livelessons

video instruction from technology experts

7+ HOURS OF VIDEO INSTRUCTION

Python Fundamentals

Wesley J. Chun

mylivelessons.com
informit.com/ph
corepython.com

PRENTICE HALL

Library of Congress Control Number: 2008941453

Visit us on the Web: informit.com/ph

Corporate and Government Sales

The publisher offers excellent discounts on this LiveLesson when ordered in quantity for bulk purchases or special sales, which may include custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside the United States, please contact: International Sales, international@pearsoned.com.

Warning and Disclaimer

Feedback Information

At Pearson, our goal is to create in-depth technical products of the highest quality and value. Each product is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community. Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this LiveLesson, or otherwise alter it to better suit your needs, you can contact us through e-mail at mylivelessons@pearsoned.com. Please make sure to include the title and ISBN in your message.

We greatly appreciate your assistance.

Trademark Acknowledgments

ISBN-13: 978-0-13-714341-2
ISBN-10: 0-13-714341-9

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2008

Publisher
Paul Boger

Editor-in-Chief
Mark L. Taub

Executive Editor
Debra Williams Cauley

Video Project Manager
John A. Herrin

Editorial Assistant
Kim Boedigheimer

Managing Editor
John Fuller

Project Editor
Anna V. Popick

Copy Editor
Barbara Wood

Multimedia Developer
Eric Strom, Pearson Video

Production Services Designer
Gary Adair

Preface

Welcome to *Python Fundamentals* LiveLessons DVD. This is a complete video course that can be used to bring you up to speed with Python. It is intended to complement and is adapted from the book *Core Python Programming, Second Edition* (ISBN-10: 0-13-226993-7, ISBN-13: 978-0-13-226993-3), also by yours truly.

This workbook provides auxiliary material to augment or supplement the lessons on the DVD and provides some exercises so you can gauge how well you absorbed the material. The exercises either come directly from *Core Python Programming* or are adapted from it, and I recommend that book for more challenging problems than you will find here. A problem prefixed with (CPPx-y) means the exercise is either adapted or taken directly from *Core Python Programming* Exercise *y* in Chapter *x*. I also describe for each lesson which chapters from the book are the most applicable. Section headings beginning with an asterisk (*) cover advanced material and are optional.

Conventions

I introduce new technical terms using *italics*. URLs are formatted as http://links. `Monospacing` is used for all code in both the presentation and booklet. Code blocks and interactive interpreter examples are set in `monospaced` blocks, like so:

```
>>> print 'Hello World!'
Hello World!
```

Book Resources

A full copy of the entire presentation, plus a bonus appendix, is accessible directly on the DVD. You will also find all of the main code examples from *Core Python Programming* on the DVD. At the time of this writing, there are no plans to create a separate Web site for this LiveLessons course, so I recommend that you visit either the *Core Python Programming* book's Web site, http://corepython.com, or the main Live-Lessons site, http://mylivelessons.com, for additional information. The author can be reached via email at pythonfun@yahoo.com.

lesson 4

Standard Types

Lesson 4 is one of the longest lessons in this LiveLessons video course, so we recommend you break it up and view different segments at a time rather than watching it all in one sitting. It corresponds to Chapters 5–7 in *Core Python Programming* and covers almost all of Python's standard types.

Standard Types

The primary types we want you to focus on are integers (`int`), floating-point numbers (`float`), strings (`str`), lists (`list`), and dictionaries (`dict`). These are the primary data types that you will be using in your applications. One word of caution about the differences between Python versions 2.x and 3.x is that the default string types are ASCII in 2.x but Unicode in 3.x. Python 2.x also features Unicode strings but under a different name; likewise, Python 3.x also features ASCII strings, so watch out for the double name changes!

The standard types are broken down into three main categories: numbers, sequences, and hashing types. Numbers are fairly straightforward, sequences are similar to arrays and ordered starting at index 0, and hashing types are those in which the primary access consists of a hashed value, i.e., dictionary keys or set values.

Numbers

Python has several numeric types. The primary three are integers, floating-point numbers, and complex numbers. We look at `int`s and `float`s but not complex—if you are familiar with what they are, you can just look up how to use them in *Core Python Programming* or the official documentation. If you are not, then you probably won't need them, so no harm, no foul.

You may not think of Booleans (`bool`) as a numeric type because of their only two possible values of `True` and `False`; however, when used in a numeric context, they take on the values 1 and 0 respectively. (We began this lesson by describing how all Python objects have some Boolean value, and how `False` is generally represented by some numeric zero or empty container while all other values are `True`.)

Older versions of Python also featured a long integer type (`long`), but those have since been merged into the normal `int` type that we are familiar with. You may, from time to time, see old code and/or output that shows a trailing *L* representing a `long`, e.g., `99999999999999L`.

There are also several numeric types that aren't directly supported by the interpreter, requiring you to import specific modules to use them. They include the `Decimal` type, found in the `decimal` module, as well as the `Fraction` type, found in the `fractions` module.

