APPENDIX A

The Short Version

This is a minimal introduction to Python, based on my popular web tutorial, "Instant Python" (`http://hetland.org/writing/instant-python.html`). It targets programmers who already know a language or two, but who want to get up to speed with Python. For information on downloading and executing the Python interpreter, see Chapter 1.

The Basics

To get a basic feel for the Python language, think of it as pseudocode, because that's pretty close to the truth. Variables don't have types, so you don't need to declare them. They appear when you assign to them, and disappear when you don't use them anymore. Assignment is done with the = operator, like this:

```
x = 42
```

Note that equality is tested by the == operator.

You can assign several variables at once, like this:

```
x,y,z = 1,2,3
first, second = second, first
a = b = 123
```

Blocks are indicated through indentation, and *only* through indentation. (No `begin`/`end` or braces.) The following are some common control structures:

```
if x < 5 or (x > 10 and x < 20):
    print "The value is OK."

if x < 5 or 10 < x < 20:
    print "The value is OK."

for i in [1,2,3,4,5]:
    print "This is iteration number", i

x = 10
while x >= 0:
    print "x is still not negative."
    x = x-1
```

The first two examples are equivalent.

The index variable given in the for loop iterates through the elements of a list[1] (written with brackets, as in the example). To make an "ordinary" for loop (that is, a counting loop), use the built-in function range:

```
# Print out the values from 0 to 99, inclusive
for value in range(100):
    print value
```

The line beginning with # is a comment and is ignored by the interpreter.

Now you know enough (in theory) to implement any algorithm in Python. Let's add some *basic* user interaction. To get input from the user (from a text prompt), use the built-in function input:

```
x = input("Please enter a number: ")
print "The square of that number is", x*x
```

The input function displays the (optional) prompt given and lets the user enter any valid Python value. In this case, we were expecting a number. If something else (such as a string) is entered, the program would halt with an error message. To avoid that, you would need to add some error checking. I won't go into that here; suffice it to say that if you want the user input returned *verbatim* as a string (so that *anything* can be entered), use the function raw_input instead. If you wanted to convert an input string s to an integer, you could then use int(s).

Note If you want to input a string with input, the user must write the quotes explicitly. In Python, strings can be enclosed in either single or double quotes. In Python 3.0, the original input disappears, and raw_input is renamed input. See Appendix D for more on Python 3.0.

So, you have control structures, input, and output covered—now you need some snazzy data structures. The most important ones are *lists* and *dictionaries*. Lists are written with brackets, and can (naturally) be nested:

```
name = ["Cleese", "John"]
x = [[1,2,3],[y,z],[[[]]]]
```

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 writing the index in brackets after the list. (Note that the first element has index 0.)

```
print name[1], name[0] # Prints "John Cleese"
name[0] = "Smith"
```

1. Or any other iterable object, actually.

Slicing is almost like indexing, except that you indicate both the start and stop index of the result, with a colon (:) separating them:

```
x = ["SPAM","SPAM","SPAM","SPAM","SPAM","eggs","and","SPAM"]
print x[5:7] # Prints the list ["eggs","and"]
```

Notice that the end is noninclusive. If one of the indices is dropped, it is assumed that you want everything in that direction. In other words, the slice `x[:3]` means "every element from the beginning of `x` up to element 3, noninclusive" (well, element 3 is actually the fourth element, because the counting starts at 0). The slice `x[3:]` would, on the other hand, mean "every element in `x`, starting at element 3 (inclusive) up to, and including, the last one." For really interesting results, you can use negative numbers, too: `x[-3]` is the third element from the end of the list.

Now then, what about dictionaries? To put it simply, they are like lists, except that their contents aren't ordered. How do you index them then? Well, every element has a *key*, or a *name*, which is used to look up the element, just as in a real dictionary. The following example demonstrates the syntax used to create dictionaries:

```
phone = { "Alice" : 23452532, "Boris" : 252336,
          "Clarice" : 2352525, "Doris" : 23624643 }

person = { 'first name': "Robin", 'last name': "Hood",
           'occupation': "Scoundrel" }
```

Now, to get `person`'s occupation, you use the expression `person["occupation"]`. If you wanted to change the person's last name, you could write this:

```
person['last name'] = "of Locksley"
```

Simple, isn't it? Like lists, dictionaries can hold other dictionaries, or lists, for that matter. And naturally, lists can hold dictionaries, too. That way, you can easily make some quite advanced data structures.

Functions

Our next step is abstraction. You want to give a name to a piece of code and call it with a couple of parameters. In other words, you want to define a *function* (also called a *procedure*). That's easy. Use the keyword `def`, as follows:

```
def square(x):
    return x*x

print square(2) # Prints out 4
```

The return statement is used to return a value from the function.

When you pass a parameter to a function, you bind the parameter to the value, thus creating a new reference. This means that you can modify the original value directly inside the function, but if you make the parameter name refer to something else (rebind it), that change won't affect the original. This works just like in Java, for example. Let's take a look at an example:

```
def change(x):
    x[1] = 4

y = [1,2,3]
change(y)
print y # Prints out [1,4,3]
```

As you can see, the original list is passed in, and if the function modifies it, these modifications carry over to the place where the function was called. Note the behavior in the following example, however, where the function body *rebinds* the parameter:

```
def nochange(x):
    x = 0

y = 1
nochange(y)
print y # Prints out 1
```

Why doesn't y change now? Because you *don't change the value*! The value that is passed in is the number 1, and you can't change a number in the same way that you change a list. The number 1 is (and will always be) the number 1. What the example *does* change is what the parameter x *refers to*, and this does *not* carry over to the calling environment.

Python has all kinds of nifty things such as *named arguments* and *default arguments*, and can handle a variable number of arguments to a single function. For more information about this, see Chapter 6.

If you know how to use functions in general, what I've told you so far is basically what you need to know about them in Python.

It might be useful to know, however, that functions are *values* in Python. So if you have a function such as square, you could do something like the following:

```
queeble = square
print queeble(2) # Prints out 4
```

To call a function without arguments, you must remember to write doit() and not doit. The latter, as shown, only returns the function itself, as a value. This goes for methods in objects, too. Methods are described in the next section.

Objects and Stuff . . .

I assume you know how object-oriented programming works. Otherwise, this section might not make much sense. No problem—start playing without the objects, or check out Chapter 7.

In Python, you define classes with the (surprise!) `class` keyword, as follows:

```
class Basket:

    # Always remember the *self* argument
    def __init__(self, contents=None):
        self.contents = contents or []

    def add(self, element):
        self.contents.append(element)

    def print_me(self):
        result = ""
        for element in self.contents:
            result = result + " " + repr(element)
        print "Contains:" + result
```

Several things are worth noting in this example:

- Methods are called like this: `object.method(arg1, arg2)`.
- Some arguments can be *optional* and given a default value (as mentioned in the previous section on functions). This is done by writing the definition like this:

  ```
  def spam(age=32): ...
  ```

- Here, `spam` can be called with one or zero parameters. If it's called without any parameters, `age` will have the default value of 32.
- `repr` converts an object to its string representation. (So if `element` contains the number 1, then `repr(element)` is the same as `"1"`, whereas `'element'` is a literal string.)

No methods or member variables (attributes) are protected (or private or the like) in Python. Encapsulation is pretty much a matter of programming style. (If you *really* need it, there are naming conventions that will allow some privacy, such as prefixing a name with a single or double underscore.)

