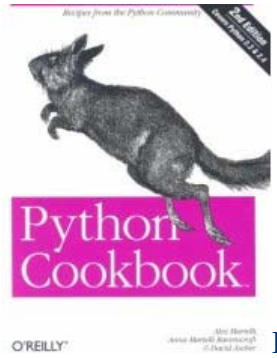

By David Ascher, Alex Martelli, Anna Ravenscroft

This book is not a typical O'Reilly book, written as a cohesive manuscript by one or two authors. Instead, it is a new kind of book a bold attempt at applying some principles of open source development to book authoring. Over 300 members of the Python community contributed materials to this book. In this Preface, we, the editors, want to give you, the reader, some background regarding how this book came about and the processes and people involved, and some thoughts about the implications of this new form.

Text 2
Files....13
Time and Money 24
Shortcuts 30
Searching and Sorting 41
Object-Oriented Programming....49
Persistence and Databases....53
Algorithms 54
Iterators and Generators 61
Descriptors, Decorators,and Metaclasses....77
Debugging and Testing 81

Text

Python strings are immutable sequences of bytes or characters. Most of the ways we create and process strings treat them as sequences of characters, but many are just as applicable to sequences of bytes. Unicode strings are immutable sequences of Unicode characters: transformations of Unicode strings into and from plain strings use **codecs** (coder-decoders) objects that embody knowledge about the many standard ways in which sequences of characters can be represented by sequences of bytes (also known as **encodings** and **character sets**). Note that Unicode strings do not serve double duty as sequences of bytes.

String values can be enclosed in either single or double quotes. The two different kinds of quotes work the same way, but having both allows you to include one kind of quotes inside of a string specified with the other kind of quotes, without needing to escape them with the backslash character:

```
'isn\'t that grand'
"isn't that grand"
```

To have a string literal span multiple lines, you can use a backslash as the last character on the line, which indicates that the next line is a continuation:

```
big = "This is a long string\
that spans two lines."
```

Using triple quotes, you don't need to use the continuation character, and line breaks in the string literal are preserved as newline characters in the resulting Python string object.

You can also make a string literal "raw" string by preceding it with an r or R:

```
big = r"This is a long string\
with a backslash and a newline in it"
```

With a raw string, backslash escape sequences are left alone, rather than being interpreted. Finally, you can precede a string literal with a u or U to make it a Unicode string:

```
hello = u'Hello\u0020World'
```

Strings are immutable, which means that no matter what operation you do on a string, you will always produce a new string object, rather than mutating the existing string. In general, you can do anything to a string that you can do to any other sequence, as long as it doesn't require changing the sequence, since strings are immutable.

String objects have many useful methods. For example, you can test a

string's contents with s.isdigit(). You can produce a new modified string with a method call such as s.toupper(), which returns a new string that is like s, but with every letter changed into its uppercase equivalent. You can search for a string inside another with haystack.count('needle'), which returns the number of times the substring 'needle' appears in the string **haystack**. When you have a large string that spans multiple lines, you can split it into a list of single-line strings with splitlines:

```
list_of_lines = one_large_string.splitlines()
```

You can produce the single large string again with join:

```
one_large_string = '\n'.join(list_of_lines)
```

Strings in Python can also be manipulated with regular expressions, via the **re** module. Regular expressions are a powerful (but complicated) set of tools that you may already be familiar with from another language (such as Perl), or from the use of tools such as the **vi** editor and text-mode commands such as **grep**. You'll find a number of uses of regular expressions in recipes in the second half of this chapter. For complete documentation, see the Library Reference and Python in a Nutshell. J.E.F. Friedl, Mastering Regular Expressions (O'Reilly) is also recommended if you need to master this subjectPython's regular expressions are basically the same as Perl's, which Friedl covers thoroughly.

Python's standard module **string** offers much of the same functionality that is available from string methods, packaged up as functions instead of methods.