Sequences

Python has three sequence types: strings (`str`), lists (`list`), and tuples (`tuple`). Of these, you will use tuples the least. Sequences are ordered containers, similar to arrays, with indices that start counting at zero. One of the most common operations you can perform on sequences is to take a subset of one or more elements from them. The terminology for doing this in Python is *slicing*.

Slicing

Whether you want only one element or perhaps more, the syntax still involves using the square brackets (`[ ]`) and providing a single element's index value or a starting-ending index pair for multiple elements. In the latter case, you separate both indices using a colon (`:`).

In this lesson we teach you multiple element slicing with indices `i` and `j`, stating that the slice will include elements starting at index `i` and take all subsequent elements up to but not including the element located at index `j`. You will find that taking something from a starting value and going up to but not including an ending value to be a common theme in Python, not just with slicing. Finally, you should already be able to figure that slicing can be accomplished with all sequence types, not just strings, even though the lesson's examples use strings.

Extended Slice Syntax

One aspect of slicing that we do not have time to cover in the video lesson is the extended slice syntax. Normally when taking a slice, the elements that are "sliced out" of the sequence are consecutive or adjacent to each other. Using the extended slicing syntax, you can specify a different "step" value. The syntax utilizes a *second* colon:

```
sequence[i:j:k]
```

The indices `i` and `j` are exactly the same as in a standard slice, but the step value `k` indicates how many and in which direction to choose the next index value to slice. For example, you can reverse a slice or even skip every other element quite easily:

```
In [1]: s = 'Python'

In [2]: s[::-1]
Out[2]: 'nohtyP'

In [3]: s[::2]
Out[3]: 'Pto'
```

Since our string is so short, we just leave `i` and `j` out, meaning take from the beginning or the end of the string, respectively. (Note that `s == s[::1]`.) Finally, we repeat again that although these examples use a string, any sequence type can be sliced.

Strings

Strings are, by default, ASCII in Python 2.x and Unicode in 3.x. This is important to note if you are going to migrate code or are going to start coding in 3.x after programming in any prior versions. In Python 2.x, Unicode strings are `unicode` while in 3.x, ASCII strings are `bytes`.

In the next lesson you will learn about mutability, and whether objects are mutable (their values can be changed) or immutable (their values cannot be changed). All string types are immutable. One interesting twist is that a new mutable string type will be introduced in 3.x (and starts in 2.6). It is called `bytearray`. Take a look at Table 4-1, which outlines the names and types each string type represents as well as its mutability. (The table also appears in the slides that accompany the video portion of the lesson.)

Table 4-1 Strings and Bytes

String Type	**2.x**	**3.x**	**Mutable?**[a]
ASCII Strings	`str ("")`	`bytes (b"")`	no
Unicode Strings	`unicode (u"")`	`str ("")`	no
Mutable Byte Array	*N/A*	`bytearray`	yes

a. Able to change its value.

In this lesson we also discuss those features that are unique to strings from other sequence types, such as string formatting, ASCII and Unicode issues, user input and output, raw strings, and triple quotes.

Lists

Lists are another extremely popular Python data type. They are versatile, flexible, robust, and heterogeneous (meaning they can hold objects of different types, unlike more traditional array types). One important feature of lists is that they are mutable, meaning you are allowed to change their values. (You will understand why this is important in the next lesson, but take our word for it for now.) Another type you would expect to be mutable is strings, but they are not (unless you use a special string type called `bytearray`). Lists are also resizable; you can grow and shrink them at will, according your application's needs.

List Comprehensions

Lists are very popular with Python programmers; because of this, it is advantageous to be able to create a list with just a single line of code containing the logic that defines the contents of the list (if that is possible). One common pattern used to build a list in the "old days"—meaning Python 1.6 and older—was to set a list to be initially empty (`[ ]`), then go into a loop calling the list `append()` method to add one element at a time to the list.

List comprehensions (or "listcomps" for short) is a feature from the Haskell programming language that helps Python coders avoid setting a blank list, followed by many "dot appends." Its syntax embeds an expression and a `for` loop as well as an optional conditional that helps determine whether an object becomes part of the created list. Best of all, this syntax takes up only one line of Python code.

In this lesson we show three different examples of listcomps, demonstrating both the syntax as well as the logic involved that results in a list being created with the exact contents as directed. Yes, two of them are more math-flavored, but we hope that you will get the hint. The final one is completely different but still should be simple enough for you to grasp, even as a beginner to the language. We repeat them here so that you can study them as well as their output outside of watching the video portion of the lesson:

```
>>> [i//3 for i in range(10)]
[0, 0, 0, 1, 1, 1, 2, 2, 2, 3]
```

We haven't covered `for` loops in detail, but based on the quick demo from Lesson 2 and perhaps some light reading in *Core Python Programming*, you should have some sort of idea that the `for` loop above counts to ten. (In reality, it only counts from zero to nine, but you get the idea.) Wouldn't it be great if we could create a list of the numbers zero through nine but all floor-divided by three? With listcomps it's a snap, as you can see from the code above.