Now, about that short-circuit logic . . .

All values in Python can be used as logic values. Some of the more empty ones (such as `False`, `[]`, `0`, `""`, and `None`) represent logical falsity; most other values (such as `True`, `[0]`, `1`, and `"Hello, world"`) represent logical truth.

Logical expressions such as `a and b` are evaluated like this:

- Check if `a` is true.
- If it is *not*, then simply return it.
- If it *is*, then simply return `b` (which will represent the truth value of the expression).

The corresponding logic for `a or b` is this:

- If `a` is true, then return it.
- If it isn't, then return `b`.

This short-circuit mechanism enables you to use and and or like the Boolean operators they are supposed to implement, but it also enables you to write short and sweet little conditional expressions. For example, this statement:

```
if a:
    print a
else:
    print b
```

could instead be written like this:

```
print a or b
```

Actually, this is somewhat of a Python idiom, so you might as well get used to it.

Note In Python 2.5, actual conditional expressions were introduced, so you could, in fact, write this:

```
print a if a else b
```

The `Basket` constructor (`Basket.__init__`) in the previous example uses this strategy in handling default parameters. The argument `contents` has a default value of `None` (which is, among other things, false); therefore, to check if it had a value, you could write this:

```
if contents:
    self.contents = contents
else:
    self.contents = []
```

Instead, the constructor uses this simple statement:

```
self.contents = contents or []
```

Why don't you give it the default value of `[]` in the first place? Because of the way Python works, this would give all the `Basket` instances the same empty list as default contents. As soon as one of them started to fill up, they all would contain the same elements, and the default would not be empty anymore. To learn more about this, see the discussion about the difference between *identity* and *equality* in Chapter 5.

Note When using `None` as a placeholder as done in the `Basket.__init__` method, using `contents is None` as the condition is safer than simply checking the argument's Boolean value. This will allow you to pass in a false value such as an empty list of your own (to which you could keep a reference outside the object).

If you would like to use an empty list as the default value, you can avoid the problem of sharing this among instances by doing the following:

```
def __init__(self, contents=[]):
    self.contents = contents[:]
```

Can you guess how this works? Instead of using the same empty list everywhere, you use the expression `contents[:]` to make a copy. (You simply slice the entire thing.)

So, to actually make a `Basket` and to use it (to call some methods on it), you would do something like this:

```
b = Basket(['apple','orange'])
b.add("lemon")
b.print_me()
```

This would print out the contents of the `Basket`: an apple, an orange, and a lemon.

There are magic methods other than `__init__`. One such method is `__str__`, which defines how the object wants to look if it is treated like a string. You could use this in the basket instead of `print_me`:

```
def __str__(self):
    result = ""
    for element in self.contents:
        result = result + " " + repr(element)
    return "Contains:" + result
```

Now, if you wanted to print the basket `b`, you could just use this:

```
print b
```

Cool, huh?

Subclassing works like this:

```
class SpamBasket(Basket):
    # ...
```

Python allows multiple inheritance, so you can have several superclasses in the parentheses, separated by commas. Classes are instantiated like this: `x = Basket()`. Constructors are, as I said, made by defining the special member function `__init__`. Let's say that `SpamBasket` had a constructor `__init__(self, type)`. Then you could make a spam basket like this: `y = SpamBasket("apples")`.

If in the constructor of `SpamBasket`, you needed to call the constructor of one or more superclasses, you could call it like this: `Basket.__init__(self)`. Note that in addition to supplying the ordinary parameters, you must explicitly supply `self`, because the superclass `__init__` doesn't know which instance it is dealing with.

For more about the wonders of object-oriented programming in Python, see Chapter 7.

Some Loose Ends

Here, I'll quickly review a few other useful things before ending this appendix. Most useful functions and classes are put in *modules*, which are really text files with the file name extension `.py` that contain Python code. You can import these and use them in your own programs. For example, to use the function `sqrt` from the standard module `math`, you can do either this:

```
import math
x = math.sqrt(y)
```

or this:

```
from math import sqrt
x = sqrt(y)
```

For more information on the standard library modules, see Chapter 10.

All the code in the module/script is run when it is imported. If you want your program to be both an importable module and a runnable program, you might want to add something like this at the end of it:

```
if __name__ == "__main__": main()
```

This is a magic way of saying that if this module is run as an executable script (that is, it is not being imported into another script), then the function `main` should be called. Of course, you could do anything after the colon there.

And for those of you who want to make an executable script in UNIX, use the following first line to make it run by itself:

```
#!/usr/bin/env python
```

Finally, a brief mention of an important concept: *exceptions*. Some operations (such as dividing something by zero or reading from a nonexistent file) produce an error condition or *exception*. You can even make your own exceptions and raise them at the appropriate times.

If nothing is done about the exception, your program ends and prints out an error message. You can avoid this with a `try`/`except` statement, as in this example:

```
def safe_division(a, b):
    try:
        return a/b
    except ZeroDivisionError: pass
```

`ZeroDivisionError` is a standard exception. In this case, you *could* have checked if `b` was zero, but in many cases, that strategy is not feasible. And besides, if you removed the `try`/`except` statement in `safe_division`, thereby making it a risky function to call (called something like `unsafe_division`), you could still do the following:

```
try:
    unsafe_division(a, b)
except ZeroDivisionError:
    print "Something was divided by zero in unsafe_division"
```

In cases in which you *typically* would not have a specific problem, but it *might* occur, using exceptions enables you to avoid costly testing and so forth.

Well, that's it. Hope you learned something. Now go and play. And remember the Python motto of learning: use the source (which basically means read all the code you can get your hands on).

APPENDIX B

Python Reference

This is not a full Python reference by far—you can find that in the standard Python documentation (`http://python.org/doc/ref`). Rather, this is a handy "cheat sheet" that can be useful for refreshing your memory as you start out programming in Python. See Appendix D for changes in the language that are introduced in version 3.0.

Expressions

This section summarizes Python expressions. Table B-1 lists the most important basic (literal) values in Python; Table B-2 lists the Python operators, along with their precedence (those with high precedence are evaluated before those with low precedence); Table B-3 describes some of the most important built-in functions; Tables B-4 through B-6 describe the list methods, dictionary methods, and string methods, respectively.

Table B-1. *Basic (Literal) Values*

Type	Description	Syntax Samples
Integer	Numbers without a fractional part	`42`
Long integer	Large integer numbers	`42L`
Float	Numbers with a fractional part	`42.5`, `42.5e-2`
Complex	Sum of a real (integer or float) and imaginary number	`38 + 4j`, `42j`
String	An immutable sequence of characters	`'foo'`, `"bar"`, `"""baz"""`, `r'\n'`
Unicode	An immutable sequence of Unicode characters	`u'foo'`, `u"bar"`, `u"""baz"""`

Table B-2. *Operators*

Operator	Description	Precedence
`lambda`	Lambda expression	1
`or`	Logical or	2
`and`	Logical and	3

