can be used to count from the right end of the string.

10. What is the name and syntax of the function that can be used to find the length of a string?

Ans: len(theString)

Lists, Part I

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along.

Introduction

What you have learned

You have been moving along rather briskly in learning how to program using Python. For example, in an earlier lesson you learned how to *index* and *slice* strings.

Introducing lists

In this lesson, we will extend that knowledge to a new type of data called a *list*. This type of data, along with a string, is referred to as a *sequence*.

A little backtracking

First, however, I want to nail down some terminology from the Python reference manual.

What is a Subscription?

I referred to a *subscription* as an *index* in the lesson on strings.

The following discussion of a *subscription* is based on the Python Reference Manual.

So, what does a subscription do?

A subscription selects an item of a *sequence* (*string*, *tuple* or *list*) or *mapping* (*dictionary*) object, as in the following:

primary "[" expression_list "]"

Don't panic! This is not as complicated as it appears, as you will see shortly.

(I discussed *strings* earlier. I will discuss *tuples, lists,* and *mappings* later.)

What is the *primary*?

According to the Reference Manual, the primary must evaluate to an object of a *sequence* or *mapping* type.

For example, in the lesson on strings, the *primary* was a reference to a string named **aStr**, as shown in Figure 1.

```
aStr = "This is a string"
print aStr[0]
print aStr[3]
```
Figure 1

Will continue to call it an index

I will probably continue to refer to a *subscription* as an *index* most of the time, simply because I believe that index is the more commonly recognized term.

What is a Sequence?

Python provides three kinds of sequences: *strings*, *lists*, and *tuples*.

I discussed strings in a previous lesson. I will discuss lists in this lesson, and I will discuss tuples in a future lesson.

What about the expression_list?

According to the Reference Manual, if the primary is a sequence, the **"expression_list"** must evaluate to a plain integer.

An example of an integer expression_list

In the string example above, the plain integer was **0** in one case and **3** in the other.

Negative integers

If the (index) value is negative, the length of the sequence is added to it to obtain the actual index used to access the sequence.

For example, **aSequence[-1]** selects the last item of the sequence.

Example negative integer

This is illustrated in the interactive code fragment shown in Figure 2, which prints the last character in a string.

```
>>> aStr = "xyz"
>>> print aStr[-1]
z
>>>
```

Figure 2

More negative-integer rules

The value resulting from adding the length of the sequence to the specified index value must be a non-negative integer less than the number of items in the sequence.

Then, the subscription selects the item whose index is that value (counting from zero).

Good and bad negative integers

This is illustrated in the interactive code fragment shown in Figure 3, which shows both valid and invalid negative subscription values. (The subscription value of -4 violates the above rule, thus producing an *IndexError*.)

```
>>> aString = "xyz"
>>> print aString[-1]
z
>>> print aString[-2]
y
```

```
>>> print aString[-3]
x
>>> print aString[-4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>>
```
Figure 3

What about a character type?

There is no character type in Python. Rather, a string's items are characters. A character is not a separate data type but a string of exactly one character.

What is a Mapping?

A *mapping* is a structure in which values are stored and retrieved according to a *key*. This is often called a dictionary, because it behaves similarly to a common dictionary.

A real dictionary example

For example, if I want to know about the word Python, I get out my Webster's Seventh New Collegiate Dictionary, (which is about forty years old), and I look up the word *python* (in this case, python is the key).

The (approximate) value associated with the key is, *"... monstrous serpent killed by Apollo ..."*

Is this a definition?

In normal conversation, we frequently refer to the value obtained from a common printed dictionary as the *definition* of the word. However, definition usually means something completely different in computer jargon.

In computer jargon, we refer to it as the *value* associated with the word (*key*). Thus, a *mapping*, is a structure that associates *values* with *keys*.

So, a common printed dictionary is an example of a mapping.

No mention of the Python language

Since my dictionary is much older than the Python programming language, I wouldn't expect to find anything about Python programming there.

What about the expression_list?

According to the Reference Manual, if the primary is a *mapping*, the **expression_list** must evaluate to an object whose value is one of the keys of the mapping (such as python in my common dictionary example).

What gets selected?

Then the subscription selects the value in the mapping that corresponds to that key.

In other words, the system looks up the word in the dictionary and returns the value that corresponds to that word.

More to say about mappings

I will have more to say about *mappings* in a future lesson.

What is a Slicing?

I discussed *slicings* at some length in the lesson on strings, using a different source for my information. In this lesson, I will paraphrase information extracted from the Python Reference Manual.

The semantics for a simple *slicing* are as described in the following paragraphs.

What about the primary?

To begin with, the primary must evaluate to a sequence object (a *string*, a *list*, or a *tuple*). In other words, **a dictionary structure cannot be sliced**.

A slicing specifies two numeric bounds

The lower and upper bound expressions, if present, must evaluate to plain integers, such as in the statements from the lesson on strings shown in Figure 4.

```
print aStr[0:4]
print aStr[10:16]
```
Figure 4

What about default values?

Both the lower and upper bounds have default values. The default values are *zero* and the *sequence's length*, respectively.

Figure 5 shows some examples of using default values from the lesson on strings.

```
print aStr[:4]
print aStr[10:]
```
Figure 5

What about negative bounds?

If either bound is negative, the sequence's length is added to it. The slicing then selects all items with index **k** such that **i <= k < j** where **i** and **j** are the specified lower and upper bounds.

This may be an empty sequence. It is not an error if **i** or **j** lie outside the range of valid indexes (such items don't exist so they aren't selected).

An example of negative bounds

Figure 6 shows an example of the use of negative bounds from the previous lesson on strings.

```
print aStr[-5:-2]
```
Figure 6

Reducing confusion

The previous lesson on strings also contains a diagram from Guido van Rossum that helps to clarify the complexity surrounding the use of negative values when slicing. If this discussion on negative indices is confusing, you might want to go back and take a look at that diagram.

What is a Mutable Sequence?

According to the Python Reference Manual, *"Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and del (delete) statements."*

There is currently a single mutable sequence type in Python, and it is a List.

What is a List?

According to the Reference Manual, *"The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets."*

Now, for a less formal discussion

The above discussion may have seemed a little heavy for an online programming course aimed at beginning programmers. However, there are certain things that we must define in a reasonably formal way in order to continue to make progress.

The remainder of this lesson will be a little less formal, and more in the style that you have come to expect in this set of programming tutorials.

Some Sample Programs

Creating, indexing, and slicing lists

A list can be written as a sequence of comma-separated values (items) surrounded by square brackets.

Lists can be nested within other lists.

List items do not all have to be of the same type.

An example list

The short Python script shown in Figure 7 creates a simple list containing four elements of different types. The types of the elements are respectively, a *float* value, an *integer*, a *string*, and another *integer*.

```
# File Lists01.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates creating,
#  indexing, and slicing lists.
#
#--------------------------------

theList = [3.14,59,"A string",
           1024]
print "Print index value 2"
print theList[2]
print "Print a short slice"
print theList[0:3]
print "Print the entire list"
print theList[:100]
```

Figure 7

Print an item using a subscription

After creating the list, the program uses a subscription (index) to extract and print the value at index 2 (remember the first item is at index 0).

Print some slices

Then it uses the slice notation to extract and print two different slices from the list.

The first slice extracts and prints the elements from index 0 through index 2 inclusive. (Remember, the items selected by a slice do not include the index specified by the upper limit value, which is 3 in this case.)

The second slice extracts and prints the entire list. If you don't understand these two slices, go back and review the lesson on strings where I discuss slicing in detail.

Program output

The output from this program is shown in Figure 8.

```
D:\Baldwin\AA-School\PyProg>python Lists01.py
Print index value 2
A string
Print a short slice
[3.14, 59, 'A string']
Print the entire list
[3.14, 59, 'A string', 1024]

D:\Baldwin\AA-School\PyProg>
```

Figure 8

Lists can be concatenated

Lists can be concatenated using the **+** operator.

The Python program shown in Figure 9 creates two lists and prints them both. Then it concatenates the two lists and prints the concatenated version.

```
# File Lists02.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates concatenating
#  lists
#
#--------------------------------
print "Create two lists"
listA = [3.14,59,"A string",
           1024]
listB = [2,4,6,16]
print "Print listA"
print listA
print "Print listB"
print listB
print "Concatenate the lists"
listC = listA + listB
print "Print concatenated list"
print listC
```

Figure 9

Program output

The output from this program is shown in Figure 10 (boldface added for emphasis). As you can see, the concatenated list contains the elements of both of the individual lists.

```
D:\Baldwin\AA-School\PyProg>python Lists02.py
Create two lists
Print listA
[3.14, 59, 'A string', 1024]
Print listB
[2, 4, 6, 16]
Concatenate the lists
Print concatenated list
[3.14, 59, 'A string', 1024, 2, 4, 6, 16]

D:\Baldwin\AA-School\PyProg>
```

Figure 10

Lists are mutable

Unlike strings, the values in a list can be modified after the list is created.

The Python program shown in Figure 11 creates and prints a list. Then it uses a subscription to modify and print the list three times.

```
# File Lists03.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates mutating lists
#
#--------------------------------
print "Create and print a list"
listA = [3.14,59,"A string",
         1024]
print listA

print "Modify the list"
listA[2] = "New string"
print "Print the modified list"
print listA

print "Modify the list again"
listA[3] = listA[3] * 2
print "Print the modified list"
print listA

print "Modify the list again"
listA[2] = 0.99999
print "Print the modified list"
print listA
```

Figure 11

Replace a string with another string

The first modification replaces an existing string in the list with a new string.

Multiply an integer element by two

The second modification multiplies an integer value in the list by a factor of two.

Replace a string by a float

The third modification replaces a string in the list by a float value of 0.99999.

Program output

The output from the program is shown in Figure 12 (boldface added for emphasis).

```
D:\Baldwin\AA-School\PyProg>python Lists03.py
Create and print a list
[3.14, 59, 'A string', 1024]
Modify the list
Print the modified list
[3.14, 59, 'New string', 1024]
Modify the list again
Print the modified list
[3.14, 59, 'New string', 2048]
Modify the list again
Print the modified list
[3.14, 59, 0.99999, 2048]

D:\Baldwin\AA-School\PyProg>
```

Figure 12

More to come

There is a lot more for you to learn about lists that is not included in the above discussion. I will continue this discussion of Lists, including more sample programs, in a future lesson. Now it is time to review what we have learned so far.

Review

1. A *subscription* is how you go about getting magazines delivered to your mailbox, True or False?

Ans: False in the Python context. In Python, A *subscription* selects an item of a *sequence* object.

2. Name three types of *sequence* objects.

Ans: string, tuple and list

3. Given the following nomenclature

primary "[" expression_list "]"

what is the requirement for the *primary*?

Ans: The *primary* must evaluate to an object of a *sequence* or *mapping* type.

4. If the primary is a *sequence*, what must be the type of **expression_list**?

Ans: If the primary is a *sequence*, the **expression_list** must evaluate to a plain integer.

5. In question 4 above, the integer must be positive, True or False?

Ans: False. The integer may be positive or negative.

6. Which item in the sequence is selected for an index value of -1?

Ans: The last item in the sequence is selected for an index value of -1.

7. Just like C, C++, and Java, Python supports a *character* type, True or False?

Ans: False. There is no character type in Python. Rather, a string's items are characters. A character is not a separate data type but a string containing exactly one character.

8. What is another name for a *mapping*?

Ans: A dictionary.

9. What must be the type of the *primary* (see the above nomenclature) in order to support the use of *slicing*.

Ans: The primary must evaluate to a *sequence* object for the use of *slicing*.

10. What is a *mutable* sequence?

Ans: Mutable sequences can be changed after they are created.

11. A string is a mutable sequence, True or False?

Ans: False. The characters in a string cannot be modified after the string is created. There is currently a single mutable sequence type in Python, and it is a *list*.

12. What kinds of items can be placed in a *list*?

Ans: The items of a *list* are arbitrary Python objects.

13. How are *lists* formed?

Ans: *Lists* are formed by placing a comma-separated sequence of expressions in square brackets.

14. Show how to create a simple list using program code.

Ans: See Figure 13.

```
theList = [3.14,59,"A string",
           1024]
```
Figure 13

15. Show how to access an item in a list using a subscription.

Ans: See Figure 14.

```
print theList[2]
```
Figure 14

16. Show how to access a slice from a list.

Ans: See Figure 15.

```
print theList[0:3]
```
Figure 15

17. Show how to concatenate two lists.

Ans: See Figure 16.

```
listA = [3.14,59,"A string",
           1024]
listB = [2,4,6,16]

listC = listA + listB

```
Figure 16

18. Show how to modify an item in a list using a subscription.

Ans: See Figure 17.

```
listA = [3.14,59,"A string",
           1024]

listA[2] = "New string"
```

Figure 17

Lists, Part II

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

A previous lesson introduced you to lists.

Plus some other structures

It also introduced you to subscriptions, sequences, mutable sequences, mappings, slicings, and tuples.

Manipulating lists

That lesson showed you some of the ways that you can manipulate lists. The discussion was illustrated using sample programs.

Other ways to manipulate lists

This lesson carries that discussion forward by using sample programs to teach you other ways to manipulate lists.

Some Sample Programs

Replacing a slice

You can replace a slice in a list with the elements from another list through assignment.

Can change the length of the list

Note that this operation can change the length of the original list.

Replacing a slice in a list with the elements from another list is illustrated in Figure 1.

```
# File Lists04.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates replacing a slice
#
#--------------------------------
print "Create and print a list"
listA = [100,200,300,400,500]
print listA
print "Original length is:"
print len(listA)
print "Replace a slice"
listA[1:4] = [2,4,8,16,32,64]
print "Print the modified list"
print listA
print "Modified length is:"
print len(listA)

Figure 1
```

The function named len()

This program also illustrates the use of the function named **len()** to get and print the length of the list.

Replaces a three-element slice with a six-element list

In this program, a slice of an original five-element list, consisting of the elements from 1 through 3 inclusive, is replaced by the elements of a new list consisting of six new elements.

Length of the list is increased by 3

Since three existing elements are replaced by six new elements, the overall length of the list is increased by three elements.

Program output

The output from this program is shown in Figure 2 (boldface added for emphasis).

```
D:\Baldwin\AA-School\PyProg>python Lists04.py
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace a slice
Print the modified list
[100, 2, 4, 8, 16, 32, 64, 500]
Modified length is:
8

D:\Baldwin\AA-School\PyProg>
```

Figure 2

Six new elements replaced three original elements

As you can see, the six new elements replaced the three original elements to increase the length of the list from 5 to 8 elements.

Replacing an element with a list

It is also possible to replace an element in an existing list with a new list, as illustrated in the following program.

Behavior is different from above

In this case, the behavior is different from that shown above where a slice from the original list was replaced with the elements from a different list (even though the right operand of the assignment operator is the same in both cases).

Produces nested lists

When a single element is replaced by a list, the result is that *a new list is nested inside the original list*.

Length is unchanged

It is also interesting to note that the length of the list is unchanged by this operation since the list that replaces the element is itself considered to be a single element. Therefore, the number of elements is not changed.