In Python, characters are just strings of length one. You can loop over a string to access each of its characters, one by one. You can use **map** for much the same purpose, as long as what you need to do with each character is call a function on it. Finally, you can call the built-in type **list** to obtain a list of the length-one substrings of the string.

```
thelist = list(thestring)

for c in thestring:
    do_something_with(c)

results = [do_something_with(c) for c in thestring]

results = map(do_something, thestring)
```

chr takes as its argument a small integer and returns the corresponding single-character string according to ASCII, while **str**, called with any integer, returns the string that is the decimal representation of that integer.

```
>>> print repr(chr(97))
```

```
'a'
>>> print repr(str(97))
'97'
```

To turn a string into a list of character value codes, use the built-in functions **map** and **ord** together, as follows:

```
>>> print map(ord, 'ciao')
[99, 105, 97, 111]
```

A simple and fast way to check whether something is a string or Unicode object is to use the built-ins **isinstance** and **basestring**, as follows:

```
def isAString(anobj):
    return isinstance(anobj, basestring)
```

basestring is a common base class for the **str** and **unicode** types, and any string-like type that user code might define should also subclass **basestring**, just to make sure that such **isinstance** testing works as intended. **basestring** is essentially an "empty" type, just like **object**, so no cost is involved in subclassing it.

the canonical **isinstance** checking fails to accept such clearly string-like objects as instances of the **UserString** class from Python Standard Library module **UserString**, since that class, alas, does not inherit from **basestring**. If you need to support such types, you can check directly whether an object behaves like a stringfor example:

```
def isStringLike(anobj):
    try: anobj + ''
    except: return False
    else: return True
```

Unfortunately, the canonical **isinstance** checking fails to accept such clearly string-like objects as instances of the **UserString** class from Python Standard Library module **UserString**, since that class, alas, does not inherit from **basestring**. If you need to support such types, you can check directly whether an object behaves like a stringfor example:

```
def isStringLike(anobj):
    try: anobj + ''
    except: return False
    else: return True
```

The general Python approach to type-checking is known as **duck typing**: if it walks like a duck and quacks like a duck, it's duck-like enough for our purposes.

The most Pythonic approach to type validation (or any validation task, really) is just to try to perform whatever task you need to do, detecting and handling any errors or exceptions that might result if the situation is somehow invalidan

approach known as "it's easier to ask forgiveness than permission" (EAFP). **try**/**except** is the key tool in enabling the EAFP style.

when you want to print a simple report with centered page numbers in a monospaced font. Because of this, Python string objects supply this functionality through three of their many methods. In Python 2.3, the padding character is always a space. In Python 2.4, however, while space-padding is still the default, you may optionally call any of these methods with a second argument, a single character to be used for the padding:

```
>print '|', 'hej'.ljust(20), '|', 'hej'.rjust(20), '|', 'hej'.center(20), '|'
| hej           |                  hej |      hej      |
>print 'hej'.center(20, '+')
++++++++hej+++++++++
```

You need to remove all whitespace (blanks, tabs, newlines, etc.) from either or both ends. Because this need is frequent, Python string objects supply this functionality through three of their many methods.

```
>>> x = '   hej   '
>>> print '|', x.lstrip(), '|', x.rstrip(), '|', x.strip(), '|'
| hej    |    hej | hej |
```

Optionally, you may call each of these methods with an argument, a string composed of all the characters you want to trim from either or both ends instead of trimming whitespace characters:

```
>>> x = 'xyxxyy hejyx  yyx'
>>> print '|'+x.strip('xy')+'|'
| hejyx  |
```

To join a sequence of small strings into one large string, use the string operator join. Say that pieces is a list whose items are strings, and you want one big string with all the items concatenated in order; then, you should code:

```
largeString = ''.join(pieces)
```

To put together pieces stored in a few variables, the string-formatting operator % can often be even handier:

```
largeString = '%s%s something %s yet more' % (small1, small2, small3)
```

In Python, the **+** operator concatenates strings and therefore offers seemingly obvious solutions for putting small strings together into a larger one.

```
largeString = small1 + small2 + ' something ' + small3 + ' yet more'
```

And similarly, when you have a sequence of small strings named **pieces**, it

seems quite natural to code something like:

```
largeString = ''
for piece in pieces:
    largeString += piece
```

Or, equivalently, but more fancifully and compactly:

```
import operator
largeString = reduce(operator.add, pieces, '')
```

In Python, string objects are immutable. Therefore, any operation on a string, including string concatenation, produces a new string object, rather than modifying an existing one. Concatenating **N** strings thus involves building and then immediately throwing away each of **N**-1 intermediate results. Performance is therefore vastly better for operations that build no intermediate results, but rather produce the desired end result at once.

Python's string-formatting operator **%** is one such operation, particularly suitable when you have a few pieces (e.g., each bound to a different variable) that you want to put together, perhaps with some constant text in addition. Performance is not a major issue for this specific kind of task. However, the **%** operator also has other potential advantages, when compared to an expression that uses multiple + operations on strings. % is more readable, once you get used to it.