Continued

Table B-2. *Continued*

Operator	Description	Precedence
`not`	Logical negation	4
`in`	Membership test	5
`not in`	Negative membership test	5
`is`	Identity test	6
`is not`	Negative identity test	6
`<`	Less than	7
`>`	Greater than	7
`<=`	Less than or equal to	7
`>=`	Greater than or equal to	7
`==`	Equal to	7
`!=`	Not equal to	7
`|`	Bitwise or	8
`^`	Bitwise exclusive or	9
`&`	Bitwise and	10
`<<`	Left shift	11
`>>`	Right shift	11
`+`	Addition	12
`-`	Subtraction	12
`*`	Multiplication	13
`/`	Division	13
`%`	Remainder	13
`+`	Unary identity	14
`-`	Unary negation	14
`~`	Bitwise complement	15
`**`	Exponentiation	16
`x.attribute`	Attribute reference	17
`x[index]`	Item access	18
`x[index1:index2[:index3]]`	Slicing	19
`f(args...)`	Function call	20
`(...)`	Parenthesized expression or tuple display	21

Operator	Description	Precedence
`[...]`	List display	22
`{key:value, ...}`	Dictionary display	23
`` `expressions...` ``	String conversion	24

Table B-3. *Some Important Built-in Functions*

Function	Description
`abs(number)`	Returns the absolute value of a number.
`apply(function[, args[, kwds]])`	Calls a given function, optionally with parameters.
`all(iterable)`	Returns `True` if all the elements of `iterable` are true; otherwise, it returns `False`.
`any(iterable)`	Returns `True` if any of the elements of `iterable` are true; otherwise, it returns `False`.
`basestring()`	An abstract superclass for `str` and `unicode`, usable for type checking.
`bool(object)`	Returns `True` or `False`, depending on the Boolean value of `object`.
`callable(object)`	Checks whether an object is callable.
`chr(number)`	Returns a character whose ASCII code is the given number.
`classmethod(func)`	Creates a class method from an instance method (see Chapter 7).
`cmp(x, y)`	Compares `x` and `y`. If `x < y`, it returns a negative number; if `x > y`, it returns a positive number; and if `x == y`, it returns zero.
`complex(real[, imag])`	Returns a complex number with the given real (and, optionally, imaginary) component.
`delattr(object, name)`	Deletes the given attribute from the given object.
`dict([mapping-or-sequence])`	Constructs a dictionary, optionally from another mapping or a list of `(key, value)` pairs. May also be called with keyword arguments.
`dir([object])`	Lists (most of) the names in the currently visible scopes, or optionally (most of) the attributes of the given object.
`divmod(a, b)`	Returns `(a//b, a%b)` (with some special rules for floats).

Continued

Table B-3. *Continued*

Function	Description
`enumerate(iterable)`	Iterates over `(index, item)` pairs, for all items in `iterable`.
`eval(string[, globals[, locals]])`	Evaluates a string containing an expression, optionally in a given global and local scope.
`execfile(file[, globals[, locals]])`	Executes a Python file, optionally in a given global and local scope.
`file(filename[, mode[, bufsize]])`	Creates a file object with a given file name, optionally with a given mode and buffer size.
`filter(function, sequence)`	Returns a list of the elements from the given sequence for which `function` returns true.
`float(object)`	Converts a string or number to a float.
`frozenset([iterable])`	Creates a set that is immutable, which means it can be added to other sets.
`getattr(object, name[, default])`	Returns the value of the named attribute of the given object, optionally with a given default value.
`globals()`	Returns a dictionary representing the current global scope.
`hasattr(object, name)`	Checks whether the given object has the named attribute.
`help([object])`	Invokes the built-in help system, or prints a help message about the given object.
`hex(number)`	Converts a number to a hexadecimal string.
`id(object)`	Returns the unique ID for the given object.
`input([prompt])`	Equivalent to `eval(raw_input(prompt))`.
`int(object[, radix])`	Converts a string or number (optionally with a given radix) or number to an integer.
`isinstance(object, classinfo)`	Checks whether the given object is an instance of the given `classinfo` value, which may be a class object, a type object, or a tuple of class and type objects.
`issubclass(class1, class2)`	Checks whether `class1` is a subclass of `class2` (every class is a subclass of itself).
`iter(object[, sentinel])`	Returns an iterator object, which is `object.__iter__()`, an iterator constructed for iterating a sequence (if `object` supports `__getitem__`), or, if `sentinel` is supplied, an iterator that keeps calling `object` in each iteration until `sentinel` is returned.
`len(object)`	Returns the length (number of items) of the given object.

Function	Description
`list([sequence])`	Constructs a list, optionally with the same items as the supplied sequence.
`locals()`	Returns a dictionary representing the current local scope (do not modify this dictionary).
`long(object[, radix])`	Converts a string (optionally with a given radix) or number to a long integer.
`map(function, sequence, ...)`	Creates a list consisting of the values returned by the given function when applying it to the items of the supplied sequence(s).
`max(object1, [object2, ...])`	If `object1` is a nonempty sequence, the largest element is returned; otherwise, the largest of the supplied arguments (`object1`, `object2`, . . .) is returned.
`min(object1, [object2, ...])`	If `object1` is a nonempty sequence, the smallest element is returned; otherwise, the smallest of the supplied arguments (`object1`, `object2`, . . .) is returned.
`object()`	Returns an instance of `object`, the base class for all new style classes.
`oct(number)`	Converts an integer number to an octal string.
`open(filename[, mode[, bufsize]])`	An alias for `file` (use `open`, not `file`, when opening files).
`ord(char)`	Returns the ASCII value of a single character (a string or Unicode string of length 1).
`pow(x, y[, z])`	Returns `x` to the power of `y`, optionally modulo `z`.
`property([fget[, fset[, fdel[, doc]]]])`	Creates a property from a set of accessors (see Chapter 9).
`range([start, ]stop[, step])`	Returns a numeric range (as a list) with the given `start` (inclusive, default 0), `stop` (exclusive), and `step` (default 1).
`raw_input([prompt])`	Returns data input by the user as a string, optionally using a given prompt.
`reduce(function, sequence[, initializer])`	Applies the given function cumulatively to the items of the sequence, using the cumulative result as the first argument and the items as the second argument, optionally with a start value (`initializer`).
`reload(module)`	Reloads an already loaded module and returns it.
`repr(object)`	Returns a string representation of the object, often usable as an argument to `eval`.
`reversed(sequence)`	Returns a reverse iterator over the sequence.

Continued

Table B-3. *Continued*

Function	Description
`round(float[, n])`	Rounds off the given float to `n` digits after the decimal point (default zero).
`set([iterable])`	Returns a set whose elements are taken from `iterable` (if given).
`setattr(object, name, value)`	Sets the named attribute of the given object to the given value.
`sorted(iterable[, cmp][, key][, reverse])`	Returns a new sorted list from the items in `iterable`. Optional parameters are the same as for the list method `sort`.
`staticmethod(func)`	Creates a static (class) method from an instance method (see Chapter 7).
`str(object)`	Returns a nicely formatted string representation of the given object.
`sum(seq[, start])`	Returns the sum of a sequence of numbers, added to the optional parameter `start` (default 0).
`super(type[, obj/type])`	Returns the superclass of the given type (optionally instantiated).
`tuple([sequence])`	Constructs a tuple, optionally with the same items as the supplied sequence.
`type(object)`	Returns the type of the given object.
`type(name, bases, dict)`	Returns a new type object with the given name, bases, and scope.
`unichr(number)`	The Unicode version of `chr`.
`unicode(object[, encoding[, errors]])`	Returns a Unicode encoding of the given object, possibly with a given encoding, and a given mode for handling errors (`'strict'`, `'replace'`, or `'ignore'`; `'strict'` is the default).
`vars([object])`	Returns a dictionary representing the local scope, or a dictionary corresponding to the attributes of the given object (do not modify the returned dictionary, as the result of such a modification is not defined by the language reference).
`xrange([start, ]stop[, step])`	Similar to `range`, but the returned object uses less memory, and should be used only for iteration.
`zip(sequence1, ...)`	Returns a list of tuples, where each tuple contains an item from each of the supplied sequences. The returned list has the same length as the shortest of the supplied sequences.