Sample program

This is illustrated in Figure 3, where the element at index 2 of an original list is replaced with a new list having six elements.

```
# File Lists05.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates replacing an
#  element with a slice
#
#-------------------------------
print "Create and print a list"
listA = [100,200,300,400,500]
print listA
print "Original length is:"
print len(listA)
print "Replace an element"
listA[2] = [2,4,8,16,32,64]
print "Print the modified list"
print listA
print "Modified length is:"
print len(listA)
```

Figure 3

Program output

The output from this program is shown in Figure 4 (boldface added for emphasis).

```
D:\Baldwin\AA-School\PyProg>python Lists05.py
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5

D:\Baldwin\AA-School\PyProg>
```

Figure 4

One element is itself a list

As you can see, the result is that one of the elements in the original five-element list is replaced by a new list containing six elements. However, the length of the list is unchanged.

Again, it is important to note that this results in one list being nested inside of another list.

Extracting elements from a nested list

Now I am going to illustrate the syntax for extracting elements from a nested list using *pairs of matching square brackets*. Figure 5 is an expansion of Figure 3.

```
# File Lists06.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates extracting a
#  list element and extracting
#  elements from a nested list
#
#------------------------------
print "Create and print a list"
listA = [100,200,300,400,500]
print listA
print "Original length is:"
print len(listA)
print "Replace an element"
listA[2] = [2,4,8,16,32,64]
print "Print the modified list"
print listA
print "Modified length is:"
print len(listA)
print "Extract and display each"
print " element in the list"
print listA[0]
print listA[1]
print listA[2]
print listA[3]
print listA[4]

print "Extract and display each"
print " element in nested list"
print listA[2][0]
print listA[2][1]
print listA[2][2]
print listA[2][3]
print listA[2][4]
print listA[2][5]


Figure 5
```

Note the boldface statements

The most interesting statements in this program are highlighted in boldface.

Display the nested list combination

After nesting a list as element 2 in another list, the program displays the value of each of the elements of the original list. When element 2 is displayed, it can be seen to be another list.

Double-square-bracket notation

Then the program uses double-square-bracket notation (**listA[2][4]**) to extract and display each of the elements in the nested inner list that comprises element 2 of the outer list.

Program output

The output from this program is shown in Figure 6 (boldface added for emphasis).

```
D:\Baldwin\AA-School\PyProg>python Lists06.py
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
Extract and display each
 element in the list
100
200
[2, 4, 8, 16, 32, 64]
400
500
Extract and display each
 element in nested list
2
4
8
16
32
64

D:\Baldwin\AA-School\PyProg>
```

Figure 6

More on nested elements

The program in Figure 7 illustrates some additional aspects of nested elements.

```
# File Lists07.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates more nested
#  elements
#
#-------------------------------
print "Create and print a list\
 with 3 nested elements"
listA = [[2,4],[8,16,32],
         [64,128,256,512]]
print listA
print "Number of elements is:"
print len(listA)
print "Length of Element 0 is"
print len(listA[0])
print "Length of Element 1 is"
print len(listA[1])
print "Length of Element 2 is"
print len(listA[2])

Figure 7
```

Create a list of lists

This program defines a three-element list containing three nested lists.

In other words, each of the elements in the outer list is itself a list.

Inner lists are different lengths

Furthermore, the lengths of the inner nested lists are not the same. The lengths of the inner nested lists are 2, 3, and 4 elements each, respectively.

Program behavior and output

The output from the program is shown in Figure 8 (boldface added for emphasis). The program displays the entire list, and then gets and displays the lengths of each of the nested lists.

```
D:\Baldwin\AA-School\PyProg>python Lists07.py
Create and print a list with 3 nested elements
[[2, 4], [8, 16, 32], [64, 128, 256, 512]]
Number of elements is:
3
Length of Element 0 is
2
Length of Element 1 is
3
Length of Element 2 is
4
```

```
D:\Baldwin\AA-School\PyProg>


Figure 8
```

Getting the length of a nested list

Note in particular the syntax used to pass a parameter to the **len()** method in order to get the length of a nested list (**len(listA[1])**).

Arrays -- Not for beginners

If you are a beginning programmer, just skip this section.

If you are an experienced programmer, you may have observed that a Python lists bear a striking resemblance to arrays in other programming environments.

Python lists are more powerful

However, Python lists are more powerful than the arrays I am aware of in other programming environments, including Java.

Sub-arrays can be different sizes

For example as in Java, when a Python list is used to construct a multidimensional array, the sub arrays don't have to be of the same size.

Types can also be different

However, unlike Java, the elements in the array don't even have to be of the same type *(granted that the elements in a Java array can be of different types so long as there is an inheritance or Interface relationship among them).*

A three-dimensional array program

In any event, the program in Figure 9 might represent what an experienced programmer would consider to be a three-dimensional array of integer data in some other programming environment.

```
# File Lists08.py
# Rev 2/4/00
# Copyright 2000, R. G. Baldwin
# Illustrates a three-
#  dimensional array list
#
#--------------------------------
print "Create and print a\
```

```
 three-dimensional array list"
listA = [[[[1],[2]],[[3],[4]]],
          [[[5],[6]],[[7],[8]]]]
print listA
print "Print each element"
print listA[0][0][0]
print listA[0][0][1]
print listA[0][1][0]
print listA[0][1][1]
print listA[1][0][0]
print listA[1][0][1]
print listA[1][1][0]
print listA[1][1][1]
```

Figure 9

Triple-square-bracket notation

Pay particular attention to the triple square bracket notation that is used to access and print each element in the array.

Program behavior

This program

- Creates and populates the list that represents a three-dimensional array.
- Displays the entire array as a set of nested lists.
- Displays the contents of each element in the array.

Program output

The output from the program is shown in Figure 10.

```
D:\Baldwin\AA-School\PyProg>python Lists08.py
Create and print a three-dimensional array list
[[[[1], [2]], [[3], [4]]], [[[5], [6]], [[7], [8]]]]
Print each element
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]

D:\Baldwin\AA-School\PyProg>
```

Figure 10

The triple-square-bracket notation in the program of Figure 9 is essentially the same notation that would be used to access the individual elements in a three-dimensional array in C, C++, or Java.

More Information on Lists

There is quite a lot more that you will need to learn about lists. However, I will defer that discussion for a future lesson where I discuss the use of a list as a *data structure* or a *container*.

Review

1. Show how to replace a slice of a list with the elements from a new list.

Answer:

```
listA = [100,200,300,400,500]
listA[1:4] = [2,4,8,16,32,64]

Figure 11
```

2. Show how to replace an element in a list with a nested list.

Answer:

```
listA = [100,200,300,400,500]
listA[2] = [2,4,8,16,32,64]

Figure 12
```

3. Show how to extract the elements from one list that is nested in another list.

Answer:

```
listA = [100,200,300,400,500]
listA[2] = [2,4,8,16,32,64]
print listA[2][0]
print listA[2][1]
print listA[2][2]
print listA[2][3]
print listA[2][4]
print listA[2][5]

Figure 13
```

4. Show how to create a list of nested lists where the nested lists have different lengths.

Answer:

```
listA = [[2,4],[8,16,32],
         [64,128,256,512]]

Figure 14
```

Tuples, Index and Slice

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

Previous lessons have introduced you to lists, subscriptions, sequences, mutable sequences, mappings, slicings, and tuples.

Manipulating lists

Those lessons showed you some of the ways that you can manipulate lists. The discussion was illustrated using sample programs.

Now let's talk about tuples

The introduction to tuples in the previous lesson was very brief. This and several subsequent lessons use sample programs to show you a variety of ways to manipulate and use tuples.

What Is a Tuple?

As a practical matter, a tuple is like a list whose values cannot be modified. In other words, a tuple is *immutable*.

According to Lutz and Ascher, Learning Python from O'Reilly, tuples are *"Ordered collections of arbitrary objects."*

Can't be changed in place...

Again according to Lutz and Ascher, *"They work exactly like lists, except that tuples can't be changed in place (they're immutble)..."*

Parentheses replace square brackets

Unlike lists, however, tuples don't use square brackets for containment. Rather, they are normally written as a sequence of items contained in parentheses.

An immutable sequence

Like a string or a list, a tuple is a *sequence*. Like a string (but unlike a list), a tuple is an *immutable* sequence.

Tuples can be nested

Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

An array of references

One way to think of a tuple is to consider it to be an array of references to other objects.

While the tuple itself cannot be changed in place, the values contained in the objects to which it holds references can be changed (assuming that those objects are mutable).

What can you do with a tuple?

You can do just about anything with a tuple that you can do with a list, taking into account the fact that the tuple is *immutable*. Therefore, those list operations that change the value of a list in place cannot be performed on a tuple.

Accessed by index

As with strings and lists, items in a tuple are accessed using a numeric idex. The first item in a tuple is at index value 0.

Why do tuples exist?

Tuples provide some degree of integrity to the data stored in them. You can pass a tuple around through a program and be confident that its value can't be accidentally changed. *Note, however, that the values stored in the items referred to in a tuple can be changed (more about this later).*

In addition, in a future lesson, we will see some sample programs that require the use of tuples.

A Sample Program

Indexing and Slicing Tuples

Figure 1 shows the beginning of a Python script that first creates, and then manipulates a simple tuple (boldface added for emphasis). *The remainder of this program is shown as code fragments in subsequent figures. The entire program is shown near the end of the lesson.*

```
# File Tuple01.py
#------------------------------
print "Create a simple tuple"
aTuple = \
       (3.14,59,"A string",1024)
```

Figure 1

Let's see some output

At this point, I am going to show you the output produced by executing the Python script in Figure 1 so that you will have it available for reference during the discussion that follows.

The output is shown in Figure 2 with some lines highlighted using boldface for emphasis.

```
Create a simple tuple
Print index value 2
A string
Print a short slice
(3.14, 59, 'A string')
Print the entire tuple
(3.14, 59, 'A string', 1024)
```

Figure 2

Tuple syntax

From a syntax viewpoint, you create a tuple by placing a sequence of items inside a pair of enclosing parentheses and separating them by commas. *Note that the parentheses can be omitted when such omission will not lead to ambiguity.*

The fragment in Figure 1 creates a simple four-item tuple and assigns it to the variable named **aTuple**.

Different types allowed

Note that the items in a tuple can be different types. This simple tuple contains a float, an integer, a string, and another integer.

Could omit the parentheses

In this case, the parentheses could be omitted from the tuple syntax, because such omission would not lead to ambiguity. Figure 3 shows what this code fragment would look like if the parentheses were omitted. The tuple in Figure 3 was highlighted using boldface for emphasis.

```
print "Create a simple tuple"
aTuple = 3.14,59,"A string",1024
```
Figure 3

The scripts in Figure 1 and Figure 3 are operationally identical and produce the same output, as shown in Figure 2.

Indexing tuple items

The items in a tuple can be accessed using an index enclosed in square brackets as shown in Figure 4. (Learn to Program Using Python: Lists, Part II showed how to use an index in square brackets to access the items in a list.)

```
print "Print index value 2"
print aTuple[2]
```
Figure 4

Tuple item is printed

The third item in the tuple is accessed and printed in Figure 4. *(Remember, index values begin with the value 0, so index value 2 points to the third item in the tuple.)*

The output produced by the code in Figure 4 is the first boldface line in Figure 2, which reads "A string."

Tuples can be sliced

Tuples can be sliced just like lists (see Learn to Program Using Python: Lists, Part II .) This is illustrated by the code in Figure 5.

This code uses a slice to access and print the first three items in the tuple. *(Remember, a slice begins with the index shown by the first specified value and ends with the index whose value is one less than the second specified value.)*

```
print "Print a short slice"
print aTuple[0:3]
```
Figure 5

The output produced by the code in Figure 5 is shown by the second boldface line of output in Figure 2.

Print the entire tuple

Finally, the code fragment in Figure 6 causes the entire tuple to be accessed and printed, as shown by the third boldface line in Figure 2.

```
print "Print the entire tuple"
```

```
print aTuple[:100]

Figure 6
```

If you are unfamiliar with this slicing syntax, please see the material on slicing in Learn to Program using Python: Strings, Part II

Complete Program Listing

A complete listing of the program is shown in Figure 7.

```
# File Tuple01.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates indexing and
#  slicing a simple tuple
#
#-------------------------------
print "Create a simple tuple"
aTuple = \
       (3.14,59,"A string",1024)
print "Print index value 2"
print aTuple[2]
print "Print a short slice"
print aTuple[0:3]
print "Print the entire tuple"
print aTuple[:100]

Figure 7
```

What's Next?

There is a great deal more to be learned about manipulating tuples, and this will be the topic of future lessons.

Review

1. How does a tuple compare with a list?

Ans: A tuple is like a list whose values cannot be modified. In other words, a tuple is *immutable*.

2. True or false? A tuple is constructed by enclosing a series of comma-separated items with square brackets.

Ans: False. Square brackets are used for this purpose with lists. Parentheses are used with tuples.

3. Which if the following is true?

- A. A tuple is a mutable sequence.
- B. A tuple is an immutable sequence.

Ans: B. A tuple is an immutable sequence.

4. True or false? Tuples cannot be nested.

Ans: False. Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

5. True or false? The values contained in the objects to which a tuple holds references can be changed (assuming that those objects are mutable).

Ans: True

6. True or false? Those list operations that change the value of a list in place can also be performed on a tuple.

Ans: False. Because a tuple is immutable, operations that would change its value in place are not available.

7. True or false? The items in a tuple are accessed by key values like a dictionary.

Ans: False. Items in a tuple are accessed by numeric index.

8. Write a Python script that shows how to create and print a simple tuple.

Ans: See Figure 8.

```
# File Tuple08.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates creating and
#  printing a simple tuple
#-------------------------------
t1 = "a",1,"c"
print t1
```

Figure 8

9. Write a Python script that shows how to access and print an item in a tuple using a numeric index.

Ans: See Figure 9.

```
# File Tuple09.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates accessing a tuple
#  item with an index
#-------------------------------
t1 = "a",1,"c"
print t1[1]
```

Figure 9

10. Write a Python script that shows how to access a slice of a tuple and print it.

Ans: See Figure 10.

```
# File Tuple10.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates accessing a slice
#  of items from a tuple
#-------------------------------
t1 = "a","b","c", "d", "e"
print t1[1:4]
```

Figure 10

11. True or false? All of the items in a tuple must be of the same type.

Ans: False. A tuple can contain items of mixed types.

12. True or false? The enclosing parentheses must always be used to enclose the items in a tuple.

Ans: False. The parentheses can be omitted when this will not lead to ambiguity.

Nested Tuples

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different code fragments while you are reading about them.

Introduction

Previous lessons have introduced you to lists, subscriptions, sequences, mutable sequences, mappings, slicings, and tuples.

Let's talk some more about tuples

The previous lesson (see Learn to Program using Python: Tuples, Index and Slice) showed you

- How to create a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.

This lesson will expand your knowledge of tuples by teaching you about nesting tuples within other tuples.

What Is a Tuple?

To briefly repeat part of what you learned in Learn to Program using Python: Tuples, Index and Slice , a tuple is like a list whose values cannot be modified. In other words, a tuple is *immutable*.

- Tuples are normally written as a sequence of items contained in matching parentheses.
- A tuple is an *immutable* sequence.
- Items in a tuple are accessed using a numeric index.