Python does not have a specific type called **sequence**, but **sequence** is still an often-used term in Python. **sequence**, strictly speaking, means: a container that can be iterated on, to get a finite number of items, one at a time, and that also supports indexing, slicing, and being passed to the built-in function **len** (which gives the number of items in a container). Python **list**s are the "sequences" you'll meet most often, but there are many others (strings, unicode objects, tuples, **array.array**s, etc.).

Often, one does not need indexing, slicing, and **len**the ability to iterate, one item at a time, suffices. In that case, one should speak of an **iterable** (or, to focus on the finite number of items issue, a bounded iterable). Iterables that are not sequences include dictionaries (iteration gives the keys of the dictionary, one at a time in arbitrary order), file objects (iteration gives the lines of the text file, one at a time), and many more, including iterators and generators. Any iterable can be used in a **for** loop statement and in many equivalent contexts (the **for** clause of a list comprehension or Python 2.4 generator expression, and also many built-ins such as **min**, **max**, **zip**, **sum**, **str.join**, etc.).

At http://www.python.org/moin/PythonGlossary, you can find a Python Glossary that can help you with these and several other terms.

When you have many small string pieces in a sequence, performance can become a truly important issue. Fortunately, Python offers an excellent

alternative. The **join** method of a string object **s** takes as its only argument a sequence of strings and produces a string result obtained by concatenating all items in the sequence, with a copy of **s** joining each item to its neighbors. For example, **''.join(pieces)** concatenates all the items of **pieces** in a single gulp, without interposing anything between them, and **', '.join(pieces)** concatenates the items putting a comma and a space between each pair of them. It's the fastest, neatest, and most elegant and readable way to put a large string together.

When the pieces are not all available at the same time, but rather come in sequentially from input or computation, use a list as an intermediate data structure to hold the pieces (to add items at the end of a list, you can call the **append** or **extend** methods of the list). At the end, when the list of pieces is complete, call **''.join(thelist)** to obtain the big string that's the concatenation of all pieces. Of all the many handy tips and tricks I could give you about Python strings, I consider this one by far the most significant: the most frequent reason some Python programs are too slow is that they build up big strings with **+** or **+=**. So, train yourself never to do that. Use, instead, the **''.join** approach recommended in this recipe.

Python 2.4 makes a heroic attempt to ameliorate the issue, reducing a little the performance penalty due to such erroneous use of **+=**. Similarly, psyco (a specializing just-in-time [JIT] Python compiler found at http://psyco.sourceforge.net/), can reduce the **+=** penalty even further. Nevertheless, **''.join** remains the best approach in all cases.

Strings are immutable, so, to reverse one, we need to make a copy. The simplest approach for reversing is to take an extended slice with a "step" of -1, so that the slicing proceeds backwards:

```
revchars = astring[::-1]
```

To flip words, we need to make a list of words, reverse it, and join it back into a string with a space as the joiner:

```
revwords = astring.split()    # string -> list of words
revwords.reverse()           # reverse the list in place
revwords = ' '.join(revwords)  # list of strings -> string
```

or, if you prefer terse and compact "one-liners":

```
revwords = ' '.join(astring.split()[::-1])
```

If you need to reverse by words while preserving untouched the intermediate whitespace, you can split by a regular expression:

```
import re
revwords = re.split(r'(\s+)', astring)        # separators too, since '(...)'
revwords.reverse()       # reverse the list in place
```

```
revwords = ''.join(revwords)      # list of strings -> string
```

Note that the joiner must be the empty string in this case, because the whitespace separators are kept in the **revwords** list (by using **re.split** with a regular expression that includes a parenthesized group). Again, you could make a one-liner, if you wished:

```
revwords = ''.join(re.split(r'(\s+)', astring)[::-1])
```

In Python 2.4, you may make the by-word one-liners more readable by using the new built-in function **reversed** instead of the less readable extended-slicing indicator [::-1]:

```
revwords = ' '.join(reversed(astring.split()))
revwords = ''.join(reversed(re.split(r'(\s+)', astring)))
```

For the by-character case, though, astring[::-1] remains best, even in 2.4, because to use reversed, you'd have to introduce a call to ''.join as well:

```
revchars = ''.join(reversed(astring))
```

The new reversed built-in returns an iterator, suitable for looping on or for passing to some "accumulator" callable such as ''.join it does not return a ready-made string!