Table B-4. *List Methods*

Method	Description
`aList.append(obj)`	Equivalent to `aList[len(aList):len(aList)] = [obj]`.
`aList.count(obj)`	Returns the number of indices `i` for which `alist[i] == obj`.
`aList.extend(sequence)`	Equivalent to `aList[len(aList):len(aList)] = sequence`.
`aList.index(obj)`	Returns the smallest `i` for which `aList[i] == obj` (or raises a `ValueError` if no such `i` exists).
`aList.insert(index, obj)`	Equivalent to `aList[index:index] = [obj]` if `index >= 0`; if `index < 0`, `object` is prepended to the list.
`aList.pop([index])`	Removes and returns the item with the given index (default –1).
`aList.remove(obj)`	Equivalent to `del aList[aList.index(obj)]`.
`aList.reverse()`	Reverses the items of `aList` in place.
`aList.sort([cmp][, key][, reverse])`	Sorts the items of `aList` in place (stable sorting). Can be customized by supplying a comparison function, `cmp`; a key function, `key`, which will create the keys for the sorting); and a `reverse` flag (a Boolean value).

Table B-5. *Dictionary Methods*

Method	Description
`aDict.clear()`	Removes all the items of `aDict`.
`aDict.copy()`	Returns a copy of `aDict`.
`aDict.fromkeys(seq[, val])`	Returns a dictionary with keys from `seq` and values set to `val` (default `None`). May be called directly on the dictionary type, `dict`, as a class method.
`aDict.get(key[, default])`	Returns `aDict[key]` if it exists; otherwise, it returns the given default value (default `None`).
`aDict.has_key(key)`	Checks whether `aDict` has the given key.
`aDict.items()`	Returns a list of `(key, value)` pairs representing the items of `aDict`.
`aDict.iteritems()`	Returns an iterable object over the same `(key, value)` pairs as returned by `aDict.items`.
`aDict.iterkeys()`	Returns an iterable object over the keys of `aDict`.
`aDict.itervalues()`	Returns an iterable object over the values of `aDict`.
`aDict.keys()`	Returns a list of the keys of `aDict`.

Continued

Table B-5. *Continued*

Method	Description
`aDict.pop(key[, d])`	Removes and returns the value corresponding to the given key, or the given default, `d`.
`aDict.popitem()`	Removes an arbitrary item from `aDict` and returns it as a `(key, value)` pair.
`aDict.setdefault(key[, default])`	Returns `aDict[key]` if it exists; otherwise, it returns the given default value (default `None`) and binds `aDict[key]` to it.
`aDict.update(other)`	For each item in `other`, adds the item to `aDict` (possibly overwriting existing items). Can also be called with arguments similar to the dictionary constructor, `aDict`.
`aDict.values()`	Returns a list of the values in `aDict` (possibly containing duplicates).

Table B-6. *String Methods*

Method	Description
`string.capitalize()`	Returns a copy of the string in which the first character is capitalized.
`string.center(width[, fillchar])`	Returns a string of length `max(len(string), width)` in which a copy of `string` is centered, padded with `fillchar` (the default is space characters).
`string.count(sub[, start[, end]])`	Counts the occurrences of the substring `sub`, optionally restricting the search to `string[start:end]`.
`string.decode([encoding[, errors]])`	Returns decoded version of the string using the given encoding, handling errors as specified by `errors` (`'strict'`, `'ignore'`, or `'replace'`).
`string.encode([encoding[, errors]])`	Returns the encoded version of the string using the given encoding, handling errors as specified by `errors` (`'strict'`, `'ignore'`, or `'replace'`).
`string.endswith(suffix[, start[, end]])`	Checks whether `string` ends with `suffix`, optionally restricting the matching with the given indices `start` and `end`.
`string.expandtabs([tabsize])`	Returns a copy of the string in which tab characters have been expanded using spaces, optionally using the given `tabsize` (default 8).
`string.find(sub[, start[, end]])`	Returns the first index where the substring `sub` is found, or –1 if no such index exists, optionally restricting the search to `string[start:end]`.

Method	Description
`string.index(sub[, start[, end]])`	Returns the first index where the substring `sub` is found, or raises a `ValueError` if no such index exists, optionally restricting the search to `string[start:end]`.
`string.isalnum()`	Checks whether the string consists of alphanumeric characters.
`string.isalpha()`	Checks whether the string consists of alphabetic characters.
`string.isdigit()`	Checks whether the string consists of digits.
`string.islower()`	Checks whether all the case-based characters (letters) of the string are lowercase.
`string.isspace()`	Checks whether the string consists of whitespace.
`string.istitle()`	Checks whether all the case-based characters in the string following non-case–based letters are uppercase and all other case-based characters are lowercase.
`string.isupper()`	Checks whether all the case-based characters of the string are uppercase.
`string.join(sequence)`	Returns a string in which the string elements of `sequence` have been joined by `string`.
`string.ljust(width[, fillchar])`	Returns a string of length `max(len(string), width)` in which a copy of `string` is left-justified, padded with `fillchar` (the default is space characters).
`string.lower()`	Returns a copy of the string in which all case-based characters have been lowercased.
`string.lstrip([chars])`	Returns a copy of the string in which all `chars` have been stripped from the beginning of the string (the default is all whitespace characters, such as spaces, tabs, and newlines).
`string.partition(sep)`	Searches for `sep` in the string and returns `(head, sep, tail)`.
`string.replace(old, new[, max])`	Returns a copy of the string in which the occurrences of `old` have been replaced with `new`, optionally restricting the number of replacements to `max`.
`string.rfind(sub[, start[, end]])`	Returns the last index where the substring `sub` is found, or –1 if no such index exists, optionally restricting the search to `string[start:end]`.
`string.rindex(sub[, start[, end]])`	Returns the last index where the substring `sub` is found, or raises a `ValueError` if no such index exists, optionally restricting the search to `string[start:end]`.

Continued

Table B-6. *Continued*

Method	Description
`string.rjust(width[, fillchar])`	Returns a string of length `max(len(string), width)` in which a copy of `string` is right-justified, padded with `fillchar` (the default is space characters).
`string.rpartition(sep)`	Same as `partition`, but searches from the right.
`string.rstrip([chars])`	Returns a copy of the string in which all `chars` have been stripped from the end of the string (the default is all whitespace characters, such as spaces, tabs, and newlines).
`string.rsplit([sep[, maxsplit]])`	Same as `split`, but when using `maxsplit`, counts from right to left.
`string.split([sep[, maxsplit]])`	Returns a list of all the words in the string, using `sep` as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to `maxsplit`.
`string.splitlines([keepends])`	Returns a list with all the lines in `string`, optionally including the line breaks (if `keepends` is supplied and is true).
`string.startswith(prefix[, start[, end]])`	Checks whether `string` starts with `prefix`, optionally restricting the matching with the given indices `start` and `end`.
`string.strip([chars])`	Returns a copy of the string in which all `chars` have been stripped from the beginning and the end of the string (the default is all whitespace characters, such as spaces, tabs, and newlines).
`string.swapcase()`	Returns a copy of the string in which all the case-based characters have had their case swapped.
`string.title()`	Returns a copy of the string in which all the words are capitalized.
`string.translate(table[, deletechars])`	Returns a copy of the string in which all characters have been translated using `table` (constructed with the `maketrans` function in the `string` module), optionally deleting all characters found in the string `deletechars`.
`string.upper()`	Returns a copy of the string in which all the case-based characters have been uppercased.
`string.zfill(width)`	Pads `string` on the left with zeros to fill `width`.