The main point of this example is to show that you can apply an expression (some math formula, function call, etc.) to a sequence and get a new sequence with the results that you expect. Armed with this knowledge, we take it one step further with the second example:

```
>>> [i*3 for i in range(20) if i % 2 == 0]
[0, 6, 12, 18, 24, 30, 36, 42, 48, 54]
```

Before we apply an expression to a sequence, we can filter the final elements chosen using an `if` statement immediately following the `for` loop. In this case, we take only the even numbers between zero and 19, and of those that meet our criteria, we multiply them by three.

There is certainly nothing wrong with *not* applying any functionality to the chosen values, as in this slightly altered IPython snippet:

```
In [2]: [i for i in range(20) if i % 2]
Out[2]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

We changed the expression to a "NOP" (no operation) so it does nothing other than use the selected elements that meet the criteria, which we changed to odd numbers. We didn't have room to use this example in the video part of the lesson and hope that it serves as a good example here.

```
f = open('test.txt', 'r')
data = [line.strip() for line in f]
f.close()
```

The final listcomp example is different from the other two. We have not covered any material related to dealing with files in Python, yet here we are, exposing you to some code that opens a file for read, iterates through each line, stripping it of any leading or trailing whitespace, and closes the file. All the while we expect you, a new Python developer, to be able to read this code without too much difficulty, asking only for a little bit of common sense and imagination along with your prior programming experience.

One thing that we missed telling you is that in the code above, there is no data structure that represents all the lines of a file. If we had used the file `readlines()` method, we would have, but we didn't. Instead, we read in each line (as a string), one at a time using the listcomp `for` loop. We then use the string `strip()` method to remove the whitespace, and when the file iteration is complete, we have an entire list of all the lines in the file without any leading or trailing whitespace. Oh, yes, and it's always a good idea to close the file when you are done with it.

Generator Expressions

If you are someone who is always thinking ahead or looking for any problems with what you've been told or taught, you may already be thinking along our lines. While listcomps bring much flexibility to the Python programming environment, there is one caveat that you should be aware of when using them: They can take up a lot of memory unintentionally.

Think about our final listcomp. What if you opened an Apache log file that is 150MB in size, made up of about a million lines of data? It's probably not a good idea, regardless of programming language, to create a data structure with that number of elements very often.

Quite often, your intention is to create a sequence that will be iterated over in a `for` loop anyway. If you could create an object that can be looped over without necessarily creating a full list to hand back to you, wouldn't you rather use it than a listcomp? Of course! So this is why generator expressions (or "genexps" for short) were created.

Their syntax is nearly identical to that of lists, the only difference being that instead of using square brackets, genexps are created using regular parentheses. We didn't have room to show you an example during the video portion of the lesson, but it wasn't necessary because of how closely they resemble listcomps:

```
f = open('test.txt', 'r')
data = (line.strip() for line in f)
f.close()
```

This looks just like our example above, the primary difference being that we do not have a list that is composed of all the stripped lines of the file. Instead, we have an iterator (generator expression object) that you can loop over:

```
for line in data:
    print line
```

Looks good now, right? Not quite yet. This memory conservation technique is often referred to as "lazy evaluation." What does that mean? Lazy or delayed evaluation means that an operation isn't performed until the data is actually needed at runtime. So in the example above, the stripping of each line of the file doesn't actually happen until our second `for` loop where we are printing each (stripped) line.

The core problem with our solution is that we have already closed the file! We did that right after our generator expression was created, so if you actually try to run these five lines of code, you will get the following error (assuming you actually had a file called `test.txt`, as shown in our output here):

```
>>> f = open('test.txt', 'r')
>>> data = (line.strip() for line in f)
>>> f.close()
>>> for line in data:
...     print line
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <genexpr>
ValueError: I/O operation on closed file
```

Well, you can't win everything. We pay a small price but gain a significant advantage in memory conservation. The price we have to pay is that we cannot close the file until after we finish iterating through the data, as in:

```
f = open('test.txt', 'r')
data = (line.strip() for line in f)
```

```
for line in data:
    print line
f.close()
```

Tuples

At this stage of the game, you won't really see too much in the way of tuples. When you gain some more experience with Python, you will understand that they are just like lists but immutable, meaning that you cannot modify them. So after a while, you will just see them as "read-only lists." This is a common sentiment until you realize that they are really not to be used as data structures to solve problems with. That's not what their primary purpose or focus is.

Once you come to this realization, you will see them as mechanisms to get data to and from functions. Tuples can be used to pass a sequence to foreign functions that you don't want modified, as long as the tuple is accepted as one of the parameters you pass to the function and not merely used as the shopping bag for argument grouping or expansion, which you will learn about in Lesson 9.

Throughout most of Python's history, tuples have not had methods. They were designed this way because tuples are immutable objects, and mutable object methods are likely to modify themselves. However, strings seem to be doing just fine with methods, so starting with Python 2.6, a couple of familiar methods—`count()` and `index()`—were added to tuples.

Don't get too carried away, however. As mentioned earlier, you won't really be creating tuples with objects and manipulating them. That's what lists are for. You will probably be getting tuples back from function calls and extracting the elements of interest. They serve more of a background role.

Hashing

Of Python's container types, you will find lists and dictionaries to be the most commonly used. What one can't do, the other excels at. Lists are quite useful when you want an ordered array of objects to manipulate. But if order doesn't matter as much and you want faster data retrieval, you need a hashing type.

If you desire lookup via a key instead of an index, dictionaries are your tool; or, if you simply have a group of values that you want to randomly access, you may wish to consider a set type. Python sets come in two flavors, mutable and immutable, so you will have to determine which one you want based on whether or not those set values will change in your application.

Dictionaries

The first hashing type that we look at is the dictionary. It is a data structure that is similar to lists in that it is both mutable and is a container. That is about where the similarities end, however. Rather than using indices to access values, dictionaries (or "dicts" for short) use keys to access values.

Think of a coat check: You give up your jacket during check-in and are given a tag with which to recover your item later. With dictionaries, you provide both the key and the value, and the dictionary will save it off for you. You can then retrieve that value later by providing the key again.

Dictionaries are at the very heart of the Python language. They not only are used by users but are also present in management of Python objects. What we mean by that is that after spending a good amount of time developing in Python, you will begin to see a special attribute named `__dict__` pop up everywhere.

This special value represents all attributes belonging to an object (with the key being the attribute name and the value being the attribute value).

Accessing Values

One aspect of dictionary access that we did not get a chance to cover in the video portion of the lesson is that there are many ways that users (programmers like you) can access dictionary values.

Let's say, as an example, that we have a key named `foo` with a value `bar` in the dictionary `d`. To pull this value out of the dictionary, you would issue

```
d['foo']
```

Simple enough. However, there are situations where we do not know whether a key is in the dictionary yet or still. If the key does not exist, we get an error:

```
>>> d['foo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'foo'
```

A `KeyError` means that you tried to access a key that is not currently part of the dictionary. Either it hasn't been created yet, or it was removed. If your entire dictionary doesn't exist yet, you still get an error, but a different one:

```
>>> d['foo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
```

A `NameError` simply means that you haven't assigned any object to a variable yet; in other words, you tried to use (or access the value of) a variable that doesn't exist at the moment.

To combat this, because keys are likely to be variables at runtime, you can ask whether a key is in the dictionary or not:

```
>>> 'foo' in d
False
```

Only when this comparison results in `True` should you try to access the value with `d['foo']`. When in doubt, there are two other alternatives. The first one is the `get()` dictionary method.

The `get()` method has one obvious advantage over direct key access (`d['foo']`): You are allowed to provide a default value *in case* a key is not in a dictionary:

```
>>> d.get('foo', '(N/A)')
'(N/A)'
```

In the above example, if `d['foo']` does exist, you get its value as usual, but if it doesn't, you get the default, given here as `'(N/A)'`. The behavior of our `get()` call can be represented by this Python conditional expression (Python 2.5 and newer only):

```
d['foo'] if 'foo' in d else '(N/A)'
```

Even if you *don't* have or pass in a default value, at least it won't give an error—it returns `None`:

```
>>> d.get('foo')
>>>
```

Using `get()` is seen as an "upgrade" to straight key access because it does the same thing but loses one possibility of error. Another alternative that represents a further upgrade *on top of* `get()` is the `setdefault()` method.

```
>>> d.setdefault('foo', 'bar')
'bar'
```

This takes it one step further. The signature of this method—its arguments—is similar to that of `get()` where you pass in the key and a value, but instead of the default value being returned if the key doesn't exist, it *sets* that value with the associated key in the dictionary. If it *does* exist, however, the second value is ignored, and you get back the value that's currently associated with the key.

```
>>> d.setdefault('foo', 1)
>>> 'bar'
```

Also like `get()`, `setdefault()` doesn't require a second argument. If the key is nonexistent, it will be added to the dictionary with a default value of `None`.

Sets

The other hashing type discussed in this lesson is sets. These are the same sets that you studied in your mathematics class back in high school or college. A set is just a finite group of objects. They don't have to be of the same type, and they are not ordered. Because they are hashed types, there cannot be duplicates either.

```
>>> s = set()
>>> s
set([])
>>> s.add(1)
>>> s.add('bar')
>>> s
set([1, 'bar'])
>>> for item in s:
...     print item
...
1 bar
```

The syntax above is Python 2.x. Due to the similarities of sets to dictionaries, a shorthand notation using the familiar Python dictionary braces was added to the 3.x syntax—these are known as *set literals*:

```
>>> t = set()   # s = {} still reserved for dicts
>>> t.add('foo')
>>> t.add(0)
>>> t
```

```
{0, 'foo'}
>>> s | t       # union operation
{0, 1, 'foo', 'bar'}
```

In the video lesson we demonstrate performing the intersection and union operations using set methods, but in the last part of the example just above, we use the union operator (|) instead. Similarly to lists and list comprehensions, the need to be able to create a set based on a single line of logic led to the creation of *set comprehensions* in Python 3.x, which take advantage of the curly brace syntax:

```
>>> odd = {i for i in range(20) if i % 2}
>>> odd
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

Sorting Dictionary Keys or Set Values

Internally (and externally, as shown above), sets are very similar to dictionaries. At the programmer's level, the major difference is that rather than key-value pairs, you have only values. In either case, they both involve the hashed objects; for dictionaries, they're the keys, and for sets, the values.