You need to check for the occurrence of any of a set of characters in a string. The simplest approach is clear, fast, and general (it works for any sequence, not just strings, and for any container on which you can test for membership, not just sets):

```
def containsAny(seq, aset):
    """ Check whether sequence seq contains ANY of the items in aset. """
    for c in seq:
        if c in aset: return True
    return False
```

A pure set-based approach would be something like:

```
def containsAny(seq, aset):
    return bool(set(aset).intersection(seq))
```

A term you can see often in discussions about programming is **predicate**: it just means a function (or other callable object) that returns **True** or **False** as its result. A predicate is said to be **satisfied** when it returns **true**.

If your application needs some function such as **containsAny** to check whether a string (or other sequence) contains any members of a set, you may also need such variants as:

```
def containsAll(seq, aset):
```

```
      """ Check whether sequence seq contains ONLY items in aset. """
      for c in seq:
         if c not in aset: return False
      return True
```

A pure set-based approach would be something like:

```
   def containsAll(seq, aset):
      """ Check whether sequence seq contains ALL the items in aset. """
      return not set(aset).difference(seq)
```

The **Translate** method of strings is quite powerful and flexible, it may be a nice idea to front it with a "facade" that simplifies its typical use.

```
   import string
   def translator(frm='', to='', delete='', keep=None):
      if len(to) == 1:
         to = to * len(frm)
      trans = string.maketrans(frm, to)
      if keep is not None:
         allchars = string.maketrans('', '')
         delete   =   allchars.translate(allchars,   keep.translate(allchars,
delete))
      def translate(s):
         return s.translate(trans, delete)
      return translate

   digits_only = translator(keep=string.digits)
   print digits_only('Chris Perkins : 224-7992')
   '2247992'
   no_digits = translator(delete=string.digits)
   print no_digits('Chris Perkins : 224-7992')
   'Chris Perkins : -'
   digits_to_hash = translator(frm=string.digits, to='#')
   print digits_to_hash('Chris Perkins : 224-7992')
   'Chris Perkins : ###-####'
   trans = translator(delete='abcd', keep='cdef')
   print trans('abcdefg')
   'ef'
```

A **closure** is nothing terribly complicated: just an "inner" function that refers to names (variables) that are local to an "outer" function containing it. Canonical toy-level example:

```
   def make_adder(addend):
```

```
    def adder(augend): return augend+addend
    return adder
```

Executing p = make_adder(23) makes a closure of inner function **adder** internally referring to a name **addend** that is bound to the value 23. Then, q = make_adder(42) makes another closure, for which, internally, name **addend** is instead bound to the value 42.

Calling **make_adder** a **factory** (or factory function) is both accurate and concise; you may also say it's a closure factory to specify it builds and returns closures.

The **TRanslate** method of string objects is fast and handy for all tasks of this ilk. However, to call **translate** effectively to solve this recipe's task, we must do some advance preparation. The first argument to **TRanslate** is a translation table: in this recipe, we do not want to do any translation, so we must prepare a first argument that specifies "no translation". The second argument to **TRanslate** specifies which characters we want to delete.

```
def makefilter(keep):
    """ Return a function that takes a string and returns a partial copy
        of that string consisting of only the characters in 'keep'.
        Note that `keep' must be a plain string.
    """
    # Make a reusable string of all characters, which does double duty
    # as a translation table specifying "no translation whatsoever"
    allchars = string.maketrans('', '')
    # Make a string of all characters that are not in 'keep': the "set
    # complement" of keep, meaning the string of characters we must delete
    delchars = allchars.translate(allchars, keep)
    # Make and return the desired filtering function (as a closure)
#   def thefilter(s):
#       return s.translate(allchars, delchars)
#   return thefilter
    return lambda s: s.translate(allchars, delchars)
```

TRanslate returns a copy of the string you call it on, replacing each character in it with the corresponding character in the translation table passed in as the first argument and deleting the characters specified in the second argument. **maketrans** is a utility function to create translation tables.

If you prefer, you could use **lambda** instead, changing the **def** and **return** statements in function **makefilter** into just one **return lambda** statement:

```
    return lambda s: s.translate(allchars, delchars)
```

Thanks to the **translate** method's power and speed, it's often faster to work directly on strings, rather than go through sets, for tasks of this ilk. However,

the functions in this recipe only work for normal strings, not for Unicode strings.