Statements

This section gives you a quick summary of each of the statement types in Python.

Simple Statements

Simple statements consist of a single (logical) line.

Expression Statements

Expressions can be statements on their own. This is especially useful if the expression is a function call or a documentation string.

Example:

```
"This module contains SPAM-related functions."
```

Assert Statements

Assert statements check whether a condition is true and raise an `AssertionError` (optionally with a supplied error message) if it isn't.

Example:

```
assert age >= 12, 'Children under the age of 12 are not allowed'
```

Assignment Statements

Assignment statements bind variables to values. Multiple variables may be assigned to simultaneously (through sequence unpacking) and assignments may be chained.

Examples:

```
x = 42                   # Simple assignment
name, age = 'Gumby', 60  # Sequence unpacking
x = y = z = 10           # Chained assignments
```

Augmented Assignment Statements

Assignments may be augmented by operators. The operator will then be applied to the existing value of the variable and the new value, and the variable will be rebound to the result. If the original value is mutable, it may be modified instead (with the variable staying bound to the original).

Examples:

```
x *= 2                   # Doubles x
x += 5                   # Adds 5 to x
```

The pass Statement

The pass statement is a "no-op," which does nothing. It is useful as a placeholder, or as the only statement in syntactically required blocks where you want no action to be performed.

Example:

```
try: x.name
except AttributeError: pass
else: print 'Hello', x.name
```

The del Statement

The del statement unbinds variables and attributes, and removes parts (positions, slices, or slots) from data structures (mappings or sequences). It cannot be used to delete values directly, because values are only deleted through garbage collection.

Examples:

```
del x                    # Unbinds a variable
del seq[42]              # Deletes a sequence element
del seq[42:]             # Deletes a sequence slice
del map['foo']           # Deletes a mapping item
```

The print Statement

The print statement writes one or more values (automatically formatted with str, separated by single spaces) to a given stream, with sys.stdout being the default. It adds a line break to the end of the written string unless the print statement ends with a comma.

Examples:

```
print 'Hello, world!'     # Writes 'Hello, world\n' to sys.stdout
print 1, 2, 3             # Writes '1 2 3\n' to sys.stdout
print >> somefile, 'xyz'  # Writes 'xyz' to somefile
print 42,                 # Writes '42 ' to sys.stdout
```

The return Statement

The return statement halts the execution of a function and returns a value. If no value is supplied, None is returned.

Examples:

```
return                   # Returns None from the current function
return 42                # Returns 42 from the current function
return 1, 2, 3           # Returns (1, 2, 3) from the current function
```

The yield Statement

The yield statement temporarily halts the execution of a generator and yields a value. A generator is a form of iterator and can be used in for loops, among other things.

Example:

```
yield 42                 # Returns 42 from the current function
```

The raise Statement

The `raise` statement raises an exception. It may be used without any arguments (inside an `except` clause, to re-raise the currently caught exception), with a subclass of `Exception` and an optional argument (in which case, an instance is constructed), or with an instance of a subclass of `Exception`.

Examples:

```
raise                    # May only be used inside except clauses
raise IndexError
raise IndexError, 'index out of bounds'
raise IndexError('index out of bounds')
```

The break Statement

The `break` statement ends the immediately enclosing loop statement (`for` or `while`) and continues execution immediately after that loop statement.

Example:

```
while True:
    line = file.readline()
    if not line: break
    print line
```

The continue Statement

The `continue` statement is similar to the `break` statement in that it halts the current iteration of the immediately enclosing loop, but instead of ending the loop completely, it continues execution at the beginning of the next iteration.

Example:

```
while True:
    line = file.readline()
    if not line: break
    if line.isspace(): continue
    print line
```

The import Statement

The `import` statement is used to import names (variables bound to functions, classes, or other values) from an external module. This also covers `from __future__ import ...` statements for features that will become standard in future versions of Python.

Examples:

```
import math
from math import sqrt
from math import sqrt as squareroot
from math import *
```

The global Statement

The `global` statement is used to mark a variable as global. It is used in functions to allow statements in the function body to rebind global variables. Using the `global` statement is generally considered poor style and should be avoided whenever possible.

Example:

```
count = 1
def inc():
    global count
    count += 1
```

The exec Statement

The exec statement is used to execute strings containing Python statements, optionally with a given global and local namespace (dictionaries).

Examples:

```
exec 'print "Hello, world!"'
exec 'x = 2' in myglobals, mylocals # ... where myglobals and mylocals are dicts
```

Compound Statements

Compound statements contain groups (blocks) of other statements.

The if Statement

The `if` statement is used for conditional execution, and it may include `elif` and `else` clauses.

Example:

```
if x < 10:
    print 'Less than ten'
elif 10 <= x < 20:
    print 'Less than twenty'
else:
    print 'Twenty or more'
```

The while Statement

The `while` statement is used for repeated execution (looping) while a given condition is true. It may include an `else` clause (which is executed if the loop finishes normally, without any `break` or `return` statements, for instance).

Example:

```
x = 1
while x < 100:
    x *= 2
print x
```

The for Statement

The `for` statement is used for repeated execution (looping) over the elements of sequences or other iterable objects (objects having an `__iter__` method that returns an iterator). It may include an `else` clause (which is executed if the loop finishes normally, without any `break` or `return` statements, for instance).

Example:

```
for i in range(10, 0, -1):
    print i
print 'Ignition!'
```

The try Statement

The `try` statement is used to enclose pieces of code where one or more known exceptions may occur, and enables your program to trap these exceptions and perform exception-handling code if an exception is trapped. The `try` statement can combine several `except` clauses (handling exceptional circumstances) and `finally` clauses (executed no matter what; useful for cleanup).

Example:

```
try:
    1/0
except ZeroDivisionError:
    print "Can't divide anything by zero."
finally:
    print "Done trying to calculate 1/0"
```

The with Statement

The `with` statement is used to wrap a block of code using a so-called context manager, allowing the context manager to perform some setup and cleanup actions. For example, files can be used as context managers, and they will close themselves as part of the cleanup.

Note In Python 2.5, you need `from __future__ import with_statement` for the `with` statement to work as described.

Example:

```
with open("somefile.txt") as myfile:
    dosomething(myfile)
# The file will have been closed here
```

Function Definitions

Function definitions are used to create function objects and to bind global or local variables to these function objects.

Example:

```
def double(x):
    return x*2
```

Class Definitions

Class definitions are used to create class objects and to bind global or local variables to these class objects.