Tuples can be nested

Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested inside of other tuples.

Sample Program

Nesting tuples

Listing 1 shows the beginning of a Python script that

- Creates two tuples.
- Nests them in a third tuple.
- Determines the length (number of items) in the tuple containing the two nested tuples.

```
# File Tuple02.py
#------------------------------
print "Create/print one tuple"
t1 = 1,2
print t1
print "Create/print another \
tuple"
t2 = "a","b"
print t2
```

Listing 1

(The boldface in Listing 1 was added for emphasis.)

The remaining parts of this program are shown as code fragments in subsequent listings. A listing of the entire program is shown in Listing 7 *near the end of the lesson.*

Let's look at some output

Listing 2 shows the output produced by the code fragment in Listing 1.

```
Create/print one tuple
(1, 2)
Create/print another tuple
('a', 'b')
```

Listing 2

Some of the lines in the output were highlighted in boldface for emphasis. The two lines highlighted in boldface in Listing 2 show the two tuples that were produced and printed by the code in Listing 1.

Now let's nest them

Listing 3 shows code that nests the two tuples, **t1** and **t2**, produced earlier, along with two strings, in a new tuple. The new tuple is assigned to the variable named **t3**.

```
print "Create/print nested \
tuple"
t3 = "A",t1,"B",t2
```

```
print t3
```

Listing 3

Nesting is easy

All that is required to nest the existing tuples in a new tuple is to list the variables representing the two existing tuples in a comma-separated list of items for creation of the new tuple.

What does the new tuple look like?

Listing 4 shows the printed output for the new tuple containing two nested tuples.

```
Create/print nested tuple
('A', (1, 2), 'B', ('a', 'b'))
```

Listing 4

Nested tuples retain their identity

Note that the two nested tuples retain their identity as tuples, as indicated by the fact that the parentheses surrounding the items in the two nested tuples are preserved in the new tuple.

Let's get the length of the new tuple

Listing 5 shows code that gets and displays the length of the new tuple, which contains the two nested tuples.

```
print "Length of nested \
tuple is:"
print len(t3)
```

Listing 5

What is the length?

The length is a measure of the number of items in the tuple, and is obtained using the method named **len()**.

The length is only four items

Listing 6 shows the output produced by the code in Listing 5, including the length of the new tuple containing the two nested tuples.

```
Length of nested tuple is:
4

Listing 6
```

The important point...

The important point is that even though the tuple shown in Listing 4 actually consists of six individual items (ignoring parentheses), each of the nested tuples is treated as a single item, giving a length of only four items for the tuple that contains the two nested tuples.

This would be true regardless of the length of the nested tuples.

Accessing an item in a nested tuple

Later, we will see that a double square-bracket indexing notation can be used to gain access to the individual items in tuples that are nested inside of other tuples.

Listing of Sample Program

A complete listing of the program is shown in Listing 7.

```
# File Tuple02.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates creating and
#  displaying nested tuples
#
#--------------------------------
print "Create/print one tuple"
t1 = 1,2
print t1
print "Create/print another \
tuple"
t2 = "a","b"
print t2
print "Create/print nested \
tuple"
t3 = "A",t1,"B",t2
print t3
print "Length of nested \
tuple is:"
print len(t3)

Listing 7
```

What's Next?

There is more to be learned about tuples. The next lesson will show you how to create and use empty tuples along with tuples containing only one item.

Review

1. True or false? Tuples can be nested inside other tuples.

Ans: True.

2. Write a Python script that illustrates how to nest two or more tuples inside another tuple.

Ans: See Listing 8.

```
# File Tuple11.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates tuple nesting
#
#--------------------------------
t1 = "a",("b","c"),"d",("e","f")
print t1
print len(t1)

Listing 8
```

3. Write a Python script that illustrates how to determine the length of a tuple.

Ans: See Listing 8.

4. True or false? All that is required to nest existing tuples into a new tuple is to list the variables representing the existing tuples in a comma-separated list of items for creation of the new tuple.

Ans: True.

5. True or false? When you nest one or more tuples inside another tuple, the items in the nested tuples are melted into the new tuple. That is to say, the length of the new tuple is the sum of the lengths of the nested tuples plus the non-tuple items in the new tuple.

Ans: False. Nested tuples retain their identity as tuples, and each nested tuple counts as only one item in the new tuple, regardless of the lengths of the nested tuples.

6. How do you access the individual items inside a tuple that is nested inside another tuple?

Ans: There is a double square-bracket indexing notation that can be used for this purpose.

Empty and Single-Item Tuples

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different code fragments, without losing your place, while you are reading about them.

Introduction

This is the third in a series of lessons designed to teach you about tuples.

Previous lessons (see Learn to Program using Python: Nested Tuples) have illustrated

- How to create a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.

This lesson will teach you how to create empty tuples and tuples containing only one item.

What Is a Tuple?

A tuple is like a list whose values cannot be modified. It is an ordered list of objects, and it can contain references to any type of object.

- Tuples are normally written as a sequence of items contained in matching parentheses.
- A tuple is an *immutable* sequence.
- Items in a tuple are accessed using a numeric index.
- Tuples can be nested.

Sample Program

Empty and single-item tuples

Listing 1 shows the beginning of a Python script that

- Creates an empty tuple.
- Creates a single-item tuple.
- Nests the two tuples along with some strings in a third tuple.
- Determines the length of each of the tuples.
- Displays all of the above

```
# File Tuple03.py
#------------------------------
print "Create/print empty \
tuple"
t1 = ()
print t1
print "Length of empty tuple is"
print len(t1)

Listing 1
```

(Note that some of the text in the Listings was highlighted using boldface for emphasis.)

The remaining parts of this program are shown as code fragments in subsequent Listings. A listing of the entire program is shown in Listing 7 near the end of the lesson.

What is an empty tuple?

As you might have guessed from the name, an empty tuple is just a pair of empty parentheses as shown by the first boldface line in Listing 1.

Listing 2 shows the output produced by the code in Listing 1. The empty tuple is displayed simply as a pair of empty parentheses, and the length of the empty tuple

is shown to be zero (0).

```
Create/print empty tuple
()
Length of empty tuple is
0

Listing 2
```

Are you surprised?

There are probably no surprises regarding an empty tuple. However, there may be some surprises in the code fragment shown in Listing 3. This fragment deals with a tuple containing only one element.

```
print "Create/print one-\
element tuple"
# Note the req trailing comma
t2 = "a",
print t2

print "Length of one-element \
tuple is:"
print len(t2)

Listing 3
```

Ugly syntax

The syntax for creating a tuple with only one element is rather ugly, but is required to avoid ambiguity. In particular, it is necessary to follow the single tuple item with a comma as shown in the first boldface line in Listing 3.

Why is this comma necessary?

Had I written that line simply as follows without the extra comma,

```
t2 = "a"
```

the result would have been to create a new variable named **t2** whose contents would be the string **"a"**.

Not a tuple

This would not indicate a tuple at all. Thus, the extra comma is required to make a single-item tuple unique and to distinguish it from other possibilities.

Output for the single-item tuple

Listing 4 shows the output produced by the code in Listing 3. The single-item tuple is shown in the first boldface line. As is always the case, the tuple is displayed in parentheses.

```
Create/print one-element tuple
('a',)
Length of one-element tuple is:
1

Listing 4
```

What is the length of the tuple?

There is no surprise here. The length of the tuple as shown in Listing 4 is one (1) item.

Nested tuples

Just to give you a little more practice in dealing with nested tuples, the code in Listing 5 nests the two tuples created above into a new tuple and stores the new tuple in the variable named **t3**.

```
print "Create/print nested \
tuple"
t3 = "A",t1,"B",(t2,"Z"), "C"
print t3

print "Length of nested tuple \
is"
print len(t3)

Listing 5
```

Doubly-nested tuples

However unlike previous sample programs, in this case, literal parentheses are used to cause the tuple named **t2** to be *doubly nested*.

In particular, as shown by the first boldface portion of code in Listing 5, the tuple named **t2** and the string **"Z"** are used to create a tuple, which in turn, is nested in the tuple assigned to the variable named **t3**.

What does a doubly-nested tuple look like?

The double nesting is evidenced by the extra parentheses in the boldface portion of the output shown in Listing 6.

```
Create/print nested tuple
```

```
('A', (), 'B', (('a',), 'Z'), 'C')
Length of nested tuple is
5

Listing 6
```

What is the length of this tuple?

The length of the tuple is also shown in Listing 6.

As you may have determined already, even though the tuple named **t3** contains two nested tuples (one of which is doubly-nested), its overall length is only five (5) items.

Note that even though one of the tuples nested inside of **t3** has a length of zero, it counts as one item when the length of **t3**, is determined.

Listing of Sample Program

A complete listing of the program is shown in Listing 7.

```
# File Tuple03.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates empty tuples and
#  tuples with only one element
#
#------------------------------
print "Create/print empty \
tuple"
t1 = ()
print t1
print "Length of empty tuple is"
print len(t1)

print "Create/print one-\
element tuple"
# Note the req trailing comma
t2 = "a",
print t2
print "Length of one-element \
tuple is:"
print len(t2)


print "Create/print nested \
tuple"
t3 = "A",t1,"B",(t2,"Z"), "C"
print t3

print "Length of nested tuple \
is"
print len(t3)
```

```
Listing 7
```

What's Next?

Upcoming lessons will show you how to:

- Unpack a tuple.
- Index inside of nested tuples.
- Slice nested tuples.
- Modify values stored in an object referred to by an item in a tuple.

Review

1. True or false? A tuple is an unordered list of objects.

Ans: False. A tuple is an ordered list of objects. This is evidenced by the fact that the objects can be accessed through the use of an ordinal index.

2. True or false? A tuple can only store references to other tuples.

Ans: False. A tuple can store references to any type of object.

3. True or false? All of the objects referred to by the items in a tuple must be of the same type.

Ans: False. The references stored as items in a tuple can refer to different types of objects.

4. True of false? A tuple is a mutable sequence.

Ans: False. A tuple is an immutable sequence.

5. True or false? The items in a tuple are accessed using a key, as in looking things up in a dictionary.

Ans: False. Items in a tuple are accessed using a numeric index that begins with the value zero (0).

6. Write a Python script that creates and displays an empty tuple.

Ans: See Listing 8.

```
# File Tuple13.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates empty tuples
#
#--------------------------------
print "Create/print empty \
```

```
tuple"
t1 = ()
print t1

Listing 8
```

7. Write a Python script that creates and displays a tuple having only one item.

Ans: See Listing 9.

```
# Copyright 2000, R. G. Baldwin
# Illustrates tuples with only
#  one element
#-------------------------------
print "Create/print one-\
element tuple"
# Note the req trailing comma
t2 = "a",
print t2

Listing 9
```

8. Write a Python script that creates and displays a tuple that is nested at least three levels deep.

Ans: See Listing 10.

```
# Copyright 2000, R. G. Baldwin
# Illustrates deeply-nested
#  tuple
#-------------------------------
t1 = "a",
print t1
t2 = t1,"b","c"
print t2
t3 = t2,"d"
print t3
t4 = t3,"e"
print t4

Listing 10
```

9. True or false? An empty tuple is created by placing a single comma inside of a pair of parentheses.

Ans: False. An empty tuple is just a pair of empty parentheses.

10. True or false? The **len()** method reports the length of an empty tuple as 1.

Ans: False**.** The **len()** method reports the length of an empty tuple as 0.

11. True or false? The syntax for a tuple with a single item is simply the element enclosed in a pair of matching parentheses as shown below:

```
t = ("a")
```

Ans: False. The syntax for a tuple with a single item requires the item to be followed by a comma as shown below:

```
t = ("a",)
```

12. True or false? The **len()** method reports the length of a tuple containing one item as 1.

Ans: True.

13. What is the length of the tuple shown below?

((((('a', 1), 'b', 'c'), 'd', 2), 'e', 3)

Ans: The length of this tuple is 3. See Listing 11 for confirmation.

```
# File Tuple15.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates length of deeply-
#  nested tuple
#--------------------------------
t1 = "a",1
t2 = t1,"b","c"
t3 = t2,"d",2
t4 = t3,"e",3
print t4
print len(t4)
```

Listing 11

Learn to Program using Python: Unpacking Tuples

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language. The ultimate objective is to progress to JPython, forming the link between Python and Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings, without losing your place, while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started , provides an overall description of this online programming course.

Introduction

This is another lesson in a series of lessons designed to teach you about tuples.

Previous lessons (see Learn to Program using Python: Nested Tuples) have illustrated

- How to create (pack) a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.
- How to create empty tuples.
- How to create single-item tuples.

Preview

This lesson will teach you about unpacking tuples.

What Is a Tuple?

A tuple is an immutable ordered list of objects. It can contain references to any type of object. See Learn to Program Using Python: Empty and Single-Item Tuples for a more detailed description.

Sample Program

Unpacking a tuple

Listing 1 shows the beginning of a Python script that:

- Creates (packs) two simple tuples.
- Creates (packs) a third tuple by concatenating the first two tuples.
- Displays the third tuple.
- Unpacks the third tuple, assigning each item of the tuple into a separate variable.
- Displays each of the variables.
- Creates and displays a mutable list object containing five strings.
- Unpacks the tuple created earlier assigning the four items from the tuple into the first four items in the list.
- Displays the list.

```
# File Tuple04.py
#------------------------------
# Create a pair of tuples
t1 = 1,2
t2 = "A","B"

# Concatenate and print them
t3 = t1 + t2
print t3
```

Listing 1

(Note that some of the text in the listings was highlighted using boldface for emphasis.)

The remaining parts of this program are shown as code fragments in subsequent listings. A listing of the entire program is shown in Listing 7 near the end of the lesson.

Tuples can be concatenated

As shown in Listing 1, tuples support the concatenation (+) operator. You can concatenate two or more tuples to produce a new tuple. This program creates two simple tuples, and then concatenates them to create a third tuple.

The output

Listing 2 shows the output produced by the code in Listing 1. By now, the creation and display of simple tuples should be very familiar to you based on earlier lessons (see Learn to Program using Python: Tuples, Index and Slice) . Therefore, I won't discuss this part of the program further.

```
(1, 2, 'A', 'B')
```

Listing 2

Now for something strange...

The code in Listing 3 is not quite so straightforward. In fact, it looks rather strange if you come from a conventional C, C++, or Java programming background.

```
# Unpack the tuple and print
# individual elements
w,x,y,z = t3
print w
print x
print y
print z
```

Listing 3

What do Lutz and Ascher have to say?

Here is what Lutz and Ascher, authors of Learning Python, from O'Reilly, have to say about this syntax. They refer to this as *Tuple assignment (positional)*.

"When you use tuples or lists on the left side of the =, Python pairs objects on the right side with targets on the left and assigns them from left to right."

What does Guido van Rossom have to say?

The Python Tutorial by Guido van Rossum refers to the boldface statement in Listing 3 as *tuple unpacking*.

Guido van Rossom points out

"Tuple unpacking requires that the list of variables on the left has the same number of elements as the length of the tuple."

Otherwise, an error occurs

If you try to run a script that doesn't meet these criteria, you will get the following error:

ValueError: unpack tuple of wrong size

What is *tuple packing?*

Interestingly, Guido van Rossum refers to the creation of a tuple (see the three lines of code used to create tuples in Listing 1) as *tuple packing*.