We can use the same heuristic criteria as Perl does, deeming a string binary if it contains any nulls or if more than 30% of its characters have the high bit set (i.e., codes greater than 126) or are strange control codes.

```
from __future__ import division         # ensure / does NOT truncate
import string

text_characters = "".join(map(chr, range(32, 127))) + "\n\r\t\b"
_null_trans = string.maketrans("", "")
def istext(s, text_characters=text_characters, threshold=0.30):
    # an "empty" string is "text" (arbitrary but reasonable choice):
    if not s:
        return True
    # if s contains any null, it's not text:
    if "\0" in s:
        return False
    # Get the substring of s made up of non-text characters
    t = s.translate(_null_trans, text_characters)
    # s is 'text' if less than 30% of its characters are non-text ones:
    return len(t)/len(s) <= threshold
```

Again, we can use the same heuristics as Perl, checking just the first block of the file with the **istext** function shown in this recipe's Solution:

```
def istextfile(filename, blocksize=512, **kwds):
    return istext(open(filename).read(blocksize), **kwds)
```

In some future version (probably Python 3.0, a few years away), Python will change the meaning of the **/** operator so that it performs division without truncationif you really do want truncation, you should use the truncating-division operator, **//**. it's handy for us to get the division behavior that is scheduled for introduction in some future release, we start our module with the statement:

```
from __future__ import division
```

You need to convert a string from uppercase to lowercase, or vice versa. That's what the upper and lower methods of string objects are for.

```
big = little.upper()
little = big.lower()
```

s.capitalize is similar to **s[:1].upper()+s[1:].lower()**: the first character is changed to uppercase, and all others are changed to lowercase. **s.title** is again similar, but it capitalizes the first letter of each word (where a "word" is a

sequence of letters) and uses lowercase for all other letters:

```
>>> print 'one tWo thrEe'.capitalize()
One two three
>>> print 'one tWo thrEe'.title()
One Two Three
```

Moreover, you can also check whether a string object is already in a given case form, with the methods **isupper**, **islower**, and **istitle.** There is no analogous **iscapitalized** method, and coding it is not trivial, if we want behavior that's strictly similar to strings' **is**... methods.

```
def iscapitalized(s):
    return s == s.capitalize()
```

Files

In Python, a file object is an instance of built-in type **file**. The built-in function **open** creates and returns a file object. The first argument, a string, specifies the file's path. The second argument to **open**, also a string, specifies the mode in which to open the file. For example:

```
input = open('data', 'r')
output = open('/tmp/spam', 'w')
```

For the mode argument, use '**r**' to read the file in text mode; this is the default value and is commonly omitted, so that **open** is called with just one argument. Other common modes are '**rb**' to read the file in binary mode, '**w**' to create and write to the file in text mode, and '**wb**' to create and write to the file in binary mode.

The distinction between text mode and binary mode is important on non-Unix-like platforms because of the line-termination characters used on these systems. When you open a file in binary mode, Python knows that it doesn't need to worry about line-termination characters; it just moves bytes between the file and in-memory strings without any kind of translation. When you open a file in text mode on a non-Unix-like system, however, Python knows it must translate between the '**\n**' line-termination characters used in strings and whatever the current platform uses in the file itself.

Nevertheless, it is good programming practice to close your files as soon as possible, and it is especially a good idea in larger programs, which otherwise may be at more risk of having excessive numbers of uselessly open files lying about. Note that **try**/**finally** is particularly well suited to ensuring that a file gets closed, even when a function terminates due to an uncaught exception.

The readline method reads and returns the next line from a text file. Consider the following loop:

```
while True:
    line = input.readline()
    if not line: break
    process(line)
```

This was once idiomatic Python but it is no longer the best way to read and process all of the lines from a file. Another dated alternative is to use the readlines method, which reads the whole file and returns a list of lines:

```
for line in input.readlines():
    process(line)
```

readlines is useful only for files that fit comfortably in physical memory. If the file is truly huge, readlines can fail or at least slow things down quite

drastically.

In today's Python, just loop on the file object itself to get a line at a time with excellent memory and performance characteristics:

```
for line in input:
    process(line)
```

Of course, you don't always want to read a file line by line. You may instead want to read some or all of the bytes in the file, particularly if you've opened the file for binary-mode reading, where lines are unlikely to be an applicable concept. In this case, you can use the read method. When called without arguments, read reads and returns all the remaining bytes from the file. When read is called with an integer argument N, it reads and returns the next N bytes.

Everywhere in Python, object interfaces, rather than specific data types, are the unit of coupling.