Example:

```
class Doubler:
    def __init__(self, value):
        self.value = value
    def double(self):
        self.value *= 2
```

APPENDIX C

Online Resources

As you learn Python, the Internet will serve as an invaluable resource. This appendix describes some of the web sites that may be of interest to you as you are starting out. If you are looking for something Python-related that isn't described here, I suggest that you first check the official Python web site (`http://python.org`), and then use your favorite web search engine, or the other way around. There is a lot of information about Python online; chances are you'll find something. If you don't, you can always try `comp.lang.python` (described in this appendix). If you're an IRC user (see `http://irchelp.org` for information), you might want to check out the `#python` channel on `irc.freenode.net`.

Python Distributions

Several Python distributions are available. Here are some of the more prominent ones:

Official Python distribution (`http://python.org/download`): This comes with a default integrated development environment called IDLE (for more information, see `http://docs.python.org/lib/idle.html`).

ActivePython (`http://activestate.com`): This is ActiveState's Python distribution, which includes several nonstandard packages in addition to the official distribution. This is also the home of Visual Python, a Python plug-in for Visual Studio .NET.

Jython (`http://www.jython.org`): Jython is the Java implementation of Python.

IronPython (`http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython`): IronPython is the C# implementation of Python.

MacPython (`http://homepages.cwi.nl/~jack/macpython/index.html`): MacPython is the Macintosh port of Python for older versions of Mac OS. The new Mac version can be found on the main Python site (`http://python.org`). You can also get Python through MacPorts (`http://macports.org`).

pywin32 (`http://sf.net/projects/pywin32/`): These are the Python for Windows extensions. If you have ActivePython installed, you already have all these extensions.

Python Documentation

Answers to most of your Python questions are most likely somewhere on the `python.org` web site. The documentation can be found at `http://python.org/doc`, with the following subdivisions:

Python Tutorial (`http://python.org/doc/tut`): This is a relatively simple introduction to the language.

Python Reference Manual (`http://python.org/doc/ref`): This document contains a precise definition of the Python language. It may not be the place to start when learning Python, but it contains precise answers to most questions you might have about the language.

Python Library Reference (`http://python.org/doc/lib`): This is probably the most useful piece of Python documentation you'll ever find. It describes all (or most) of the modules in the standard Python library. If you are wondering how to solve a problem in Python, this should be the first place you look—perhaps the solution already exists in the libraries.

Extending and Embedding the Python Interpreter (`http://python.org/doc/ext`): This is a document that describes how to write Python extension modules in the C language, and how to use the Python interpreter as a part of larger C programs. (Python itself is implemented in C.)

Macintosh Library Modules (`http://python.org/doc/mac`): This document describes functionality specific to the Macintosh port of Python.

Python/C API Reference Manual (`http://python.org/doc/api`): This is a rather technical document describing the details of the Python/C application programming interface (API), which enables C programs to interface with the Python interpreter.

Two other useful documentation resources are Python Documentation Online (`http://pydoc.org`) and `pyhelp.cgi` (`http://starship.python.net/crew/theller/pyhelp.cgi`), which allow you to search the standard Python documentation. If you want some "recipes" and solutions provided by the Python community, the Python Cookbook (`http://aspn.activestate.com/ASPN/Python/Cookbook`) is a good place to look.

The future of Python is decided by the language's Benevolent Dictator For Life (BDFL), Guido van Rossum, but his decisions are guided and informed by so-called Python Enhancement Proposals, which may be accessed at `http://python.org/dev/peps`. Various HOWTO documents (relatively specific tutorials) can be found at `http://python.org/doc/howto`.

Useful Toolkits and Modules

One source for finding software implemented in Python (including useful toolkits and modules you can use in your own programs) is the Vaults of Parnassus (`http://www.vex.net/parnassus`); another is the Python Package Index (`http://pypi.python.org/pypi`). If you can't find what you're looking for on either of these sites, try a standard web search, or perhaps take a look at freshmeat (`http://freshmeat.net`) or SourceForge (`http://sf.net`).

Table C-1 lists the URLs of some of the most well-known GUI toolkits available for Python. For a more thorough description, see Chapter 12. Table C-2 lists the URLs of the third-party packages used in the ten projects (Chapters 20–29).

Table C-1. *Some Well-Known GUI Toolkits for Python*

Toolkit	URL
Tkinter	`http://python.org/topics/tkinter/doc.html`
wxPython	`http://www.wxpython.org`
PythonWin	`http://sf.net/projects/pywin32/`
Java Swing	`http://java.sun.com/docs/books/tutorial/uiswing`
PyGTK	`http://www.pygtk.org`
PyQt	`http://www.thekompany.com/projects/pykde`

Table C-2. *The Third-Party Modules Used in This Book's Ten Projects*

Package	URL
Psycopg	`http://initd.org/pub/software/psycopg/`
MySQLdb	`http://sourceforge.net/projects/mysql-python`
Pygame	`http://www.pygame.org`
PyXML	`http://sourceforge.net/projects/pyxml`
ReportLab	`http://www.reportlab.org`

Newsgroups, Mailing Lists, and Blogs

An important forum for Python discussion is the Usenet group `comp.lang.python`. If you're serious about Python, skimming this group regularly can be quite useful. Its companion group, `comp.lang.python.announce`, contains announcements about new Python software (including new Python distributions, Python extensions, and software written using Python).

Several official mailing lists are available. For instance, the `comp.lang.python` group is mirrored in the `python-list@python.org` mailing list. If you have a Python problem and need help, simply send an email to `help@python.org` (assuming that you've exhausted all other options, of course). For learning about programming in Python, the tutor list (`tutor@python.org`) may be useful. For information about how to join these (and other) mailing lists, see `http://mail.python.org/mailman/listinfo`.

A couple of useful blogs are Unofficial Planet Python (`http://planetpython.org`) and The Daily Python-URL (`http://pythonware.com/daily`).

APPENDIX D

Python 3.0

This book describes mainly the language defined by Python version 2.5. Python version 3.0 (and its companion “transition” release, 2.6) isn’t all that different. Most things work just as they did before, but the language cleanups introduced mean that some existing code will break.

If you’re transitioning from older code to Python 3.0, a couple of tools can come in quite handy. First, Python 2.6 comes with optional warnings about 3.0 incompatibilities (run Python with the `-3` flag). If you first make sure your code runs without errors in 2.6 (which is largely backward-compatible), you can refactor away any incompatibility warnings. (Needless to say, you should have solid unit tests in place before you do this; see Chapter 16 for more advice on testing.) Second, Python 3.0 ships with an automatic refactoring tool called `2to3`, which can automatically upgrade your source files. (Be sure to back up or check in your files before performing any large-scale transformations.) If you wish to have both 2.6 and 3.0 code available, you could keep working on the 2.6 code (with the proper warnings turned on), and generate 3.0 code when it’s time for releasing.

Throughout the book, you’ll find notes about things that change in Python 3.0. This appendix gives a more comprehensive set of pointers for moving to the world of 3.0. I’ll describe some of the more noticeable changes, but not everything that is new in Python 3.0. There are many changes, both major and minor. Table D-1 (which is based on the document *What’s New in Python 3.0?*, by Guido van Rossum), at the end of this appendix, lists quite a few more changes and also refers to relevant PEP documents, when applicable (available from `http://python.org/dev/peps`). Table D-2 lists some other sources of further information.

Strings and I/O

The following sections deal with new features related to text. Strings are no longer simply byte sequences (although such sequences are still available), the `input`/`print` pair has been revamped slightly, and string formatting has had a major facelift.