Some more output

Listing 4 shows the output produced by the code in Listing 3.

```
1
2
A
B

Listing 4
```

If you compare this output with the original tuple in Listing 1, or with the previous output in Listing 2, you will see that each of the individual items in the tuple (in left-to-right order) were assigned respectively to the variables named **w**, **x**, **y**, and **z**.

Thus, the lines of output produced by printing these four variables in Listing 4 match the items in the original tuple that was created in Listing 1 and displayed in Listing 2.

A mutable list

Just to make things a little more interesting, I decided to combine the use of a tuple (an immutable list) and a regular mutable list in this program.

Listing 5 contains the code to create a mutable list populated with five string characters.

```
# Create and print a list
L1 = ["a","b","c","d","e"]
```

```
print L1

# Unpack tuple into the list
# and print it
L1[0],L1[1],L1[2],L1[3] = t3
print L1
```

Listing 5

Then the list is displayed, as shown in Listing 6. The first line of output in Listing 6 shows the contents of the list just after it is created and populated.

```
['a', 'b', 'c', 'd', 'e']
[1, 2, 'A', 'B', 'e']
```

Listing 6

Unpack the tuple into the mutable list

Then, as shown in Listing 5 the tuple from above is unpacked with the individual items being assigned to the first four items in the list *(remember, a list is mutable, so the values of its items can be changed)*.

Look at the list again

Then the contents of the list are displayed again. As you can see from Listing 6, the first four items in the list were replaced by the four items from the tuple. The fifth item in the list was not modified.

The real difference...

So, what is the real difference between a tuple (an immutable list) and an ordinary mutable list as used in this sample program?

An experiment

Try the following experiment. There is a line in the program (shown in Listing 7 near the end of this lesson) that reads as follows:

L1 = ["a","b","c","d","e"]

Modify the program, changing this line so that it reads as follows:

L1 = ("a","b","c","d","e")

Converting a list into a tuple

I hope you recognize that by replacing the square brackets with parentheses, you have changed **L1** from an ordinary mutable list to a tuple (an immutable list).

Execute the script

Now execute the script. Your output should look something like that shown in Listing 8.

```
(1, 2, 'A', 'B')
1
2
A
B
('a', 'b', 'c', 'd', 'e')
Traceback (innermost last):
  File "junk.py", line 28, in ?
    L1[0],L1[1],L1[2],L1[3] = t3
TypeError: object doesn't support
item assignment
```

Listing 8

Everything should work well until the attempt is made to unpack the tuple named **t3** and to assign the items from that tuple into the individual items of the new tuple named **L1**.

Game over

The items in a tuple are immutable, meaning that they cannot be changed. Therefore, the program crashes at this point with the following error message:

```
TypeError: object doesn't support
item assignment
```

Summary

Among other things, this lesson has illustrated:

- Tuple concatenation.
- Tuple unpacking (or tuple assignment).
- Tuple packing.

What's Next?

Upcoming lessons will show you how to:

- Index inside of nested tuples.
- Slice nested tuples.
- Modify values stored in an object referred to by an item in a tuple.

Review

1. True of false? Tuples support the use of the concatenation operator.

Ans: True.

2. Write a Python script that illustrates unpacking a tuple.

Ans: See Listing 9**.**

```
# File Tuple16.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates unpacking a tuple
#
#------------------------------
w,x,y,z = (1,2,3,4)
print w
print x
print y
print z
```

Listing 9

3. True or false? The following is valid Python code.

```
v,w,x,y,z = t
```

Ans: True. The above is valid Python code if **t** is a tuple, a list, or a string whose length matches the number of variables on the left of the assignment operator.

4. Write a Python script that illustrates tuple packing.

Ans: See Listing 10.

```
# File Tuple17.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates packing a tuple
#
#------------------------------
t1 = 1,2,3,4
print t1
```

Listing 10

5. Write a Python script that illustrates tuple concatenation.

Ans: See Listing 11.

```
# File Tuple18.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates concatenating
#  tuples
#-------------------------------
t1 = (1,2) + (3,4)
print t1
```

Listing 11

6. Write a Python script that illustrates how to unpack a tuple into a list.

Ans: See Listing 12.

```
# File Tuple19.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates unpacking a tuple
#  into a list
#-------------------------------
L1 =["a","b","c"]
L1[0],L1[1],L1[2] = (0,1,2)
print L1
```

Listing 12

Listing of Sample Program

A complete listing of the program is shown in Listing 7.

```
# File Tuple04.py
# Rev 7/31/00
# Copyright 2000, R. G. Baldwin
# Illustrates unpacking a tuple
#
#-------------------------------
# Create a pair of tuples
t1 = 1,2
t2 = "A","B"

# Concatenate and print them
t3 = t1 + t2
print t3

# Unpack the tuple and print
# individual elements
w,x,y,z = t3
print w
print x
print y
print z
```

```
# Create and print a list
L1 = ["a","b","c","d","e"]
print L1

# Unpack tuple into the list
# and print it
L1[0],L1[1],L1[2],L1[3] = t3
print L1
```

Listing 7

Learn to Program Using Python: Indexing Nested Tuples

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language. The ultimate objective is to progress to JPython, forming the link between Python and Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

Introduction

This is one in a series of lessons designed to teach you about tuples.

Previous lessons (see Learn to Program using Python: Unpacking Tuples) have illustrated

- How to create (pack) a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.
- How to create empty tuples.
- How to create single-item tuples.
- How to unpack a tuple

Preview

The primary purpose of this lesson is to teach you how to use multiple square-bracket notation for indexing nested tuples.

What Is a Tuple?

A tuple is an immutable ordered list of objects. It can contain references to any type of object. See Learn to Program using Python: Nested Tuples for a more detailed description.

Sample Program

Indexing a nested tuple

Listing 1 shows the beginning of a Python script that:

- Packs a two-item tuple by applying the *repeat* operator to a single-item tuple.
- Displays information about the two-item tuple by using the *membership* operation in the conditional clause of an **if** statement.
- Packs a deeply-nested tuple by successively nesting tuples in new tuples.
- Uses multiple square-bracket index notation to successively access and display information about the nested tuples.

```
# File Tuple05.py
#------------------------------
t1 = "a",
t2 = t1*2

Listing 1
```

(Note that some of the text in the listings was highlighted using boldface for emphasis.)

The remaining parts of this program are shown as code fragments in subsequent listings. A listing of the entire program is shown in Listing 15 near the end of the lesson.

Tuples support the *repeat* operator

As shown in Listing 1, tuples support the *repeat* (*) operator.

The *repeat* operator behaves like a form of concatenation. You can cause a tuple to be concatenated onto itself a specified number of times using the syntax shown in the boldface line in Listing 1.

The operands

The left operand of the ***** operator specifies the tuple to be concatenated onto itself. The integer right operand specifies the number of copies of the tuple that are to be concatenated together.

The tuple that resulted from performing this operation is shown displayed in boldface in Listing 3 later in the lesson.

Tuples support the *membership* operation

Tuples also support the *membership* (in) operation, as illustrated in boldface in Listing 2.

```
if ("b" in t2):
  print "OK"
if ("a" in t2):
  print "t2"
  print t2
  print "length = ",len(t2)

Listing 2
```

The *membership* operation is used twice in the code fragment shown in Listing 2. In both cases, the membership operation is used in the conditional expression of an **if** statement.

The *false* case

In the first case, a test is made to determine if the tuple referred to by **t2** contains the string **"b"**. If *true*, the string **"OK"** would be printed.

As it turns out, that string is not contained in the tuple, so the operation returns *false* and the "OK" string is not printed, as evidenced in Listing 3.

```
t2
('a', 'a')
length =  2
```

Listing 3

The *true* case

In the second case in Listing 2, a test is made to determine if the tuple contains the string **"a"**. If *true*, the tuple and its length are printed.

As you can see from Listing 3, this test returns *true*. As a result, the name of the tuple, the tuple, and the length of the tuple are displayed. *(The tuple is displayed in boldface for emphasis.)*

Pack the deeply-nested tuple

The code in Listing 4 packs the deeply-nested tuple by successively nesting the existing tuple into a new tuple.

```
t3 = 1, t2, 2
t4 = 3, t3, 4
t5 = 5, t4, 6
```

Listing 4

This code results in a tuple that is nested several levels deep as shown in Listing 6 later in the lesson.

Now print the deeply-nested tuple

As shown in Listing 4, the deeply-nested tuple is named **t5**.

The code in Listing 5 causes the name of the tuple, the tuple itself, and the length of the tuple to be displayed on the screen.

```
# print entire tuple
print "t5"
print t5
print "length = ",len(t5)
```

```
Listing 5
```

The output is shown in Listing 6 (with color added for clarity).

```
t5
(5, (3, (1, ('a', 'a'), 2), 4), 6)
length =  3

Listing 6
```

What is the length of the tuple?

It is very important to understand that even though you can count the following eight separate *things* in the tuple,

5 3 1 'a' 'a' 2 4 6

its length is only three (3), meaning that it only contains three items.

What are the available index values?

The three items contained in the tuple can be accessed using index values of 0, 1, and 2.

Everything that is highlighted in red in Listing 6 is a single item, which can be accessed using the index value of 1.

That item is a nested tuple, which also contains other nested tuples.

Using single square-bracket index notation

I discussed the use of single square-bracket indexing of tuples in a previous lesson.

The code in Listing 7 uses square-bracket index notation twice to access the item whose index value is 1. Both instances are highlighted in boldface.

```
# now print using index values
print "t5[1]"
print t5[1]
print "length = ",len(t5[1])

Listing 7
```

In the first instance, the item is displayed on the screen.

In the second instance, the length of the item is determined and the length is displayed on the screen.

What does this code produce?

The output produced by this code is shown in Listing 8.

```
t5[1]
(3, (1, ('a', 'a'), 2), 4)
length =  3

Listing 8
```

The code in Listing 7 accesses and displays the tuple, which is nested at index value 1 in the tuple named **t5**.

What is the length of the tuple?

Again, note that the reported length of this tuple is 3 (see Listing 8), even though you can count six different *things* in the tuple.

As before, I used red to highlight the item at index value 1 in Listing 8.

The highlighted material is another tuple nested in this tuple. The nested tuple contains even another nested tuple.

Using multiple square-bracket index notation

Now we have arrived at the primary purpose of this lesson, which is to teach you how to use multiple square-bracket notation for indexing nested tuples.

The code in Listing 9 illustrates how this is done. I used color to highlight two instances in the code where multiple square-bracket indexing is used.

```
print "t5[1][1]"
print t5[1][1]
print "length = ",len(t5[1][1])

Listing 9
```

The first instance reads as follows:

```
print t5[1][1]
```

What does this mean?

You might interpret the above statement as follows:

Step 1
First extract the item at index value 1 from the tuple named **t5**. This is indicated by the green portion of the above statement.

Step 2
Having extracted that item (which in this case is another nested tuple) extract the item from that tuple whose index value is also 1. This is indicated by the violet portion of the above code.

Step 3
Having extracted that item (which in this case is another nested tuple), print it on the screen.

Play it again Sam!

A similar indexing operation is used in Listing 9 to determine the length of the extracted tuple (shown below with color added as above).

```
len(t5[1][1])
```

What is the length?

The output produced by the code in Listing 9 is shown in Listing 10. As you can see, the length is 3 items.

```
t5[1][1]
(1, ('a', 'a'), 2)
length =  3
```

Listing 10

Just another tuple

As mentioned above, the extracted item shown in Listing 10 is another tuple with a length of 3.

The item at index value 1 of the extracted tuple, which I have highlighted using boldface in Listing 10, is another nested tuple.

Triple square brackets

That tuple can be extracted using three sets of square brackets as illustrated by the code in Listing 11.

```
print "t5[1][1][1]"
print t5[1][1][1]
print "length = ",\
               len(t5[1][1][1])
```

Listing 11

And the output is...

Listing 12 shows the output produced by the code in Listing 11.

```
t5[1][1][1]
('a', 'a')
length =  2

Listing 12
```

In this case, the extracted item is another tuple, with a length of 2.

Not just another pretty tuple

However, it doesn't contain another nested tuple. Instead, it contains two strings.

Either of the items in this extracted tuple can be accessed using four sets of square brackets as illustrated by the code in Listing 13.

```
print "t5[1][1][1][1]"
print t5[1][1][1][1]
print "length = ",\
             len(t5[1][1][1][1])

Listing 13
```

The output produced by the code in Listing 13 is shown in Listing 14.

```
t5[1][1][1][1]
a
length =  1

Listing 14
```

Finally, the end of the nesting

At this point, we have worked our way down inside the nested structure to extract an item from the innermost nested tuple.

In this case, the item that we extract is not a tuple. Rather, it is a string with a length of 1 and a value of "a" as shown in Listing 14.

Does it look familiar?

This is the value that was packed into the original tuple in Listing 1 before the nesting operation began.

Summary

The primary purpose of this lesson has been to teach you how to use multiple square-bracket notation for indexing nested tuples. In addition, I have illustrated several other concepts including:

- The repeat operator.
- The membership operation.
- Packing a deeply-nested tuple.
- Printing a deeply-nested tuple.
- Obtaining the length of a nested tuple.

What's Next?

Upcoming lessons will show you how to:

- Slice nested tuples.
- Modify values stored in an object referred to by an item in a tuple.

Review

1. Write a Python script that illustrates the use of the *repeat* operator with tuples.

Ans: See Listing 16.

```
# File Tuple20.py
# Rev 08/02/00
# Copyright 2000, R. G. Baldwin
# Illustrates the repeat
#  operator with tuples
#-------------------------------
t1 = "a",
t2 = t1*2
print t2

Listing 16
```

2. Write a Python script that illustrates the use of the *membership* operator with tuples.

Ans: See Listing 17.

```
# File Tuple21.py
# Rev 08/02/00
# Copyright 2000, R. G. Baldwin
# Illustrates the membership
#  operator with tuples
#-------------------------------
t1 = "a",
t2 = t1*2
print t2

if("a" in t2):
```

```
  print "Contains a"
if("b" in t2):
  print "Contains b"


Listing 17
```

3. Write a Python script that illustrates the use of multiple square-bracket indexing to access and display an item in a nested tuple.