You want to read text or data from a file. Here's the most convenient way to read all of the file's contents at once into one long string:

```
all_the_text = open('thefile.txt').read()   # all text from a text file
all_the_data = open('abinfile', 'rb').read() # all data from a binary file
```

However, it is safer to bind the file object to a name, so that you can call close on it as soon as you're done, to avoid ending up with open files hanging around. For example, for a text file:

```
file_object = open('thefile.txt')
try:
    all_the_text = file_object.read()
finally:
    file_object.close()
```

You don't necessarily have to use the try/finally statement here, but it's a good idea to use it, because it ensures the file gets closed even when an error occurs during reading.

The simplest, fastest, and most Pythonic way to read a text file's contents at once as a list of strings, one per line, is:

```
list_of_all_the_lines = file_object.readlines()
```

This leaves a '\n' at the end of each line; if you don't want that, you have alternatives, such as:

```
list_of_all_the_lines = file_object.read().splitlines()
list_of_all_the_lines = file_object.read().split('\n')
list_of_all_the_lines = [L.rstrip('\n') for L in file_object]
```

The simplest and fastest way to process a text file one line at a time is simply to loop on the file object with a for statement:

```
for line in file_object:
    process line
```

This approach also leaves a '\n' at the end of each line; you may remove it by starting the for loop's body with:

```
    line = line.rstrip('\n')
```

or even, when you're OK with getting rid of trailing whitespace from each line (not just a trailing '\n'), the generally handier:

```
    line = line.rstrip()
```

Unless the file you're reading is truly huge, slurping it all into memory in one gulp is often fastest and most convenient for any further processing. The built-in function **open** creates a Python file object (alternatively, you can equivalently call the built-in type **file**). You call the **read** method on that object to get all of the contents (whether text or binary) as a single long string. If the contents are text, you may choose to immediately split that string into a list of lines with the **split** method or the specialized **splitlines** method. Since splitting into lines is frequently needed, you may also call **readlines** directly on the file object for faster, more convenient operation.

On Unix and Unix-like systems, such as Linux, Mac OS X, and other BSD variants, there is no real distinction between text files and binary data files. On Windows and very old Macintosh systems, however, line terminators in text files are encoded, not with the standard '**\n**' separator, but with '**\r\n**' and '**\r**', respectively. Python translates these line-termination characters into '**\n**' on your behalf. This means that you need to tell Python when you open a binary file, so that it won't perform such translation. To do so, use '**rb**' as the second argument to **open**. This is innocuous even on Unix-like platforms, and it's a good habit to distinguish binary files from text files even there, although it's not mandatory in that case. Such good habits will make your programs more immediately understandable, as well as more compatible with different platforms.

If you're unsure about which line-termination convention a certain text file might be using, use '**rU**' as the second argument to **open**, requesting universal endline translation. This lets you freely interchange text files among Windows, Unix (including Mac OS X), and old Macintosh systems, without worries: all kinds of line-ending conventions get mapped to '**\n**', whatever platform your code is running on.

You can call methods such as **read** directly on the file object produced by the

open function. When you do so, you no longer have a reference to the file object as soon as the reading operation finishes. In practice, Python notices the lack of a reference at once, and immediately closes the file. However, it is better to bind a name to the result of **open**, so that you can call **close** yourself explicitly when you are done with the file. This ensures that the file stays open for as short a time as possible, even on platforms such as Jython, IronPython.

Ensure that a file object is closed even if errors happen during its processing, the most solid and prudent approach is to use the **try**/**finally** statement:

```
file_object = open('thefile.txt')
try:
    for line in file_object:
        process line
finally:
    file_object.close()
```

If you choose to read the file a little at a time, rather than all at once, the idioms are different. Here's one way to read a binary file 100 bytes at a time, until you reach the end of the file:

```
file_object = open('abinfile', 'rb')
try:
    while True:
        chunk = file_object.read(100)
        if not chunk:
            break
        do_something_with(chunk)
finally:
    file_object.close()
```

Complicated loops are best encapsulated as reusable generators. In this case, we can encapsulate the logic only partially, because a generator's **yield** keyword is not allowed in the **try** clause of a **try**/**finally** statement.

```
def read_file_by_chunks(filename, chunksize=100):
    file_object = open(filename, 'rb')
    while True:
        chunk = file_object.read(chunksize)
        if not chunk:
            break
        yield chunk
    file_object.close()
```

Reading a text file one line at a time is a frequent task. Just loop on the file object, as in:

```
for line in open('thefile.txt', 'rU'):
```

```
    do_something_with(line)
```

In order to be 100% certain that no uselessly open file object will ever be left just hanging around, you may want to code this snippet in a more rigorously correct and prudent way:

```
file_object = open('thefile.txt', 'rU'):
try:
    for line in file_object:
        do_something_with(line)
finally:
    file_object.close()
```

You want to write text or data to a file. Here is the most convenient way to write one long string to a file:

```
open('thefile.txt', 'w').write(all_the_text)  # text to a text file
open('abinfile', 'wb').write(all_the_data)    # data to a binary file
```

Often, the data you want to write is not in one big string, but in a list (or other sequence) of strings. In this case, you should use the **writelines** method.

```
file_object.writelines(list_of_text_strings)
open('abinfile', 'wb').writelines(list_of_data_strings)
```

Calling **writelines** is much faster than the alternatives of joining the strings into one big string.