Strings, Bytes, and Encodings

The distinction between text and byte sequences is significantly cleaned up in Python 3.0. Strings in previous versions were based on the somewhat outmoded (yet still prevalent) notion that text characters can easily be represented as single bytes. While this is true for English and most western languages, it fails to account for ideographic scripts, such as Chinese.

The Unicode standard was created to encompass all written languages, and it admits about 100,000 different characters, each of which has a unique numeric code. In Python 3.0, `str` is, in fact, the `unicode` type from earlier versions, which is a sequence of Unicode characters. As there is no unique way of encoding these into byte sequences (which you need to do in order to perform disk I/O, for example), you must supply an encoding (with UTF-8 as the default in most cases). So, text files are now assumed to be encoded versions of Unicode, rather than simply arbitrary sequences of bytes. (Binary files are still just byte sequences, though.) As a consequence of this, constants such as `string.letters` have been given the prefix `ascii_` (for example, `string.ascii_letters`) to make the link to a specific encoding clear.

To avoid losing the old functionality of the previous `str` class, there is a new class called `bytes`, which represents immutable sequences of bytes (as well as `bytearray`, which is its mutable sibling).

Console I/O

There is little reason to single out console printing to the degree that it has its own statement. Therefore, the `print` statement is changed into a function. It still works in a manner very similar to the original statement (for example, you can print several arguments by separating them with commas), but the stream redirection functionality is now a keyword argument. In other words, instead of writing this:

```
print >> sys.stderr, "fatal error:", error
```

you would write this:

```
print("fatal error:", error, file=sys.stderr)
```

Also, the behavior of the original `input` no longer has its own function. The name `input` is now used for what used to be `raw_input`, and you need to explicitly say `eval(input())` to get the old functionality.

New String Formatting

Strings now have a new method, called `format`, which allows you to perform rather advanced string formatting. The fields in the string where values are to be spliced in are enclosed in braces, rather than prefaced with a `%` (and braces are escaped by using double braces). The replacement fields refer to the arguments of the `format` method, either by numbers (for positional arguments) or names (for keyword arguments):

```
>>> "{0}, {1}, {x}".format("a", 1, x=42)
'a 1 42'
```

In addition, the replacement fields can access attributes and elements of the values to be replaced, such as in `"{foo.bar}"` or `"{foo[bar]}"`, and can be modified by format specifiers similar to those in the current system. This new mechanism is quite flexible, and because it allows classes to specify their own format string behavior (through the magic `__format__` method), you will be able to write much more elegant output formatting code.

Classes and Functions

Although none of the changes are quite as fundamental as the introduction of new-style classes, Python 3 has some goodies in store in the abstraction department: functions can now be annotated with information about parameters and return values, there is a framework for abstract base classes, metaclasses have a more convenient syntax, and you can have keyword-only parameters and nonlocal (but not global) variables.

Function Annotation

The new function annotation system is something of a wildcard. It allows you to annotate the arguments and the return type of a function (or method) with the values of arbitrary expressions, and then to retrieve these values later. However, what this system is to be used for is not specified. It is motivated by several practical applications (such as more fine-grained docstring functionality, type specifications and checking, generic functions, and more), but you can basically use it for anything you like.

A function is annotated as follows:

```
def frozzbozz(x: foo, y: bar = 42) -> baz:
    pass
```

Here, `foo`, `bar`, and `baz` are annotations for the positional argument `x`, the keyword argument `y`, and the return value of `frozzbozz`, respectively. These can be retrieved from the dictionary `frozzbozz.func_annotations`, with the parameter names (or `"return"` for the return value) as keys.

Abstract Base Classes

Sometimes you might want to implement only *parts* of a class. For example, you may have functionality that is to be shared among several classes, so you put it in a superclass. However, the superclass isn't really complete and shouldn't be instantiated by itself—it's only there for others to inherit. This is called an *abstract base class* (or simply an *abstract class*). It's quite common for such abstract classes to define nonfunctional methods that the subclasses need to override. In this way, the base class also acts as an interface definition, in a way.

You can certainly simulate this with older Python versions (for example, by raising `NotImplementedError`), but now there is a more complete framework for abstract base classes. This framework includes a new metaclass (`ABCMeta`), and the decorators `@abstractmethod` and `@abstractproperty` for defining abstract (that is, unimplemented) methods and properties, respectively. There's also a separate module (`abc`) that serves as a "support framework" for abstract base classes.

Class Decorators and New Metaclass Syntax

Class decorators work in a manner similar to function decorators. Simply put, instead of the following:

```
class A:
    pass
A = foo(A)
```

you could write this:

```
@foo
class A:
    pass
```

In other words, this lets you do some processing on the newly created class object. In fact, it may let you do many of the things you might have used a metaclass for in the past. But in case you need a metaclass, there is even a new syntax for those. Instead of this:

```
class A:
    __metaclass__ = foo
```

you can now write this:

```
class A(metaclass=foo):
    pass
```

For more information about class decorators, see PEP 3129 (`http://python.org/dev/peps/pep-3129`), and for more on the new metaclass syntax, see PEP 3115 (`http://python.org/dev/peps/pep-3115`).

Keyword-Only Parameters

It's now possible to define parameters that must be supplied as keywords (if at all). In previous versions, any keyword parameter could also be supplied as a positional parameter, unless you used a function definition such as `def foo(**kwds):` and processed the `kwds` dictionary yourself. If a keyword argument was required, you needed to raise an exception explicitly when it was missing.

The new functionality is simple, logical, and elegant. You can now put parameters after a varargs argument:

```
def foo(*args, my_param=42): ...
```

The parameter `my_param` will never be filled by a positional argument, as they are all eaten by `args`. If it is to be supplied, it must be supplied as a keyword argument. Interestingly, you do not even need to give these keyword-only parameters a default. If you don't, they become *required* keyword-only parameters (that is, not supplying them would be an error). If you don't want the varargs argument (`args`), you could use the new syntactical form, where the varargs operator (`*`) is used without a variable:

```
def foo(x, y, *, z): ...
```

Here, `x` and `y` are required positional parameters, and `z` is a required keyword parameter.

Nonlocal Variables

When nested (static) scopes were introduced in Python, they were *read-only*, and they have been ever since; that is, you can access the local variables of outer scopes, but you can't rebind them. There's a special case for the global scope, of course. If you declare a variable to be global (with the `global` keyword), you can rebind it globally. Now you can do the same for outer, non-global scopes, using the `nonlocal` keyword.

Iterables, Comprehensions, and Views

Some other new features include being able to collect excess elements when unpacking iterables, constructing dictionaries and sets in a manner similar to list comprehension, and creating dynamically updatable views of a dictionary. The use of iterable objects has also extended to the return values of several built-in functions.