Neither dictionary keys nor set values are ordered; this is due to the nature of hashed values and fast lookup. You can always manually sort them by calling the `sorted()` built-in function, i.e., `sorted(d)` for our earlier dictionary `d`.

Operators and Built-in Functions

Throughout this entire lesson as we introduce Python's standard types, we group them together based on similarity, i.e., numbers, sequences, and hashing types. For each group as well as each individual type, we cover the various operators and built-in functions (BIFs) as well as built-in methods (BIMs) that apply. We began the lesson with a higher-level view, describing operators and BIFs that apply to most Python types, as shown in Table 4-2.

Table 4-2 Standard Type Operators and Functions

Operator/Function	**Description**	**Result**[a]
Built-in and factory functions		
`cmp(`*`obj1, obj2`*`)`	Compares two objects	`int`
`repr(`*`obj`*`)`	Evaluatable string representation	`str`
`str(`*`obj`*`)`	Printable string representation	`str`
`type(`*`obj`*`)`	Object type	`type`
Value comparisons		
<	Less than	`bool`
>	Greater than	`bool`

continues

Table 4-2 Standard Type Operators and Functions *(Continued)*

Operator/Function	Description	Result[a]
Value comparisons		
`<=`	Less than or equal to	`bool`
`>=`	Greater than or equal to	`bool`
`==`	Equal to	`bool`
`!=`	Not equal to	`bool`
Object comparisons		
`is`	The same as	`bool`
`is not`	Not the same as	`bool`
Boolean operators		
`not`	Logical negation	`bool`
`and`	Logical conjunction	`bool`
`or`	Logical disjunction	`bool`

a. Boolean comparisons return either `True` or `False`.

We then discussed Python's numeric types and, in a similar fashion, briefly introduced you to the operators and BIFs that apply only to numbers. These are listed in Table 4-3.

Table 4-3 Operators and Built-in Functions for All Numeric Types

Operator/Built-in	Description	int	long	float	complex	Result[a]
`abs()`	Absolute value	•	•	•	•	*number*[a]
`chr()`	Character	•	•			`str`
`coerce()`	Numeric coercion	•	•	•	•	`tuple`
`complex()`	Complex factory function	•	•	•	•	`complex`
`divmod()`	Division/modulo	•	•	•	•	`tuple`
`float()`	Float factory function	•	•	•	•	`float`
`hex()`	Hexadecimal string	•	•			`str`
`int()`	Int factory function	•	•	•	•	`int`
`oct()`	Octal string	•	•			`str`
`ord()`	Ordinal			(string)		`int`
`pow()`	Exponentiation	•	•	•	•	*number*
`round()`	Float rounding			•		`float`

continues

Table 4-3 Operators and Built-in Functions for All Numeric Types *(Continued)*

Operator/Built-in	**Description**	**int**	**long**	**float**	**complex**	**Result**[a]
+[b]	No change	•	•	•	•	*number*
-[b]	Negation	•	•	•	•	*number*
~[b]	Bit inversion	•	•			`int/long`
**[c]	Exponentiation	•	•	•	•	*number*
*	Multiplication	•	•	•	•	*number*
/	Classic or true division	•	•	•	•	*number*
//	Floor division	•	•	•	•	*number*
%	Modulo/remainder	•	•	•	•	*number*
+	Addition	•	•	•	•	*number*
-	Subtraction	•	•	•	•	*number*
<<	Bit left shift	•	•			`int/long`
>>	Bit right shift	•	•			`int/long`
&	Bitwise AND	•	•			`int/long`
^	Bitwise XOR	•	•			`int/long`
|	Bitwise OR	•	•			`int/long`

a. A result of "number" indicates any of the numeric types, perhaps the same as the operands.
b. Unary operator.
c. ** has a unique relationship with unary operators; see Section 5.5.3 and Table 5.3 in *Core Python Programming*.

Next, we looked at sequences and their operators, BIFs, and BIMs, as shown in Table 4-4.

Table 4-4 Sequence Type Operators, Built-in Functions, and Methods

Operator, Built-in Function, or Method	**`str`**	**`list`**	**`tuple`**
`[]` (list creation)		•	
`()`			•
`""`	•		
`append()`		•	
`capitalize()`	•		
`center()`	•		
`chr()`	•		
`cmp()`	•	•	•

continues

Table 4-4 Sequence Type Operators, Built-in Functions, and Methods *(Continued)*

Operator, Built-in Function, or Method	`str`	`list`	`tuple`
`count()`	•	•	•[a]
`decode()`	•		
`encode()`	•		
`endswith()`	•		
`expandtabs()`	•		
`extend()`		•	
`find()`	•		
`format()`	•		
`hex()`	•		
`index()`	•	•	•[a]
`insert()`		•	
`isdecimal()`	•		
`isdigit()`	•		
`islower()`	•		
`isnumeric()`	•		
`isspace()`	•		
`istitle()`	•		
`isupper()`	•		
`join()`	•		
`len()`	•	•	•
`list()`	•	•	•
`ljust()`	•		
`lower()`	•		
`lstrip()`	•		
`max()`	•	•	•
`min()`	•	•	•
`oct()`	•		
`ord()`	•		
`pop()`		•	
`raw_input()` [or `input()` for Python 3.x]	•		