You want to read from a text file a single line, given the line number.

```
import linecache
theline = linecache.getline(thefilepath, desired_line_number)
```

linecache is particularly useful when you have to perform this task repeatedly for several lines in a file, since **linecache** caches information to avoid uselessly repeating work. When you know that you won't be needing any more lines from the cache for a while, call the module's **clearcache** function to free the memory used for the cache. You can also use **checkcache** if the file may have changed on disk and you must make sure you are getting the updated version.

linecache reads and caches all of the text file whose name you pass to it, so, if it's a very large file and you need only one of its lines, **linecache** may be doing more work than is strictly necessary.

```
def getline(thefilepath, desired_line_number):
    if desired_line_number < 1: return ''
    for current_line_number, line in enumerate(open(thefilepath, 'rU')):
```

```
            if current_line_number == desired_line_number-1: return line
        return ''
```

enumerate counts from 0, so, since we assume the **desired_line_number** argument counts from 1, we need the **-1** in the **==** comparison.

You need to compute the number of lines in a file. The simplest approach for reasonably sized files is to read the file as a list of lines, so that the count of lines is the length of the list.

```
count = len(open(thefilepath, 'rU').readlines())
```

If the file is larger than available memory, the simplest solution can become unacceptably slow, as the operating system struggles to fit the file's contents into virtual memory. A loop on the file always works:

```
count = -1
for count, line in enumerate(open(thefilepath, 'rU')):
    pass
count += 1
```

You need to do something with each and every word in a file.

```
for line in open(thefilepath):
    for word in line.split():
        dosomethingwith(word)
```

For other definitions of words, you can use regular expressions.

```
import re
re_word = re.compile(r"[\w'-]+")
for line in open(thefilepath):
    for word in re_word.finditer(line):
        dosomethingwith(word.group(0))
```

In this case, a word is defined as a maximal sequence of alphanumerics, hyphens, and apostrophes.

zip files are a popular, cross-platform way of archiving files. The Python Standard Library comes with a **zipfile** module to access such files easily:

```
import zipfile
z = zipfile.ZipFile("zipfile.zip", "r")
for filename in z.namelist():
    print 'File:', filename,
    bytes = z.read(filename)
    print 'has', len(bytes), 'bytes'
```

Your program receives a **zip** file as a string of bytes in memory, and you need to read the information in this **zip** file.

Module **cStringIO** lets you wrap a string of bytes so it can be accessed as a file object.

```
import cStringIO, zipfile
class ZipString(ZipFile):
    def __init__(self, datastring):
        ZipFile.__init__(self, cStringIO.StringIO(datastring))
```

Not to worry: the Python Standard Library always includes module **StringIO**, which is coded in pure Python (and thus is usable from any compliant implementation of Python), and implements the same functionality as module **cStringIO.**

You need to archive all of the files and folders in a subtree into a **tar** archive file, compressing the data with either the popular **gzip** approach or the higher-compressing **bzip2** approach. The Python Standard Library's **tarfile** module directly supports either kind of compression.

```
import tarfile, os
def make_tar(folder_to_backup, dest_folder, compression='bz2'):
    if compression:
        dest_ext = '.' + compression
    else:
        dest_ext = ''
    arcname = os.path.basename(folder_to_backup)
    dest_name = '%s.tar%s' % (arcname, dest_ext)
    dest_path = os.path.join(dest_folder, dest_name)
    if compression:
        dest_cmp = ':' + compression
    else:
        dest_cmp = ''
    out = tarfile.TarFile.open(dest_path, 'w'+dest_cmp)
    out.add(folder_to_backup, arcname)
    out.close()
    return dest_path
```

The string '**gz**' to get **gzip** compression instead of the default **bzip2**, or you can pass the empty string '' to get no compression at all. Besides making the file extension of the result either **.tar**, **.tar.gz**, or **.tar.bz2**, as appropriate, your choice for the **compression** argument determines which string is passed as the second argument to **tarfile.TarFile.open**: '**w**', when you want no compression, or '**w:gz**' or '**w:bz2**' to get two kinds of compression.