Extended Iterable Unpacking

Iterable unpacking (such as `x, y, z = iterable`) has previously required that you know the exact number of items in the iterable object to be unpacked. Now you can use the * operator, just for parameters, to gather up extra items as a list. This operator can be used on any one of the variables on the left-hand side of the assignment, and that variable will gather up any items that are left over when the other variables have received their items:

```
>>> a, *b, c, d = [1, 2, 3, 4, 5]
>>> a, b, c, d
(1, [2, 3], 4, 5)
```

Dictionary and Set Comprehension

It is now possible to construct dictionaries and sets using virtually the same comprehension syntax as for list comprehensions and generator expressions:

```
>>> {i:i for i in range(5)}
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
>>> {i for i in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

The last result also demonstrates the new syntax for sets (see the section "Some Minor Issues," later in this appendix).

Dictionary Views

You can now access different *views* on dictionaries. These views are collection-like objects that change automatically to reflect updates to the dictionary itself. The views returned by `dict.keys` and `dict.items` are set-like, and cannot include duplicates, while the views returned by `dict.values` can. The set-like views permit set operations.

Iterator Return Values

Several functions and methods that used to return lists now return more lazy iterable objects instead. Examples include `range`, `zip`, `map`, and `filter`.

Things That Have Gone

Some functions will simply disappear in Python 3.0. For example, you can no longer use `apply`. Then again, with the * and ** operators for argument splicing, you don't really need it. Another notable example is `callable`. With it gone, you now have two main options for finding out whether an object is callable: you can check whether it has the magic method `__callable__`, or you can simply try to call it (using `try`/`except`). Other examples include `execfile` (use `exec` instead), `reload` (use `exec` here, too), `reduce` (it's now in the `functools` module), `coerce` (not needed with the new numeric type hierarchy), and `file` (use `open` to open files).

Some Minor Issues

The following are some minor issues that might trip you up:

- The old (and deprecated) form of the inequality operator, `<>`, is no longer allowed. You should write `!=` instead (which is common practice already).
- Backquotes won't work anymore. You should use `repr` instead.
- Comparison operators (`<`, `<=`, and the like) won't allow you to compare incompatible types. For example, you can no longer check whether `4` is greater than `"5"` (this is consistent with the existing rules for addition).
- There is a new syntax for sets: `{1, 2, 3}` is the same as `set([1, 2, 3])`. However, `{}` is still an empty dictionary. Use `set()` to get an empty set.
- Division is now real division! In other words, `1/2` will give you `0.5`, not `0`. For integer division, use `1//2`. Because this is a "silent error" (you won't get any error messages if you try to use `/` for integer division), it can be insidious.

The Standard Library

The standard library is reorganized quite a bit in Python 3.0. A thorough discussion can be found in PEP 3108 (`http://www.python.org/dev/peps/pep-3108`). Here are some examples:

- Several modules are removed. This includes previously deprecated modules (such as `mimetools` and `md5`), platform-specific ones (for IRIX, Mac OS, and Solaris), and some that are hardly used (such as `mutex`) or obsolete (such as `bsddb185`). Important functionality is generally preserved through other modules.
- Several modules are renamed, to conform to PEP 8: Style Guide for Python Code (`http://www.python.org/dev/peps/pep-0008`), among other things. For example, `copy_reg` is now `copyreg`, `ConfigParser` is `configparser`, `cStringIO` is dropped, and `StringIO` is added to the `io` module.
- Several modules have been grouped into packages. For example, the various HTTP-related modules (such as `httplib`, `BaseHTTPServer`, and `Cookie`) are now collected in the new `http` packages (as `http.client`, `http.server`, and `http.cookies`).

The idea behind these changes is, of course, to tidy things up a bit.

Other Stuff

As I mentioned at the beginning of this appendix, Python 3.0 has a lot of new features. Table D-1 lists many of them, including some I haven't discussed in this appendix. If there's something specific that's tripping you up, you might want to take a look at the official documentation or play around with the `help` function. See also Table D-2 for some sources of further information.

Table D-1. *Important New Features in Python 3.0*

Feature	Related PEP
`print` is a function.	PEP 3105
Text files enforce an encoding.	
`zip`, `map`, and `filter` return iterators.	
`dict.keys()`, `dict.values()`, and `dict.items()` return views, not lists.	
The `cmp` argument is gone from `sorted` and `list.sort`. Use `key` instead.	PEP 3100
Division is now true division: `1/2 == 0.5`.	PEP 238
There is only one string type, `str`, and it's equivalent to the Python 2.*x* `unicode` type.	
The `basestring` class is removed.	
The new `bytes` type is used for representing binary data and encoded text.	PEP 3137
`bytes` literals are written as `b"abc"`.	PEP 3137
UTF-8 is the default Python source encoding. Non-ASCII identifiers are permitted.	PEP 3120
`StringIO` and `cStringIO` are superseded by `io.StringIO` and `io.BytesIO`.	PEP 0364
New built-in string formatting replaces the `%` operator.	PEP 3101
Functions can have their parameters and return type annotated.	PEP 3107
Use `raise Exception(args)`, not `raise Exception, args`.	PEP 3109
Use `except MyException as identifier:`, not `except MyException, identifier:`.	PEP 3110
Classic/old-style classes are gone.	
Set metaclass with class `Foo(Base, metaclass=Meta):`.	PEP 3115
Abstract classes, `@abstractmethod`, and `@abstractproperty` are added.	PEP 3119
Class decorators, similar to function decorators, are added.	PEP 3129
Backquotes are gone. Use `repr`.	
`<>` is gone. Use `!=`.	
`True`, `False`, `None`, `as`, and `with` are keywords (they can't be used as names).	
`long` is renamed to `int`, and is now the only integer type, but without the `L`.	PEP 237
`sys.maxint` is gone, as there is no longer a maximum.	PEP 237

Continued

Table D-1. *Continued*

Feature	Related PEP
`x < y` is now an error if `x` and `y` are of incompatible types.	
`__getslice__` and friends are gone. Instead, `__getitem__` is called with a `slice`.	
Parameters can be specified as *keyword-only*.	PEP 3102
After `nonlocal x`, you can assign to `x` in an outer (nonglobal) scope.	PEP 3104
`raw_input` is renamed to `input`. For the old input behavior, use `eval(input())`.	PEP 3111
`xrange` is renamed to `range`.	
Tuple parameter unpacking is removed. `def foo(a, (b, c)):` won't work.	PEP 3113
`next` in iterators is renamed `x.__next__`. `next(x)` calls `x.__next__`.	PEP 3114
There are new octal literals. Instead of `0666`, write `0o666`.	PEP 3127
There are new binary literals. `0b1010 == 10`. `bin()` is the binary equivalent to `hex()` and `oct()`.	PEP 3127
Starred iterable unpacking is added, as for parameters: `a, b, *rest = seq` or `*rest, a = seq`.	PEP 3132
`super` may now be invoked without arguments, and will do the right thing.	PEP 3135
`string.letters` and friends are gone. Use `string.ascii_letters`.	
`apply` is gone. Replace `apply(f, x)` with `f(*x)`.	
`callable` is gone. Replace `callable(f)` with `hasattr(f, "__call__")`.	
`coerce` is gone.	
`execfile` is gone. Use `exec` instead.	
`file` is gone.	
`reduce` is moved to the `functools` module.	
`reload` is gone. Use `exec` instead.	
`dict.has_key` is gone. Replace `d.has_key(k)` with `k in d`.	
`exec` is now a function.	

Table D-2. *Sources of Information for Python 2.6 and 3.0*

Name	URL
Python v3.0 Documentation	`http://docs.python.org/dev/3.0`
What's New in Python 3.0?	`http://docs.python.org/dev/3.0/whatsnew/3.0.html`
PEP 3000: Python 3000	`http://www.python.org/dev/peps/pep-3000`
Python 3000 and You	`http://www.artima.com/weblogs/viewpost.jsp?thread=227041`

Index

Symbols and Numerics

A

B

C

D

E

G

H

I

O

P

S

V

W

X

Y

Z