continues

Table 4-4 Sequence Type Operators, Built-in Functions, and Methods *(Continued)*

Operator, Built-in Function, or Method	`str`	`list`	`tuple`
`remove()`		•	
`replace()`	•		
`repr()`	•	•	•
`reverse()`		•	
`rfind()`	•		
`rindex()`	•		
`rjust()`	•		
`rstrip()`	•		
`sort()`		•	
`split()`	•		
`splitlines()`	•		
`startswith()`	•		
`str()`	•	•	•
`strip()`	•		
`swapcase()`	•		
`split()`	•		
`title()`	•		
`tuple()`	•	•	•
`type()`	•	•	•
`upper()`	•		
`zfill()`	•		
`.` (attributes)	•	•	
`[]` (slice)	•	•	•
`[:]`	•	•	•
`*`	•	•	•
`%`	•		
`+`	•	•	•
`in`	•	•	•
`not in`	•	•	•

a. New in Python 2.6.

Strings and lists are the most commonly used sequence types, and they have their own set of operators, BIFs, and BIMs, as charted in Tables 4-5, 4-6, 4-7, and 4-8.

Table 4-5 String Format Operator Conversion Symbols

Format Symbol	Conversion
`%c`	Character (integer [ASCII value] or string of length 1)
`%r`[a]	String conversion via `repr()` prior to formatting
`%s`	String conversion via `str()` prior to formatting
`%d / %i`	Signed decimal integer
`%u`[b]	Unsigned decimal integer
`%o`[b]	(Unsigned) octal integer
`%x`[b] / `%X`[b]	(Unsigned) hexadecimal integer (lower/UPPERcase letters)
`%e / %E`	Exponential notation (with lowercase `'e'`/UPPERcase `'E'`)
`%f / %F`	Floating-point real number (fraction truncates naturally)
`%g / %G`	The shorter of `%e` and `%f`/`%E%` and `%F%`
`%%`	Percent character (`%`) unescaped

a. New in Python 2.0; likely unique only to Python.
b. `%u/%o/%x/%X` of negative `int` will return a signed string in Python 2.4+.

Table 4-6 Format Operator Auxiliary Directives

Symbol	Functionality
`*`	Argument specifies width or precision
`-`	Use left justification
`+`	Use a plus sign (+) for positive numbers
<sp>	Use space-padding for positive numbers
`#`	Add the octal leading zero (`'0'`) or hexadecimal leading `'0x'` or `'0X'`, depending on whether `'x'` or `'X'` was used
`0`	Use zero-padding (instead of spaces) when formatting numbers
`%`	`'%%'` leaves you with a single literal `'%'`
`(`*var*`)`	Mapping variable (dictionary arguments)
m.n	*m* is the minimum total width and *n* is the number of digits to display after the decimal point (if applicable)

Table 4-7 String Type Built-in Methods

Method Name	Description
`string.capitalize()`	Capitalizes first letter of `string`
`string.center(width)`	Returns a space-padded `string` with the original `string` centered to a total of `width` columns
`string.count(str, beg=0, end=len(string))`	Counts how many times `str` occurs in `string`, or in a substring of `string` if starting index `beg` and ending index `end` are given
`string.decode(encoding='UTF-8', errors='strict')`[a]	Returns decoded string version of string; on error, default is to raise a `ValueError` unless `errors` is given with `'ignore'` or `'replace'`
`string.encode(encoding='UTF-8', errors='strict')`[b]	Returns encoded string version of string; on error, default is to raise a `ValueError` unless `errors` is given with `'ignore'` or `'replace'`
`string.endswith(str, beg=0, end=len(string))`[c]	Determines if `string` or a substring of `string` (if starting index `beg` and ending index `end` are given) ends with `str`; returns `True` if so, and `False` otherwise
`string.expandtabs(tabsize=8)`	Expands tabs in `string` to multiple spaces; defaults to 8 spaces per tab if `tabsize` not provided
`string.find(str, beg=0 end=len(string))`	Determines if `str` occurs in `string`, or in a substring of `string` if starting index `beg` and ending `index end` are given; returns `index` if found and –1 otherwise
`string.format(fmtstr, *t, **d)`[d]	Alternative to string format operator (`%`) to creating formatted strings using `fmtstr` with data from `t` and `d`
`string.index(str, beg=0, end=len(string))`	Same as `find()`, but raises an exception if `str` not found
`string.isalnum()`[b,c,e]	Returns `True` if `string` has at least 1 character and all characters are alphanumeric and `False` otherwise
`string.isalpha()`[b,c,e]	Returns `True` if `string` has at least 1 character and all characters are alphabetic and `False` otherwise
`string.isdecimal()`[c,e,f]	Returns `True` if `string` contains only decimal digits and `False` otherwise
`string.isdigit()`[c,e]	Returns `True` if `string` contains only digits and `False` otherwise
`string.islower()`[c,e]	Returns `True` if `string` has at least 1 cased character and all cased characters are in lowercase and `False` otherwise
`string.isnumeric()`[c,e,f]	Returns `True` if `string` contains only numeric characters and `False` otherwise
`string.isspace()`[c,e]	Returns `True` if `string` contains only whitespace characters and `False` otherwise
`string.istitle()`[c,e]	Returns `True` if `string` is properly "titlecased" (see `title()`) and `False` otherwise