Ans: See Listing 18.

```
# File Tuple22.py
# Rev 08/02/00
# Copyright 2000, R. G. Baldwin
# Illustrates the use of
#  multiple square-bracket
#  indexing
#-------------------------------
t1 =(1,(4,(7,8,9),6),3)
print t1[1][1][2]


Listing 18
```

4. What is the length of the following tuple?

```
(1,(4,(7,8,9),6),3,10,11)
```

Ans: The length is 5. See Listing 19.

```
# File Tuple23.py
# Rev 08/02/00
# Copyright 2000, R. G. Baldwin
# Illustrates the length of a
#  nested tuple
#-------------------------------
t1 =(1,(4,(7,8,9),6),3,10,11)
print len(t1)


Listing 19
```

Listing of Sample Program

A complete listing of the program is shown in Listing 15.

```
# File Tuple05.py
# Rev 08/02/00
# Copyright 2000, R. G. Baldwin
```

```
# Illustrates indexing nested
#  tuples
#
#-------------------------------
t1 = "a",
t2 = t1*2
if ("b" in t2)
  print "OK"
if ("a" in t2):
  print "t2"
  print t2
  print "length = ",len(t2)

t3 = 1,t2,2
t4 = 3,t3,4
t5 = 5,t4,6

# print entire tuple
print "t5"
print t5
print "length = ",len(t5)

# now print using index values
print "t5[1]"
print t5[1]
print "length = ",len(t5[1])

print "t5[1][1]"
print t5[1][1]
print "length = ",len(t5[1][1])

print "t5[1][1][1]"
print t5[1][1][1]
print "length = ",\
                len(t5[1][1][1])

print "t5[1][1][1][1]"
print t5[1][1][1][1]
print "length = ",\
             len(t5[1][1][1][1])
```

Listing 15

Learn to Program Using Python: Slicing Nested Tuples

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language. After we cover regular Python, the lessons will cover JPython. This will form a link between Python and Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings, without losing your place, while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course. (You will find a consolidated index to all of my Python, Java, and XML tutorials on my website.)

Introduction

This is one in a series of lessons designed to teach you about tuples.

Previous lessons have illustrated

- How to create (pack) a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.
- How to create empty tuples.
- How to create single-item tuples.
- How to unpack a tuple
- How to use a numeric index to access the items in nested tuples.

Preview

This lesson will teach you how to combine indexing and slicing to access groups of items in nested tuples.

Looks a lot like list processing to me...

By the way, in case you haven't figured it out before now, almost everything that I have been teaching you about tuples can also be applied to lists.

What Is a Tuple?

A tuple is an immutable ordered list of objects. It can contain references to any type of object. See previous lessons in this series for a more detailed description.

Sample Program

Slicing a nested tuple

Listing 1 shows the beginning of a Python script that:

- Packs a multiply-nested tuple by successively nesting tuples in new tuples.
- Gets and displays a slice of the top-level tuple.
- Uses a combination of indexing and slicing to access "sub-tuples" that are nested at different levels in the top-level tuple.
- Slices and displays groups of items in the sub-tuples.

```
# File Tuple06.py
#------------------------------
# Create a simple tuple
t1 = 1,2,3

# Create nested tuple
t3 = t1,"B"
t4 = "X",t3,"Y","Z"
print "A nested tuple"
print t4

Listing 1
```

The remaining parts of this program are shown as code fragments in subsequent figures. A listing of the entire program is shown in Listing 9.

Tuples support slicing

As shown in Listing 1, tuples support the **[x:y]** slicing operation.

You may recall that the slicing operation returns a slice from the sequence beginning at index value **x** and including all items up to, but not including, the item at index value **y**.

There are some special cases, and the general slicing rules were discussed in detail in an earlier lesson entitled Learn to Program using Python: Strings, Part II

Pack the multiply-nested tuple

The code in Listing 1 packs the nested tuple by successively nesting an existing tuple into a new tuple. This results in a tuple that is nested several levels deep as shown by the program output in Listing 2.

```
A nested tuple
('X', ((1, 2, 3), 'B'), 'Y', 'Z')

Listing 2
```

What is the length of the tuple?

Although this program doesn't include the use of the **len()** method, it is easy enough to determine by inspection that the tuple has a length of four items. Those four items are listed on separate lines in the chart in Figure 1 to help you identify them. In addition to listing the items on separate lines, the chart also provides the index value for each item.

0	'X'
1	((1, 2, 3), 'B')
2	'Y'
3	'Z'

Figure 1 Chart Showing Four Items

What are the available index values?

The four items contained in the tuple can be accessed using index values of 0, 1, 2, and 3.

Everything that is highlighted in red in Listing 2 is a single item, which can be accessed using the index value of 1. This item appears in the second line of the above chart.

This item is a nested tuple, which also contains another nested tuple. I will be making use of this fact later when I combine indexing with slicing. But for now...

Let's concentrate on slicing alone

Listing 3 shows a simple slice applied to the top-level nested tuple shown in Listing 2. This slicing syntax means to get and return the set of items in the tuple named **t4** beginning with the item at index value 1 and ending at index value (3-1) or 2.

```
print "A slice"
print t4[1:3]
```

```
Listing 3
```

The item at index 1 is a tuple

Listing 4 shows the output produced by the code in Listing 3. If you compare this output with the chart given earlier, you will see that the group of items from index value 1 to and including index value 2 were extracted and used to produce a new tuple.

```
A slice
(((1, 2, 3), 'B'), 'Y')

Listing 4
```

I highlighted the item extracted from index value 1 in red and the item extracted from index value 2 in blue to make it easier for you to identify them.

The item extracted from index value 1 is itself a tuple that contains another nested tuple.

The next thing that I am going to do is to extract this tuple using an index and apply a slicing operation to it.

Indexing plus slicing

Listing 5 shows a code fragment that combines indexing and slicing. This fragment

- Extracts the item at index value 1 from the top-level tuple named **t4**. This item is itself a tuple.
- Applies a slicing operation that extracts the group of items from index value 0 up to, but not including, index value 1.

```
print "An indexed slice"
print t4[1][0:1]

Listing 5
```

Doesn't that mean a group of one item?

Stated differently, the slicing operation in this case selects a group of items where the number of items in the group is one beginning at the item whose index value is 1.

What does the group of one look like?

Now consider once more the different items in the top-level tuple named **t4**. I have reproduced the earlier chart below for viewing convenience.

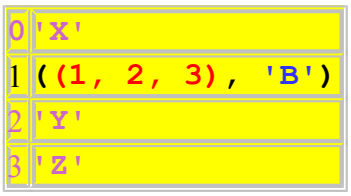

Figure 2 Items in the Top-Level Tuple

Let's be different

Just to be a little different, in this case, I highlighted the item at index value 1 by using washed-out colors for the other items. I also used color to identify the two items contained in the sub-tuple item at index value 1.

The item at index value 0 in the sub-tuple is highlighted in red. The item at index value 1 in the sub-tuple is highlighted in blue.

Let's see some output

Listing 6 shows the output produced by the code in Listing 5. As you can see, the output in this case is a single-item tuple, and that item is the item highlighted in red in the above chart.

```
An indexed slice
((1, 2, 3),)

Listing 6
```

I'm going to leave the final step in this program as an exercise for the student. See the question in the Review section.

Summary

This lesson has taught you how to combine indexing and slicing to access groups of items in nested tuples.

What's Next?

The next lesson will show you how to modify values stored in an object referred to by an item in a tuple.

Listing of Sample Program

A complete listing of the program is shown in Listing 9.

```
# File Tuple06.py
# Rev 8/4/00
# Copyright 2000, R. G. Baldwin
```

```
# Illustrates slicing nested
#  tuples - combines indexing
#  and slicing.
#-------------------------------
# Create a simple tuple
t1 = 1,2,3

# Create/print nested tuple
t3 = t1,"B"
t4 = "X",t3,"Y","Z"
print "A nested tuple"
print t4

print "A slice"
print t4[1:3]

print "An indexed slice"
print t4[1][0:1]

print "Double-indexed slice"
print t4[1][0][1:3]
```

Listing 9

Review

Show how to write the code that will use multiple square-bracket indexing to drill down two levels deep in the top-level tuple named **t4** above to produce the output shown in Listing 8.

```
Double-indexed slice
(2, 3)
```

Listing 8

Ans: See Listing 7.

```
print "Double-indexed slice"
print t4[1][0][1:3]
```

Listing 7

Learn to Program Using Python: Indirection

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language. After we cover regular Python, the lessons will cover JPython. This will form a link between Python and Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings, without losing your place, while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course. (You will find a consolidated index to all of my Python, Java, and XML tutorials on my website.)

Introduction

This is the end of a miniseries of lessons designed to teach you about tuples.

Previous lessons have illustrated

- How to create (pack) a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.
- How to create empty tuples.
- How to create single-item tuples.
- How to unpack a tuple
- How to use a numeric index to access the items in nested tuples.
- How to combine indexing and slicing to access groups of items in nested tuples.

Preview

This lesson will teach you how to use *indirection* to modify the value of an object referred to by a tuple item.

What Is a Tuple?

A tuple is an immutable ordered list of objects. It can contain references to any type of object. See previous lessons in this series for a more detailed description.

What is Indirection?

To begin with, indirection is one of the most important programming concepts in modern computer programming. I remember reading somewhere *"Almost any programming problem can be solved with enough levels of indirection."* While this statement may not be absolutely true, it does serve to illustrate the importance of indirection in modern computer programming.

Look in the mailbox

Indirection works something like a party game that I recall from my childhood.

An adult would tell the children to go look in the mailbox. When they did, they would find a note telling them to go look in the kitchen cabinet. There they would find a note telling them to go look under the bed. This process might continue through several more notes until finally they would find a note telling them to look on the back porch. There they would find a box full of goodies.

Modern languages use indirection

Most modern programming languages make use of indirection in some form or another, and Python is no exception. Figure 1 contains a Python code fragment that illustrates the children's game mentioned above:

```
underTheBed = "Back Porch"
kitchenCabinet = underTheBed
```

```
mailbox = kitchenCabinet

print mailbox
```

Figure 1 The Children's Game

In this simple example, the variable (area of memory) known as **mailbox** contains a reference or pointer to the variable known as **kitchenCabinet**.

The variable known as **kitchenCabinet** contains a reference or pointer to the variable known as **underTheBed**.

The variable known as **underTheBed** contains a reference to a string object, which in turn identifies the **Back Porch** as the end of the path.

Traversing the path

Fortunately, unlike children playing the party game, Python programmers are not required to traverse the path one step at a time. In this example, the print statement shown in the last line will cause the entire path to be traversed and the printed output will be:

Back Porch

Why is this called indirection?

This process is called indirection because the final objective is attained through an indirect path rather than a direct path.

Sample Program

Indirect modification

Listing 9, near the end of the lesson, shows a Python script that:

- Creates and displays a simple list containing three integer values.
- Modifies the value stored at index value 1 in the list and displays the modified list.
- Creates and displays a simple tuple containing the list.
- Modifies a value stored in the list, which is referred to by an item in the tuple.
- Displays the tuple with the modified list value.
- Attempts unsuccessfully to modify the value of an item in the tuple

A tuple is immutable, a list is mutable

The important point here is that while the program is able to indirectly modify a value stored in a mutable list referred to by an item in the tuple, it is unable to directly modify the value of an item in the tuple.

A list is a mutable sequence.

A tuple is an immutable sequence.

Now I will break this program down and discuss it in fragments.

The original list object

Listing 1 shows the code that creates and displays the list, directly modifies a value in the list, and then displays the modified list.

```
# File Tuple07.py
#------------------------------
# Create a list
L1 = [1,2,3]
print "Original list"
print L1
# Modify the list
L1[1] = "a"
print "Modified list"
print L1
```

Listing 1

This code was included primarily to illustrate what I mean by directly modifying a list (or tuple) item.

The boldface statement in Listing 1 modifies the value of the list item whose index value is 1. In this case, the value of the list item is modified from an integer value of **2** to a reference to a string object containing the letter **a**.

We will see later that because a tuple is immutable, a similar statement cannot be successfully applied to a tuple.

Let's see some output

Listing 2 shows the output produced by the code fragment in Listing 1. As expected from the above explanation, the modified list is different from the original list.

```
Original list
[1, 2, 3]
Modified list
[1, 'a', 3]
```

Listing 2

Put the list in a tuple

Listing 3 shows code that creates a tuple containing the list. Actually, the tuple probably doesn't physically contain the list, although we often speak of it that way.

Rather, the tuple probably contains an item that refers to the list.

```
# Create a simple tuple
#  containing the list
t1 = "a",L1,"c"
print "Tuple containing list"
print t1

Listing 3
```

Do we really care?

As high-level Python programmers, we don't need to be too concerned with the physical implementation. Rather, we need to be concerned with the functional behavior.

Later, I will use the item that refers to the list to indirectly modify the value of one of the items in the list.

The output produced by the code in Listing 3 is shown in Listing 4. As you can see, the print statement traverses the path and shows us the contents of the list as though it is physically embedded in the tuple.

```
Tuple containing list
('a', [1, 'a', 3], 'c')

Listing 4
```

The main point of the lesson

Listing 5 shows a boldface statement that:

- Gains access to the tuple item at index value 1, which is a reference to the list.
- Uses that reference to gain access to and modify the value of the list item at index value 1.

```
# "Modify list value"
t1[1][1] = "X"
print "Modified stored value"
print t1

Listing 5
```

What does this really mean?

You might interpret the behavior of this statement as follows:

- Go to index value 1 of the tuple, where you will find a reference to a list.
- Go to the list referred to by the tuple item and store a new value in the list item at index value 1.

Hence, this code uses an immutable value stored as an item in a tuple to access and modify a mutable value stored as an item in the list that the tuple item refers to. This is clearly a case of *indirection*.

Let's see the output

Listing 6 shows the output from the code fragment in Listing 5 with the list item whose value was modified highlighted in boldface.

```
Modified stored value
('a', [1, 'X', 3], 'c')
```

Listing 6

The list item was changed from a reference to a string containing a lower-case **a** to a reference to a string containing an upper-case **X**.

Try to modify a tuple item

Just for fun, let's see what happens if we attempt to modify the value of an item in the tuple. The code to do this is shown in Listing 7.

```
print "Modify the tuple"
t1[1] = "A"
```

Listing 7

We already know that because the tuple item is immutable, its value cannot be modified. Therefore, the error shown in Listing 8 is produced.

```
Modify the tuple
Traceback (innermost last):
  File "tuple07.py",
       line 28, in ?
    t1[1] = "A"
TypeError: object doesn't
  support item assignment
```

Listing 8
(Line breaks manually inserted)

Summary

This lesson has taught you how to use *indirection* to modify the value of an object referred to by a tuple item.

What's Next?

That's the end of our miniseries on tuples. The next lesson will take up the topic of dictionaries.

Review

Show how to write code that will modify a character in a string object referred to by an item in a tuple.

Ans: This is a trick question. It cannot be done because, as explained in an earlier lesson entitled Learn to Program using Python: Strings, Part II , a string object is an immutable sequence. This is illustrated by the code in Listing 10.

```
# File Tuple24.py
# Rev 08/02/00
# Copyright 2000, R. G. Baldwin
# Illustrates attempt to
#  modify char in string
#  referred to by item
#  in tuple.
#-------------------------------
t1 =(1,2,"abc")
print t1
t1[2][0] ="Z"
print t1
```

Listing 10

Listing 10, which produces the error message shown in Listing 11.

```
(1, 2, 'abc')
Traceback (innermost last):
  File "tuple24.py",
       line 11, in ?
    t1[2][0] ="Z"
TypeError: object doesn't
  support item assignment
```

Listing 11
(Line breaks manually inserted)

Listing of Sample Program

A complete listing of the program discussed in the early part of this lesson is shown in Listing 9.

```
# File Tuple07.py
# Rev 8/5/00
# Copyright 2000, R. G. Baldwin
# Illustrates modifying value
#  stored in an object referred
#  to by a tuple item.
#-------------------------------
# Create a list
L1 = [1,2,3]
print "Original list"
print L1
# Modify the list
L1[1] = "a"
print "Modified list"
print L1

# Create a simple tuple
#  containing the list
t1 = "a",L1,"c"
print "Tuple containing list"
print t1

# "Modify list value"
L1[1] = "X"
print "Modified stored value"
print t1

print "Modify the tuple"
t1[1] = "A"
```

Listing 9

Learn to Program Using Python: Getting Started with Dictionaries

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language. After we cover regular Python, the lessons will cover JPython. This will form a link between Python and Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course. (You will find a consolidated index to all of my Python, Java, and XML tutorials, including Lesson 1 mentioned above, on my website.)

Introduction

This is the beginning of a miniseries of lessons designed to teach you about dictionaries.

Containers or collections

In previous lessons, you learned about strings, lists, and tuples. Dictionaries fall in the same general family as these three types of objects (containers or collections), but with significant differences.

Preview

This lesson will introduce you to the characteristics of the Python dictionary, and will show you how to use the basic characteristics. Subsequent lessons will show you how to use the characteristics of dictionaries beyond the basics.

What Is a Dictionary?

Mutable unordered set...

A dictionary is a mutable unordered set of *key:value* pairs. Its values can contain references to any type of object.

In other languages, similar data structures are often called associative arrays or hash tables.

Not a sequence

Unlike the string, list, and tuple, a dictionary is not a sequence. The sequences are indexed by a range of ordinal numbers. Hence, they are ordered.

Indexed by keys, not numbers

Dictionaries are indexed by *keys*. According to the Python Tutorial, a key can be *"any non-mutable type."* Since strings and numbers are not mutable, you can always use a string or a number as a key in a dictionary.

What about a tuple as a key?

You can use a tuple as a key if all of the items contained in the tuple are immutable. Hence a tuple to be used as a key can contain strings, numbers, and other tuples containing references to immutable objects.

What about a list as a key?

You cannot use a list as a key because a list is mutable. That is to say, its contents can be modified using its **append()** method.

What does a dictionary look like?

You can create a dictionary as an empty pair of curly braces, {}.

Alternatively, you can initially populate a dictionary by placing a comma-separated list of key:value pairs within the braces.

Can I add key:value pairs later?

Later on, you can add new key:value pairs through indexing and assignment, using the new key as the index.

Keys must be unique, otherwise...

Each of the keys within a dictionary must be unique. If you make an assignment using an existing key as the index, the old value associated with that key is overwritten by the new value.

New key:value pairs add to the dictionary

If you make an assignment using a new key as the index, the new key:value pair will be added to the dictionary. Thus, the size of a dictionary can increase at runtime.

Removing key:value pairs

You can use **del** to remove a key:value pair from a dictionary. Thus, the size of a dictionary can also shrink at runtime.

Add/modify key:value pairs

You add or modify key:value pairs using assignment with the key as an index into the dictionary.

Fetching values associated with keys

You extract a value from a dictionary using the associated key as an index. Attempting to extract a value using a non-existent key produces an error.

Getting a list of keys

You can obtain a list of all of the keys currently contained in a dictionary by invoking the **keys()** method on the dictionary. This produces a list of keys in random order. You can invoke the **sort()** method on the list to sort the keys if need be.

Membership testing

You can determine if a specific key is contained in the dictionary by invoking the **has_key()** method on the dictionary.

Getting a list of values

You can obtain a list of all of the values currently contained in a dictionary by invoking the **values()** method on the dictionary.

Summary of Dictionary Characteristics

The following is a general summary of the characteristics of a Python dictionary:

- A dictionary is an unordered collection of objects.
- Values are accessed using a key rather than by using an ordinal numeric index.
- A dictionary can shrink or grow as needed.
- The contents of dictionaries can be modified.
- Dictionaries can be nested.
- Dictionaries are not sequences. Sequence operations such as *slice* cannot be used with dictionaries.
- According to Learning Python by Lutz and Ascher, *"Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is very fast."*

These characteristics will be illustrated using sample programs in this and subsequent lessons.

Sample Program

Listing 7, near the end of the lesson, shows a very basic Python script that:

- Creates and displays an empty dictionary and its length.
- Adds two key:value pairs to the dictionary.
- Displays the modified dictionary and its length.
- Indexes the dictionary by a key value to get and display the value associated with the key.

Now I will break this program down and discuss it in fragments.

Use of boldface

I typically use boldface for emphasis in these lessons, although Python code is not written using boldface.

The empty dictionary

The boldface line in Listing 1 shows a statement that creates an empty dictionary *(I will illustrate creating an initializing a dictionary at the same time in a subsequent lesson)*.

```
# File Dict02.py
#------------------------------
# Create an empty dictionary
d1 = {}

# Print dictionary and length
print "Dictionary contents"
print d1
print "Length = ",len(d1)
```

Listing 1

The remaining code in Listing 1 displays the empty dictionary and also gets and displays its length using **len()**.

Let's see some output

The output produced by this code fragment is shown in Listing 2.

```
Dictionary contents
{}
Length =  0
```

Listing 2

As you can see from the boldface line in Listing 2, the empty dictionary is displayed as a pair of empty curly braces.

As you probably expected, the length of the dictionary at this point is reported to be zero.

Adding key:value pairs

The first two statements in Listing 3 use indexing by key value and assignment to add two new key:value pairs to the dictionary.

```
# Add two items
d1["to"] = "two"
d1["for"] = "four"

# Print dictionary and length
print "Dictionary contents"
print d1
print "Length = ",len(d1)
```

Listing 3

For example, the boldface line creates a new key:value pair with a key of **"to"** and a value of **"two"**. Although I used a string for the key, I could have used any immutable type as explained earlier.

In the case, the value is a string. However, it could be any object of any type.

Display the modified dictionary

After adding the two new key:value pairs to the dictionary, the remaining code in Listing 3 displays the modified dictionary and its length.

The output produced by this code fragment is shown in Listing 4.

```
Dictionary contents
{'for': 'four', 'to': 'two'}
Length =  2
```

Listing 4

Order is randomized

The contents of the modified dictionary are shown by the boldface line in Listing 4.

You will note that the dictionary now contains the two key:value pairs that I added.

You should also note that they do not appear in the same order that I added them.

According to Learning Python by Lutz and Ascher, *"Python randomizes their order in order to provide quick lookup. Keys provide the symbolic (not physical) location of items in a dictionary."*

Length is now 2

As you probably already guessed, Listing 4 shows the length of the modified dictionary to be 2. In other words, it now contains two key:value pairs.

Fetching a value

The boldface portion of Listing 5 shows indexing by key value to fetch and display the value associated with the key **"to"**.

```
# Get/display value by key
print "Value for to = ",d1["to"]
```

Listing 5

Other programming languages, such as Java, require you to invoke methods with names such as **put()** and **get()** to store and fetch key:value pairs. However, in Python, a simple indexing syntax is used for that purpose.

The output produced by this code fragment is shown in Listing 6.

```
Value for to =  two
```

Listing 6

As you have undoubtedly already figured out, the value associated with the key **"to"** is **"two"** (because that is the value that I stored there in Listing 3).

Summary

This lesson has introduced you to the characteristics of the Python dictionary, and has shown you how to use those basic characteristics.

What's Next?

Upcoming lessons will contain sample programs that illustrate a number of characteristics of dictionaries, including how to:

- Create and use valid key types.
- Overwrite old values.
- Nest dictionaries.
- Delete items from dictionaries.
- Test for key membership.
- Get, sort, and use a key list.
- Get and use a value list.

Review

1. True or false? A dictionary is an immutable object.

Ans: False. A dictionary is mutable because its existing items can be modified, new items can be added, and existing items can be deleted.

2. True or false? A dictionary is an unordered collection of objects.

Ans: True. The items in a dictionary are not maintained in any specific order, and a dictionary can contain references to any type of objects.

3. True or false: Sequence operations such as *slicing* and *concatenation* can be applied to dictionaries.

Ans: False. A dictionary is not a sequence. Because it is not maintained in any specific order, operations that depend on a specific order cannot be used.

4. True or false? Dictionaries are indexed using keys instead of ordinal offset values.

Ans: True.

5. True or false? Any object can be used as a valid key in a dictionary.

Ans: False. Only non-mutable types can be used as a key.

6. True or false? Because a tuple is non-mutable, any tuple can be used as a key in a dictionary.

Ans: False. A tuple can be used as a key only if all of the objects that it contains are also non-mutable.

7. True or false? Numbers and strings can always be used as keys.

Ans: True.

8. True or false? Lists can be used as keys.

Ans: False. Lists cannot be used as keys in a dictionary because they are mutable.

9. Describe the syntax of a dictionary.

Ans: A dictionary consists of none, one, or more key:value pairs, separated by commas, and enclosed in a pair of curly braces.

10. How do you add key:value pairs to an existing dictionary?

Ans: You can add new key:value pairs by using indexing and assignment, where the new key is the index.

11. True or false? If the addition of a new key:value pair causes the size of the dictionary to grow beyond its original size, an error occurs.

Ans: False. Dictionaries grow or shrink on an as-needed basis.

12. Can you remove key:value pairs from a dictionary, and if so, how?

Ans: You can use **del** to remove an existing key:value pair from a dictionary using the following syntax:

del dictionary[key]

13. Write a Python program that will:

- Create an empty dictionary.
- Add some new key:value pairs to the dictionary.
- Fetch and display some of the values.

Ans: See Listing 7.

14. True or false? You can obtain a sorted list of the keys currently contained in a dictionary by invoking the **keys()** method on the dictionary.

Ans: False. This will give you a list of keys, but they will not be sorted. You can sort them if need be by invoking the **sort()** method on the list.

15. True or false? You can determine if a specific key is contained in a dictionary by invoking the **has_key()** method on the dictionary.

Ans: True.

16. True or false? You can obtain a sorted list of the values currently contained in a dictionary by invoking the **getValues()** method on the dictionary.

Ans: False. You can obtain an unordered list of the values by invoking the **values()** method on the dictionary.

Listing of Sample Program

A complete listing of the program is shown in Listing 7.

```
# File Dict02.py
# Rev 08/06/00
# Copyright 2000, R. G. Baldwin
# Illustrates some basic
#  characteristics of
#  dictionaries
#-------------------------------
# Create an empty dictionary
d1 = {}

# Print dictionary and length
print "Dictionary contents"
print d1
print "Length = ",len(d1)
```

```
# Add two items
d1["to"] = "two"
d1["for"] = "four"
# Print dictionary and length
print "Dictionary contents"
print d1
print "Length = ",len(d1)

# Get/display value by key
print "Value for to = ",d1["to"]
```

Listing 7

Learn to Program using Python: Valid Keys, Key Lists, Iteration

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language. After we cover regular Python, the lessons will cover JPython. This will form a link between Python and Java.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings, without losing your place, while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course. *(You will find a consolidated index to all of my Python, Java, and XML tutorials, including Lesson 1 mentioned above, on my website.)*

Introduction

A previous lesson entitled Learn to Program Using Python: Getting Started with Dictionaries introduced you to the characteristics of the Python dictionary, and showed you how to use the basic characteristics.

Preview

This lesson will teach you about valid keys, key lists, and iteration on key lists.

What Is a Dictionary?

The following is a general summary of the characteristics of a Python dictionary:

- A dictionary is an unordered collection of objects.
- Values are accessed using a key rather than by using an ordinal numeric index.
- A dictionary can shrink or grow as needed.
- The contents of dictionaries can be modified.
- Dictionaries can be nested.
- Sequence operations such as *slice* cannot be used with dictionaries.

Will illustrate these characteristics

Some of these characteristics were illustrated in a previous lesson entitled *Getting Started with Dictionaries*. Other characteristics will be illustrated using sample programs in this and subsequent lessons.

Sample Program

Listing 11, near the end of the lesson, shows a Python script that:

- Creates, initializes, and displays a dictionary.
- Adds two *key:value* pairs to the dictionary and displays the new version.
- Gets and displays a list of the keys contained in the dictionary.
- Iterates on the key list to produce a display of the keys and their associated values in a tabular format.
- Attempts unsuccessfully to use a list as an index on the dictionary, and displays the error that results.

I will break this program down and discuss it in fragments.

Use of boldface

I typically use boldface for emphasis in these lessons, although Python code is not written using boldface.

The initialized dictionary

The boldface statement in Listing 1 creates a new dictionary and initializes it to contain one key:value pair. The key is **5** and the value is **"number"**.

```
# File Dict04.py
# Rev 08/06/00
# Copyright 2000, R. G. Baldwin
# Illustrates
#  Valid keys
#  keys() method
#  Iteration on a list of keys
#-------------------------------
# Create initialized dictionary
d1 = {5:"number"}

# Display it
print d1,'\n'

Listing 1
```

The remaining code in Listing 1 displays the dictionary.

The output produced by this code fragment is shown in Listing 2.

```
{5: 'number'}
```

```
Listing 2
```

As you can see from the boldface line in Listing 2, the dictionary is displayed as a pair of curly braces containing the key:value pair discussed above.

Adding key:value pairs

The first two statements in Listing 3 use indexing by key value and assignment to add two new key:value pairs to the dictionary.

```
# Add items
d1["to"] = "string"
d1[(1,"a",("b","c"))] = "tuple"
# Display it
print d1,'\n'

Listing 3
```

Have we seen this before?

We saw something like this in a program in the earlier lesson entitled *Getting Started with Dictionaries*. However, that lesson used string objects for keys.

Using a tuple as a key

In this program, I used a string for one key and I used a tuple for the other key, as shown by the highlighted material in Listing 3. *(Note that the tuple used as a key contains a nested tuple.)*

Is a tuple a valid key?

Recall that you can use any immutable object as a key.

When the key is a tuple, the contents of the tuple must themselves be immutable.

That requirement is satisfied here because the contents of the tuple (and its nested tuple) are numbers and strings.

Values can be any type

In this case, both values of the key:value pair are strings. However, they could be any type.

Display the modified dictionary

After adding the two new key:value pairs to the dictionary, the remaining code in Listing 3 displays the modified dictionary.

The output produced by this code fragment is shown in Listing 4.

```
{(1, 'a', ('b', 'c')): 'tuple',
 'to': 'string',
 5: 'number'}

Listing 4
(Line beaks entered manually.)
```

Manually entered line breaks and color

Note that I manually entered line breaks to force the dictionary to fit in the available display space for this lesson. I also colored the keys red and the values blue to make them easier to separate visually.

Order is randomized

As shown in Listing 4, the dictionary now contains the original key:value pair, plus the two new key:value pairs that I added.

Note that the items do not appear in the same order that I added them. As mentioned in the earlier lesson entitled *Getting Started with Dictionaries*, items are purposely stored in a dictionary in a random order.

Getting a list of keys

The boldface code in Listing 5 invokes the **keys()** method on the dictionary object to get a list of the keys currently stored in the dictionary.

```
# Get and display keys
L1 = d1.keys()
print "The keys are:"
print L1,'\n'

Listing 5
```

This list of keys is then displayed as shown in Listing 6.

```
The keys are:
[(1, 'a', ('b', 'c')), 'to', 5]

Listing 6
```

This is an ordinary list, consisting of a sequence of items in square brackets, separated by commas. *(I discussed lists in an earlier lesson entitled Lists, Part I.)*

Color was added for this display

Note that I color-coded the items in the key list using alternating colors of red and blue to make them easier to separate visually.