continues

Table 4-7 String Type Built-in Methods *(Continued)*

Method Name	**Description**
`string.isupper()`[c,e]	Returns `True` if `string` has at least 1 cased character and all cased characters are in uppercase and `False` otherwise
`string.join(seq)`	Merges (concatenates) the string representations of elements in sequence `seq` into a string, with separator `string`
`string.ljust(width)`	Returns a space-padded `string` with the original string left-justified to a total of `width` columns
`string.lower()`	Converts all uppercase letters in `string` to lowercase
`string.lstrip()`	Removes all leading whitespace in `string`
`string.replace(str1, str2, num=string.count(str1))`	Replaces all occurrences of `str1` in `string` with `str2`, or at most `num` occurrences if `num` given
`string.rfind(str, beg=0, end=len(string))`	Same as `find()`, but search backwards in `string`
`string.rindex( str, beg=0, end=len(string))`	Same as `index()`, but search backwards in `string`
`string.rjust(width)`	Returns a space-padded `string` with the original string right-justified to a total of `width` columns
`string.rstrip()`	Removes all trailing whitespace of `string`
`string.split(str="", num=string.count(str))`	Splits `string` according to delimiter `str` (space if not provided) and returns list of substrings; split into at most `num` substrings if given
`string.splitlines( num=string.count('\n'))`[c,e]	Splits `string` at all (or `num`) NEWLINEs and returns a list of each line with NEWLINEs removed
`string.startswith(str, beg=0, end=len(string))`[c]	Determines if `string` or a substring of `string` (if starting index `beg` and ending index `end` are given) starts with substring `str`; returns `True` if so, and `False` otherwise
`string.strip([obj])`	Performs both `lstrip()` and `rstrip()` on `string`
`string.swapcase()`	Inverts case for all letters in `string`
`string.title()`[c,e]	Returns "titlecased" version of `string`, that is, all words begin with uppercase, and the rest are lowercase (also see `istitle()`)
`string.translate(str, del="")`	Translates `string` according to translation table `str` (256 chars), removing those in the `del` string
`string.upper()`	Converts lowercase letters in `string` to uppercase
`string.zfill(width)`	Returns original `string` left-padded with zeros to a total of `width` characters; intended for numbers, `zfill()` retains any sign given (less one zero)

a. New in Python 2.2.
b. Applicable to Unicode strings only in 1.6, but to all string types in 2.0.
c. Not available as a `string` module function in 1.5.2.
d. New in Python 2.6.
e. New in Python 2.1.
f. Applicable to Unicode strings only.

Table 4-8 List Type Built-in Methods

List Method	Operation
`list.append(obj)`	Adds *obj* to the end of *list*
`list.count(obj)`	Returns count of how many times *obj* occurs in *list*
`list.extend(seq)`	Appends contents of *seq* to *list*
`list.index(obj, i=0, j=len(list))`	Returns lowest index *k* where `list[k] == obj` and *i* <= *k* < *j*; otherwise `ValueError` raised
`list.insert(index, obj)`	Inserts *obj* into *list* at offset *index*
`list.pop(index=-1)`[a]	Removes and returns *obj* at given or last *index* from *list*
`list.remove(obj)`	Removes object *obj* from *list*
`list.reverse()`	Reverses objects of *list* in place
`list.sort(func=None, key=None, reverse=False)`	Sorts list members with optional comparison *func*tion; *key* is a callback when extracting elements for sorting, and if *reverse* flag is `True`, then list is sorted in reverse order

a. New in Python 1.5.2.

Finally, we looked at hashing types, dictionaries and sets, and *their* operators, BIFs, and BIMs, as detailed in Tables 4-9 and 4-10.

Table 4-9 Dictionary Type Methods

Method Name	Operation
`dict.clear`[a]`()`	Removes all elements of *dict*
`dict.copy`[a]`()`	Returns a (shallow[b]) copy of *dict*
`dict.fromkeys`[c]`(seq, val=None)`	Creates and returns a new dictionary with the elements of *seq* as the keys and *val* as the initial value (defaults to `None` if not given) for all keys
`dict.get(key, default=None)`[a]	For key *key*, returns value or *default* if *key* not in *dict* (note that *default*'s default is `None`)
`dict.has_key(key)`[d]	Returns `True` if *key* is in *dict*, `False` otherwise; partially deprecated by the **`in`** and **`not in`** operators in 2.2 but still provides a functional interface
`dict.items()`	Returns an iterable[e] of the (key, value) tuple pairs of *dict*
`dict.iter*`[f]`()`	`iteritems()`, `iterkeys()`, `itervalues()` are all methods that behave the same as their non-iterator counterparts but return an iterator instead of a list
`dict.keys()`	Returns an iterable[e] of the keys of *dict*
`dict.pop`[c]`(key[, default])`	Similar to `get()` but removes and returns `dict[key]` if `key` present and raises `KeyError` if `key` not in *dict* and *default* not given