The first key is a tuple, the second key is a string, and the third key as a number.

No surprise here

This should come as no surprise, because these are the same keys that were created earlier for the three key:value pairs. This is simply a different view of the keys.

Useful for iteration

However, as we will see shortly, this view of the keys is very useful for iterating on the keys in the dictionary.

Iterate on the key list

The boldface code in Listing 7 uses a **for** loop to iterate on the key list to display each key:value pair in a tabular format.

```
# Iterate on the key list
# Print dictionary and length
print "Dictionary contents"
for x in L1:
  print x,'\t',d1[x]
print "Length = ",len(d1),'\n'

Listing 7
```

According to the Python Tutorial, *"Python's **for** statement iterates over the items of any sequence (e.g., a list or a string), in the order that they appear in the sequence."*

What does this really mean?

I will have quite a lot more to say about the Python **for** loop in a subsequent lesson. For now, you can interpret the operation of the boldface code in Listing 7 as follows:

- Get an item from the list of keys named **L1** and store the item (key) in a variable named **x**.
- Print the value of **x**
- Print a *tab* character ('\t') on the same line.
- Use **x** as an index and print (also on the same line) the value in the dictionary associated with the key stored in **x**.
- Repeat this process for each item in the list of keys.

Then get and print the length

After each of the key:value pairs have been printed in the tabular format described above, the code in Listing 7 gets and prints the length of the dictionary using the **len()** method discussed in earlier lessons.

The output

Listing 8 shows the output produced by the code in Listing 7.

```
Dictionary contents
(1, 'a', ('b', 'c'))    tuple
to      string
5       number
Length =  3
```

Listing 8

In Listing 8, the keys were colored red and the values were colored blue to make it easier to separate them visually.

Not too pretty but...

Although the length of the first key is so long that it disrupts the column structure in the tabular format, you can see how the items in the key list were used in an iterative manner to fetch and display each key along with its associated value.

Although it should come as no surprise to you at this point, Listing 8 also shows the length of the dictionary to be three items.

A list is not a valid key

A list is a mutable object. Therefore, it cannot be used as a key in a dictionary.

The code in Listing 9 demonstrates the truth of the previous statement.

```
# A list is not a valid key
print "Try to index with a list"
d1[ [8,9,10] ] = "not allowed"
```

Listing 9

This code attempts to index the dictionary object using a list as a key to add a new item to the dictionary.

The output from the code in Listing 9 is shown in Listing 10.

```
Try to index with a list
Traceback (innermost last):
```

```
  File "Dict04.py", line 33, in ?
    d1[ [8,9,10] ] = "not allowed"
TypeError: unhashable type
```

Listing 10

As you can see, the program terminated at this point with an error message telling us that the list is an *unhashable type*.

Summary

This lesson has taught you about valid keys, key lists, and iteration on key lists.

What's Next?

Upcoming lessons will contain sample programs that illustrate a number of characteristics of dictionaries, including the following:

- Overwrite old values in a dictionary.
- Nest dictionaries.
- Delete items from dictionaries.
- Test for key membership.
- Get, sort, and use a key list.
- Get, sort, and use a value list.

Review

1. Write Python code that will create, initialize, and display a dictionary. The dictionary should have keys of type string and number, and should have values of type list and tuple.

Ans: See Listing 12.

```
# File Dict10.py
# Rev 08/08/00
# Copyright 2000, R. G. Baldwin
# Illustrates initializing a
#  dictionary
#-------------------------------
d1 = {5:[6,7,8],"a":(1,2,3)}
print d1
```

Listing 12

2. Write Python code that gets and displays a list of the keys contained in a dictionary.

Ans: See Listing 13.

```
# File Dict12.py
# Rev 08/08/00
# Copyright 2000, R. G. Baldwin
# Illustrates getting a list of
#  keys
#-------------------------------
d1 = {5:[6,7,8],"a":(1,2,3)}
print d1.keys()
```

Listing 13

3. Write Python code that will iterate on a key list to produce a display of the keys and their associated values. Each key:value pair should appear on a new line, and the value should be separated from the key using a colon.

Ans: See Listing 14.

```
# File Dict14.py
# Rev 08/06/00
# Copyright 2000, R. G. Baldwin
# Illustrates iterating on a key
#  list from a dictionary
#-------------------------------
# Create initialized dictionary
d1 = {5:"number",\
      "a":"string",\
      (1,2):"tuple"}

# Iterate on a key list
print "Dictionary contents"
for x in d1.keys():
  print x,':',d1[x]
```

Listing 14

4.. True or false? A list can be used as a key in a dictionary.

Ans: False. Only immutable types can be used as keys in a dictionary. A list is a mutable type.

Listing of Sample Program

A complete listing of the program is shown in Listing 11.

```
# File Dict04.py
# Rev 08/06/00
# Copyright 2000, R. G. Baldwin
# Illustrates
#  Valid keys
#  keys() method
#  Iteration on a list of keys
#-------------------------------
```

```
# Create initialized dictionary
d1 = {5:"number"}
# Display it
print d1,'\n'

# Add items
d1["to"] = "string"
d1[(1,"a",("b","c"))] = "tuple"
# Display it
print d1,'\n'

# Get and display keys
L1 = d1.keys()
print "The keys are:"
print L1,'\n'

# Iterate on the key list
# Print dictionary and length
print "Dictionary contents"
for x in L1:
  print x,'\t',d1[x]
print "Length = ",len(d1),'\n'

# A list is not a valid key
print "Try to index with a list"
d1[ [8,9,10] ] = "not allowed"
```

Listing 11

Learn to Program using Python: Using Tuples as Keys

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. The first lesson entitled Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

As of the date of this writing, EarthWeb doesn't maintain a consolidated index of my Python tutorial lessons, and sometimes my lessons are difficult to locate on the EarthWeb site. You will find a consolidated index of my tutorial lessons at my web site.

Introduction

A previous lesson entitled *Learn to Program using Python: Getting Started with Dictionaries,* introduced you to the characteristics of the Python dictionary, and showed you how to use the basic characteristics. This lesson will teach you about using tuples as keys in a Python dictionary.

What Is a Dictionary?

The following is a general summary of the characteristics of a Python dictionary:

- A dictionary is an unordered collection of objects.
- Values are accessed using a key.
- A dictionary can shrink or grow as needed.
- The contents of dictionaries can be modified.
- Dictionaries can be nested.
- Sequence operations such as *slice* cannot be used with dictionaries.

Will illustrate these characteristics

Some of these characteristics were illustrated in previous lessons. Other characteristics will be illustrated using the sample programs in this and subsequent lessons.

Sample Program

Listing 9 near the end of the lesson shows a Python script that:

- Creates, initializes, and displays a simple dictionary.
- Overwrites an existing string value in the dictionary with a tuple value.
- Adds a new key:value pair where the key is a tuple containing immutable objects.

- Attempts unsuccessfully to add a new key:value pair where the key is a tuple containing a mutable list.

I will break this program down and discuss it in fragments.

Use of boldface

Although Python code is not written using boldface, I typically use boldface for emphasis when I display Python code in these lessons.

The initialized dictionary

The boldface statement near the top of Listing 1 below creates a new dictionary and initializes it to contain two key:value pairs. For the first element, the key is **5** and the value is the string **"number"**. For the second element, the key is **"to"** and the value is the string **"string"**. *(These two keys satisfy the requirement that the keys in a dictionary must be immutable. Numbers and strings are immutable in Python.)*

```
# Create initialized dictionary
d1 = {5:"number","to":"string"}
print "Dictionary contents"
print "Key",' : ',"Value"
for x in d1.keys():print x,' : ',d1[x]

print ""# Print blank line

Listing 1
```

Display the dictionary contents

The code in Listing 1 also displays the contents of the dictionary using a **for** loop to iterate on a list of keys obtained by invoking the **keys()** method on the dictionary. *(This methodology was explained in the earlier lesson entitled "Learn to Program using Python: Valid Keys, Key Lists, Iteration.")*

The output produced by the above code is shown in Listing 2 below.

```
Dictionary contents
Key  :  Value
to  :  string
5  :  number

Listing 2
```

Keys must be unique

Each of the keys within a dictionary must be unique. If you make an assignment using an existing key as the index, the old value associated with that key is

overwritten by the new value. You can use this characteristic to advantage in order to modify an existing value for an existing key.

Modifying a value

You can modify key:value pairs using assignment with the key as an index into the dictionary. If you assign a new value using an existing key, the new value overwrites the value previously associated with that key. The new value can be of the same type as the original value, or it can be a different type.

As shown in the program output in Listing 2 above, the value associated with the key **5** is the value **number**. Thus, the value is a string. *(Note that 5 is a key here, and is not an ordinal index as is used with sequences such as lists or strings.)*

The boldface line of code in Listing 3 below assigns a new value to the key **5** *(changing the value associated with the key and not the key itself)*. In this case, the type of the value is not a string as before. Rather, it is a tuple.

```
# Overwrite existing value with a tuple
# value
d1[5]=("ab","cd","ef")
print "New dictionary contents"
print "Key",' : ',"Value"
for x in d1.keys():print x,' : ',d1[x]
print ""# Print blank line
```

Listing 3

Display the new value in the dictionary

The remaining code in Listing 3 uses the same iterative technique as before to display the keys in the dictionary along with their associated values.

Listing 4 below shows the output produced by this code. Compare this output with the output shown in Listing 2 above. *(I have highlighted the new tuple value in boldface to make it easy for you to identify.)*

```
New dictionary contents
Key  :  Value
to  :  string
5  :  ('ab', 'cd', 'ef')
```

Listing 4

A dictionary can shrink or grow as needed

If you make an assignment using a new key as the index, the new key:value pair will be added to the dictionary. Thus, the size of a dictionary can increase at runtime.

(If you delete an element, the size can decrease. I will illustrate this in a subsequent lesson.)

The next portion of this program increases the size of the dictionary by adding a new key:value pair.

Using a tuple as a key

A key must be immutable. You can use a tuple as a key if all of the elements contained in the tuple are immutable. *(If the tuple contains mutable objects, it cannot be used as a key.)* Hence a tuple to be used as a key can contain strings, numbers, and other tuples containing references to immutable objects.

The boldface statement in Listing 5 below adds a new key:value pair to the dictionary using a two-element tuple as a key.

```
# Add a key value pair with a tuple as
# a key
print "Index with a tuple"
d1[(1,"a")] = "tuple"
print "New dictionary contents"
print "Key",' : ',"Value"
for x in d1.keys():print x,' : ',d1[x]
print ""# Print blank line
```

Listing 5

The tuple used as a key in Listing 5 contains a number and a string, both of which are immutable. Therefore, this tuple is a valid key since all of its elements are immutable.

The remaining code in Listing 5 displays the new value in the dictionary.

Display the dictionary

Listing 6 below shows the output produced by the code in Listing 5 above.

```
Index with a tuple
New dictionary contents
Key  :  Value
(1, 'a')  :  tuple
to  :  string
5  :  ('ab', 'cd', 'ef')
```

Listing 6

The new information in the output is the line highlighted in boldface in Listing 6. This line shows the tuple as the key and a string containing the word tuple as a value.

(Recall that the positions of the key:value pairs in a dictionary are randomized. Therefore, the new element in the dictionary is not positioned at the end of the dictionary.)

Several combinations

At this point, the keys in the dictionary consist of one number, one string, and one tuple. The tuple contains only immutable elements.

Similarly, the values in the dictionary consist of one tuple and two strings. The values in a dictionary can be any type. The keys can be any immutable type.

A tuple with mutable elements

You cannot use a list as a key because a list is mutable. Similarly, you cannot use a tuple as a key if any of its elements are lists. *(You can only use a tuple as a key if all of its elements are immutable.)*

The code in Listing 7 below attempts, unsuccessfully, to use a tuple containing a list as a key.

```
# A tuple containing a list is not a
# valid key
print "Try to index with a"
print " tuple containing a list"
d1[(1,"a",["b","c"])] = "tuple"

Listing 7
```

Display the output

The output produced by the code in Listing 7 is shown in Listing 8 below. As you can see, this code caused the program to terminate with an **unhashable type** error.

```
Try to index with a
 tuple containing a list
Traceback (innermost last):
  File "Dict06.py", line 39, in ?
    d1[(1,"a",["b","c"])] = "tuple"
TypeError: unhashable type

Listing 8
```

What's Next?

Upcoming lessons will contain sample programs that illustrate a number of characteristics of dictionaries, including the following:

- Nesting dictionaries.
- Delete items from dictionaries.

- Test for key membership.
- Get, sort, and use a key list.
- Get, sort, and use a value list.

Review

1. Write Python code that will create, initialize, and display a dictionary. The dictionary should have keys of type string and number, and should have values of type string.

Ans: See Listing 1.

2. Describe five general characteristics of a dictionary.

Ans:

1. A dictionary is an unordered collection of objects.
2. Values are accessed using a key.
3. A dictionary can shrink or grow as needed.
4. The contents of dictionaries can be modified.
5. Dictionaries can be nested.
6. Sequence operations such as *slice* cannot be used with dictionaries.

3. True or false? The keys in a dictionary must be mutable.

Ans: False. The keys must be immutable.

4. True or false? You can iterate on a dictionary using its ordinal index.

Ans: False. Unlike strings, lists, and tuples, a dictionary doesn't have an ordinal index. Rather, you index it using key values.

5. True or false: Because a dictionary doesn't have an ordinal index, it is not possible to iterate on it.

Ans: False. You can iterate on a dictionary using a special version of a **for** loop that reads something like the following *(where d1 is the name of the dictionary)*:

```
for x in d1.keys():do something
```

6. True or false? Keys in a dictionary must be unique.

Ans: True. If you make an assignment using an existing key as the index, the old value associated with that key will be overwritten by the new value.

7. How do you modify a value in a dictionary?

Ans: You can modify key:value pairs using assignment with the key as an index into the dictionary. If you assign a new value using an existing key, the new value overwrites the value previously associated with that key.

8. True or false? When you modify a value in a dictionary, the type of the new value must be the same as the type of the old value.

Ans: False. The new value can be the same type as the old value, or it can be a different type.

9. True or false? In the following statement, where **d1** is the name of a dictionary, the use of **5** as an index causes the sixth element in the dictionary to be accessed.

```
d1[5]=("ab","cd","ef")
```

Ans: False. In this case, the 5 is simply an immutable key and has nothing to do with an ordinal index.

10. True or false? The size of a dictionary is fixed when the dictionary is originally created and cannot change thereafter.

Ans: False. A dictionary can shrink or grow as needed. When you add new elements, it can grow to accommodate those new elements. When you delete elements, it can shrink. Growing and shrinking is automatic.

11. True or false? A tuple can never be used as a key in a dictionary.

Ans: False. A tuple can be used as a key so long as all of its elements are immutable.

12. True or false? A tuple containing a list cannot be used as a key in a dictionary.

Ans: True. A list is mutable. Therefore, a tuple containing a list cannot be used as a key in a dictionary.

13. True or false? When you display the contents of a dictionary, the elements will be displayed in the same order that they were added to the dictionary.

Ans: False. The order of the contents of a dictionary is purposely randomized.

Listing of Sample Program

A complete listing of the program is shown in Listing 9.

```
# File Dict06.py
# Rev 12/23/00
# Copyright 2000, R. G. Baldwin
# Illustrates
#   Overwriting values
#   Valid tuple key
#   Invalid tuple key
#---------------------------------//
# Create initialized dictionary
d1 = {5:"number","to":"string"}
print "Dictionary contents"
```

```
print "Key",' : ',"Value"
for x in d1.keys():print x,' : ',d1[x]
print ""# Print blank line

# Overwrite existing value with a tuple
# value
d1[5]=("ab","cd","ef")
print "New dictionary contents"
print "Key",' : ',"Value"
for x in d1.keys():print x,' : ',d1[x]
print ""# Print blank line

# Add a key value pair with a tuple as
# a key
print "Index with a tuple"
d1[(1,"a")] = "tuple"
print "New dictionary contents"
print "Key",' : ',"Value"
for x in d1.keys():print x,' : ',d1[x]
print ""# Print blank line

# A tuple containing a list is not a
# valid key
print "Try to index with a"
print " tuple containing a list"
d1[(1,"a",["b","c"])] = "tuple"
```

Listing 9

Learn to Program using Python: Working with Dictionary Elements

Preface

This document is part of a series of online tutorial lessons designed to teach you how to program using the Python scripting language.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings while you are reading about them.

Something for everyone

Beginners start at the beginning, and experienced programmers jump in further along. The first lesson entitled Learn to Program using Python: Lesson 1, Getting Started provides an overall description of this online programming course.

As of the date of this writing, EarthWeb doesn't maintain a consolidated index of my Python tutorial lessons, and sometimes my lessons are difficult to locate on the EarthWeb site. You will find a consolidated index of my tutorial lessons at my web site.

Introduction

A previous lesson entitled *Learn to Program using Python: Getting Started with Dictionaries,* introduced you to the characteristics of the Python dictionary, and showed you how to use the basic characteristics of a dictionary. Subsequent lessons have taught you how to perform other operations on dictionaries.

This lesson will teach you how to

- Nest dictionaries
- Sort key lists
- Delete elements from dictionaries
- Do membership testing on dictionaries

What Is a Dictionary?

The following is a general summary of the characteristics of a Python dictionary:

- A dictionary is an unordered collection of objects.
- Values are accessed using a key.
- A dictionary can shrink or grow as needed.
- The contents of dictionaries can be modified.
- Dictionaries can be nested.
- Sequence operations such as *slice* cannot be used with dictionaries.

Will illustrate these characteristics

Some of these characteristics were illustrated in previous lessons. Other characteristics will be illustrated using the sample programs in this and subsequent lessons.

Sample Program

Listing 14 near the end of the lesson shows a Python script that:

- Creates, initializes, and displays a dictionary containing other nested dictionaries.
- Deletes some elements from the dictionary and displays the modified dictionary after sorting the dictionary keys.
- Performs membership tests on the dictionary and displays the results.

I will break this program down and discuss it in fragments.

Use of boldface

Although Python code is not written using boldface, I typically use boldface for emphasis when I display Python code in these lessons.

Three initialized dictionaries

The code in Listing 1 creates three initialized dictionaries named **d1**, **d2**, and **d3**. Each of these dictionary objects contains three elements.

```
print "Show nesting"
d1 = {1:10,2:20,3:30}
d2 = {1:40,2:50,3:60}
d3 = {1:70,2:80,3:90}

Listing 1
```

Each element in each of the three dictionary objects created in Listing 1 has a number for its key, and also has a number for its value.

Recall that a key can be any immutable object and a valuecan be any object. Numbers were used here for simplicity. Don't confuse these numeric keys with ordinal indices. These numbers have nothing to do with ordinal indices. Rather, they are simply key values.

Nesting the dictionaries

The single statement in Listing 2 below creates a new dictionary object named **d4**. This object contains three elements. The value of each element is one of the dictionary objects created in Listing 1 above. *(I highlighted them using boldface to make them easy to spot.)*

```
d4 = {"a":d1,"b":d2,"c":d3}

Listing 2
```

The keys for each of the three elements in Listing 2 are strings. Again, the keys could be any immutable objects. I used strings in this case to make it easy to identify them when examining the contents of the dictionary later.

Display the dictionary contents

The code in Listing 3 below uses nested **for** loops to iterate on the dictionary named **d4** and each of the dictionaries nested in **d4** to extract and display the contents of those nested dictionaries.

```
for x in d4.keys():
  print "KEY",'\t',"VALUE"
  print x,'\t',d4[x]
  print "  key",'\t',"value"
  for y in d4[x].keys():\
            print "  ",y,'\t',d4[x][y]

Listing 3
```

The display methodology

A somewhat simpler version of the methodology used to display the contents of the nested dictionaries was explained in the earlier lesson entitled *Learn to Program using Python: Valid Keys, Key Lists, Iteration*.

The big difference here is the use of nested **for** loops. The outer loop iterates on the dictionary named **d4** extracting each nested dictionary in turn. The inner loop is used to iterate on each nested dictionary when it is extracted. *If the loop logic in this display code escapes you at this point, don't worry too much about it. I will address that logic in more detail in a subsequent lesson on loops.*

The output

Listing 4 below shows the output produced by the code in Listing 3 above. *(Color was added for emphasis.)*

```
Show nesting
KEY     VALUE
b       {3: 60, 2: 50, 1: 40}
  key   value
   3    60
   2    50
   1    40
KEY     VALUE
c       {3: 90, 2: 80, 1: 70}
  key   value
   3    90
   2    80
   1    70
KEY     VALUE
a       {3: 30, 2: 20, 1: 10}
```

```
  key    value
   3     30
   2     20
   1     10

Listing 4
```

Why red and blue?

Although the print statements in Listing 3 above didn't produce color in the output, I added color to help you correlate the output with the code.

The three blue print statements in the code produced the output shown in blue. Likewise, the single red print statement in the code produced the output shown in red.

Output from the outer loop

The three blue print statements are in the outer loop. Each of these print statements produced one line of output during each iteration. The outer loop iterated once for each of the three dictionaries nested in the dictionary object named **d4**. Hence, there are nine blue lines in the output.

The first blue print statement caused some column headers to be printed in uppercase *(this statement was executed three times)*.

The second blue print statement caused each of the keys *(b, c, a)* in the dictionary object named **d4** to be printed *(in random order)* along with the value associated with each of those keys.

The third blue print statement caused some more column headers to be printed in lowercase *(also executed three times)*.

Output from the inner loop

The red print statement in the inner loop in Listing 3 produced one line of output for each element in each nested dictionary element. Each line of output consisted of the key for that element and the value associated with that key. Hence, there are nine red lines in the output produced for the three nested dictionary objects, each of which has three elements.

Now let's put the nested **for** loop aside and consider a different topic.

Removing elements from a dictionary

The **del** statement can be used to remove an element from a dictionary as shown in Listing 5.

```
print '\n',"Show deletion/values/sort"
```

```
del d1[1]
del d2[2]
del d3[3]

Listing 5
```

Each of the three **del** statements in Listing 5 removes one element from a dictionary that is nested in the dictionary named **d4**. *(Note that even though these dictionaries have been nested in the dictionary named d4, they are still accessible using their names: d1, d2, and d3.)*

Each of the three dictionaries nested in **d4** contained three elements before the three **del** statements were executed, and contained only two elements after the three **del** statements were executed.

Getting and sorting keys

The code shown in Listing 6 invokes the **keys()** method to get a list of keys for the dictionary named **d4**. It stores a reference to that list in **L1**. Then it invokes the **sort()** method to sort those keys into alphanumeric order. *(Alphanumeric order is like alphabetic order, but also including an ordering for numbers and special characters such as punctuation marks.)*

```
L1 = d4.keys()
L1.sort()

Listing 6
```

When the list produced by the code in Listing 6 is used to iterate on the dictionary, the results should no longer be in random order. Rather, they should be in alphanumeric order.

Display the dictionary contents

The code in Listing 7 below is used to display the contents of the dictionary. This code is similar to the code in the Listing 3 shown earlier with some important differences.

```
for x in L1:
  print "KEY",'\t',"VALUE"
  print x,'\t',d4[x]
  print x," values =",d4[x].values()
  print "  key",'\t',"value"
  L2 = d4[x].keys()
  L2.sort()
  for y in L2:\
              print "  ",y,'\t',d4[x][y]

Listing 7
```

Differences in display code

The primary differences between the code in Listing 7 above and the code in Listing 3 shown earlier is:

- The keys for the dictionary named **d4** are obtained, sorted, and stored in a list named L1 *(shown in Listing 6)* outside the outer loop. The sorted list is used for iteration control in the outer **for** loop in Listing 7.
- An extra print statement *(shown in blue)* is in the outer loop. This statement illustrates the use of the **values()** method to obtain and display a list of the values contained in each nested dictionary.
- A list of keys for each nested dictionary is obtained and sorted *(shown in red)*. The sorted list is used for iteration control in the inner **for** loop.

The output produced by the code in Listing 7 above is shown in Listing 8 below *(color added for emphasis)*.

```
Show deletion/values/sort
KEY     VALUE
a       {3: 30, 2: 20}
a  values = [30, 20]
  key   value
   2    20
   3    30
KEY     VALUE
b       {3: 60, 1: 40}
b  values = [60, 40]
  key   value
   1    40
   3    60
KEY     VALUE
c       {2: 80, 1: 70}
c  values = [80, 70]
  key   value
   1    70
   2    80


Listing 8
```

Analyze the output

The important things to note about the output shown in Listing 8 are:

- The blue lines of output correspond to the blue statement in the code shown in Listing 7.
- The data is displayed in order of ascending keys. The outer loop iterates on the keys in the order a, b, and c. The inner loop iterates on the keys in the order 1, 2, and 3. *(Note however that there are some missing keys in the data displayed by the inner loop as a result of the application of the del statement earlier.)*

Key membership testing

The **has_key()** method can be used to determine if the list of keys for a dictionary contains a particular key. This is illustrated in Listing 9.

```
print '\n',"Show membershp test"
cnt = 0
while cnt < 5:
  if d1.has_key(cnt):
    print cnt,'\t',d1[cnt]
    cnt = cnt + 1
  else:
    print "No key matches ",cnt
    cnt = cnt + 1

Listing 9
```

The logic in the code in Listing 9 is straightforward. The blue line *(that begins a while loop)* causes the lines below it to be executed for values of **cnt** equal to 0, 1, 2, 3, and 4 *(cnt less than 5)*. The code following the blue line is an **if-else** statement.

During each of the five iterations of the **while** loop, the dictionary named **d1** is tested in the red line *(an if statement)* to determine if it contains one of the keys 0, 1, 2, 3, and 4.

If the dictionary does contain a particular key, the code following the red line displays the key and the value associated with the key. Otherwise *(else)*, the code following the green line displays the statement **No key matches** followed by the test key.

The output

The output produced by the code in Listing 9 above is shown in Listing 10 below. As you can see, matches were found for keys 2 and 3. No matches were found for keys 0, 1, and 4. You should be able to confirm this by examining the output shown earlier in Listing 8, which shows the following key:value pairs for the first dictionary nested in the dictionary named **d4**: `{3: 30, 2: 20}`.

```
Show membershp test
No key matches  0
No key matches  1
2       20
3       30
No key matches  4


Listing 10
```

What's Next?

That wraps up the lessons on dictionaries. The next several lessons will concentrate on operators and flow of control.

Review

1. True or false? A dictionary can contain any kind of element other than another dictionary.

Ans: False. A dictionary can contain elements that are themselves dictionaries. This leads to the concept of nested dictionaries.

2. What is the name of the method that can be used to obtain a list of the keys of a dictionary?

Ans: The **keys()** method can be used to obtain a list of the keys in a dictionary.

3. True or false? When the **keys()** method is used to obtain a list of keys, the keys are in sorted order.

Ans: False. Dictionary keys are purposely randomized, and this is reflected in the list of keys obtained using the **keys()** method.

4. How can you obtain a list of dictionary keys in sorted order?

Ans: You can apply the **sort()** method to the list of keys.

5. What statement can be used to remove elements from a dictionary?

Ans: A **del** statement can be used to remove elements to a dictionary.

6. When the **del** statement is used to remove an element from a dictionary, the statement is applied to the value of the element.

Ans: False. The **del** statement is applied to the element's key.

7. What method can be used to determine if a particular element exists in a dictionary?

Ans: The **has_key()** method can be used to determine if a particular element is a member of a dictionary.

8. Write a program that illustrates how to create and initialize a dictionary with a list and a tuple.

Ans: See Listing 11 below.

```
# File Dict10.py
# Rev 08/08/00
# Copyright 2000, R. G. Baldwin
```

```
# Illustrates initializing a
#  dictionary
#------------------------------
# Create initialized dictionary
d1 = {5:[6,7,8],"a":(1,2,3)}
# Display it
print d1,'\n'
```

Listing 11

9. Write a program that illustrates how to get a list of keys.

Ans: See Listing 12 below.

```
# File Dict12.py
# Rev 08/08/00
# Copyright 2000, R. G. Baldwin
# Illustrates getting a list of
#  keys
#------------------------------
d1 = {5:[6,7,8],"a":(1,2,3)}
print d1.keys()
```

Listing 12

10. Write a program that illustrates how to iterate on a list of keys.

Ans: See Listing 13 below.

```
# File Dict14.py
# Rev 08/06/00
# Copyright 2000, R. G. Baldwin
# Illustrates iterating on a key
#  list from a dictionary
#------------------------------
# Create initialized dictionary
d1 = {5:"number",\
      "a":"string",\
      (1,2):"tuple"}

# Iterate on a key list
print "Dictionary contents"
for x in d1.keys():
  print x,':',d1[x]
```

Listing 13

Listing of Sample Program

A complete listing of the program is shown in Figure 14.

```
# File Dict08.py
# Rev 12/23/00
# Copyright 2000, R. G. Baldwin
# Illustrates
#  Nested dictionary
#  Sorting key list
#  Deleting items
#  has_key() membership
#---------------------------------//
print "Show nesting"
d1 = {1:10,2:20,3:30}
d2 = {1:40,2:50,3:60}
d3 = {1:70,2:80,3:90}
d4 = {"a":d1,"b":d2,"c":d3}

for x in d4.keys():
  print "KEY",'\t',"VALUE"
  print x,'\t',d4[x]
  print "  key",'\t',"value"
  for y in d4[x].keys():\
            print "  ",y,'\t',d4[x][y]

# Delete some items
# Sort the keys
print '\n',"Show deletion/values/sort"
del d1[1]
del d2[2]
del d3[3]
L1 = d4.keys()
L1.sort()
for x in L1:
  print "KEY",'\t',"VALUE"
  print x,'\t',d4[x]
  print x," values =",d4[x].values()
  print "  key",'\t',"value"
  L2 = d4[x].keys()
  L2.sort()
  for y in L2:\
            print "  ",y,'\t',d4[x][y]

print '\n',"Show membershp test"
cnt = 0
while cnt < 5:
  if d1.has_key(cnt):
    print cnt,'\t',d1[cnt]
    cnt = cnt + 1
  else:
    print "No key matches ",cnt
    cnt = cnt + 1
```

Listing 14