continues

Table 4-9 Dictionary Type Methods *(Continued)*

Method Name	Operation
`dict.setdefault(key, default=None)`[g]	Similar to `get()`, but sets `dict[key]=default` if `key` is not already in `dict`
`dict.update(dict2)`[a]	Adds the key-value pairs of `dict2` to `dict`
`dict.values()`	Returns an iterable[e] of the values of `dict`

a. New in Python 1.5.
b. More information regarding shallow and deep copies can be found in Section 6.20 of *Core Python Programming*.
c. New in Python 2.3.
d. Deprecated in Python 2.2 and removed in Python 3.0; use `in` instead.
e. The iterable is a set view starting in Python 3.0 and a list in all previous versions.
f. New in Python 2.2.
g. New in Python 2.0.

Table 4-10 Set Type Operators, Functions, and Methods

Function/Method Name	Operator Equivalent	Description
All Set Types		
`len(s)`		Set cardinality: number of elements in `s`
`set([obj])`		Mutable set factory function; if `obj` given, it must be iterable, new set elements taken from `obj`; if not, creates an empty set
`frozenset([obj])`		Immutable set factory function; operates the same as `set()` except returns immutable set
	`obj in s`	Membership test: is `obj` an element of `s`?
	`obj not in s`	Non-membership test: is `obj` not an element of `s`?
	`s == t`	Equality test: do `s` and `t` have exactly the same elements?
	`s != t`	Inequality test: opposite of ==
	`s < t`	(Strict) subset test; `s != t` and all elements of `s` are members of `t`
`s.issubset(t)`	`s <= t`	Subset test (allows improper subsets): all elements of `s` are members of `t`
	`s > t`	(Strict) superset test: `s != t` and all elements of `t` are members of `s`
`s.issuperset(t)`	`s >= t`	Superset test (allows improper supersets): all elements of `t` are members of `s`
`s.union(t)`	`s | t`	Union operation: elements in `s` or `t`
`s.intersection(t)`	`s & t`	Intersection operation: elements in `s` and `t`

continues

Table 4-10 Set Type Operators, Functions, and Methods *(Continued)*

Function/Method Name	Operator Equivalent	Description
All Set Types		
`s.difference(t)`	`s - t`	Difference operation: elements in *s* that are not elements of *t*
`s.symmetric_difference(t)`	`s ^ t`	Symmetric difference operation: elements of either *s* or *t* but not both
`s.copy()`		Copy operation: return (shallow) copy of *s*
Mutable Sets Only		
`s.update(t)`	`s |= t`	(Union) update operation: members of *t* added to *s*
`s.intersection_update(t)`	`s &= t`	Intersection update operation: *s* only contains members of the original *s* and *t*
`s.difference_update(t)`	`s -= t`	Difference update operation: *s* only contains original members not in *t*
`s.symmetric_difference_update(t)`	`s ^= t`	Symmetric difference update operation: *s* only contains members of *s* or *t* but not both
`s.add(obj)`		Add operation: add *obj* to *s*
`s.remove(obj)`		Remove operation: remove *obj* from *s*; `KeyError` raised if *obj* not in *s*
`s.discard(obj)`		Discard operation: friendlier version of `remove()`—remove *obj* from *s* if *obj* in *s*
`s.pop()`		Pop operation: remove and return an arbitrary element of *s*
`s.clear()`		Clear operation: remove all elements of *s*

Exercises

Review

1. *Boolean Values.* The `bool` objects `True` and `False` obviously have Boolean values. What about all other Python objects? What are the rules determining whether an object has a `True` or `False` value? What about for objects *you* create?
2. *Division.* What is the difference between the `/` and `//` division operators?
3. *Strings.* What string types does Python have? Which one is the default type?
4. *Functions.* What are factory functions? How are they named?
5. *Numbers.* Why do some Python floats look odd, e.g., 1.1000000000000001?
6. *Sequences and Slicing.* In the lesson we demonstrate slicing using `s = "Python"`. Provide the slice of this string that will result in the substring `"on"`.
7. *Sequences and Mappings.* What is a sequence type? How is data accessed differently from dictionaries?

Coding

8. *(CPP6-5a)* ***Strings.*** Write code that takes a string and displays it one character at a time going forward and backward.

9. *(CPP5-3)* ***Standard Type Operators.*** Take test score input from the user and output letter grades according to the following grade scale/curve:
   A: 90–100
   B: 80–89
   C: 70–79
   D: 60–69
   F: <60

10. *(CPP5-6)* ***Arithmetic.*** Create a calculator application. Write code that will take two numbers and an operator in the format *N1 OP N2*, where *N1* and *N2* are floating-point or integer values, and *OP* is one of the following: +, -, *, /, %, **, representing addition, subtraction, multiplication, division, modulus/remainder, and exponentiation, respectively. Display the result of carrying out that operation on the input operands. Hint: You may use the string `split()` method, but you cannot use the `eval()` built-in function.