Once we have an instance of class **tarfile.TarFile** that is set to use the kind of

compression we desire, the instance's method **add** does all we require. In particular, when string **folder_to_backup** names a "directory", **add** recursively adds all of the subtree rooted in that directory. If we wanted to change this behavior to get precise control on what is archived, we could pass to **add** an additional named argument **recursive=False** to switch off this implicit recursion.

You need to examine a "directory", or an entire directory tree rooted in a certain directory, and iterate on the files (and optionally folders) that match certain patterns.

The generator **os.walk** from the Python Standard Library module **os** is sufficient for this task, but we can dress it up a bit by coding our own function to wrap **os.walk.**

```
import os, fnmatch
def all_files(root, patterns='*', single_level=False, yield_folders=False):
    # Expand patterns from semicolon-separated string to list
    patterns = patterns.split(';')
    for path, subdirs, files in os.walk(root):
        if yield_folders:
            files.extend(subdirs)
        files.sort()
        for name in files:
            for pattern in patterns:
                if fnmatch.fnmatch(name, pattern):
                    yield os.path.join(path, name)
                    break
        if single_level:
            break
```

To specify multiple patterns, join them with a semicolon. Note that this means that semicolons themselves can't be part of a pattern.

you can easily get a list of all Python and HTML files in directory **/tmp** or any subdirectory thereof:

```
thefiles = list(all_files('/tmp', '*.py;*.htm;*.html'))

for path in all_files('/tmp', '*.py;*.htm;*.html'):
    print path
```

If your platform is case-sensitive, and you want case-sensitive matching, then you need to specify the patterns more laboriously, e.g., '*.[Hh][Tt][Mm][Ll]' instead of just '*.html'.

You need to rename files throughout a subtree of directories, specifically

changing the names of all files with a given extension so that they have a different extension instead.

```
import os
def swapextensions(dir, before, after):
    if before[:1] != '.':
        before = '.' + before
    thelen = -len(before)
    if after[:1] != '.':
        after = '.' + after
    for path, subdirs, files in os.walk(dir):
        for oldfile in files:
            if oldfile[thelen:] == before:
                oldfile = os.path.join(path, oldfile)
                newfile = oldfile[:thelen] + after
                os.rename(oldfile, newfile)
```

This technique is useful for changing the extensions of a whole batch of files in a folder structure, such as a web site.

Given a search path (a string of directories with a separator in between), you need to find the first file along the path with the requested name.

```
import os
def search_file(filename, search_path, pathsep=os.pathsep):
    """ Given a search path, find file with requested name """
    for path in search_path.split(pathsep):
        candidate = os.path.join(path, filename)
        if os.path.isfile(candidate):
            return os.path.abspath(candidate)
    return None

search_path = '/bin' + os.pathsep + '/usr/bin'
find_file = search_file('ls', search_path)
if find_file:
    print "File 'ls' found at %s" % find_file
```

The explicit **return None** after the loop is not strictly needed, since **None** is returned by Python when a function falls off the end.

You want to extract the text content from each Microsoft Word document in a directory tree on Windows into a corresponding text file.

With the PyWin32 extension, we can access Word itself, through COM, to perform the conversion.

```
import fnmatch, os, sys, win32com.client
wordapp = win32com.client.gencache.EnsureDispatch("Word.Application")
try:
    for path, dirs, files in os.walk(sys.argv[1]):
        for filename in files:
            if not fnmatch.fnmatch(filename, '*.doc'): continue
            doc = os.path.abspath(os.path.join(path, filename))
            print "processing %s" % doc
            wordapp.Documents.Open(doc)
            docastxt = doc[:-3] + 'txt'
            wordapp.ActiveDocument.SaveAs(docastxt,
                FileFormat=win32com.client.constants.wdFormatText)
            wordapp.ActiveDocument.Close()
finally:
    # ensure Word is properly shut down even if we get an exception
    wordapp.Quit()
```

A useful aspect of most Windows applications is that you can script them via COM, and the PyWin32 extension makes it fairly easy to perform COM scripting from Python. The extension enables you to write Python scripts to perform many kinds of Window tasks.

You want to make a backup copy of a file, before you overwrite it, with the standard convention of appending a three-digit version number to the name of the old file.

```
def versionFile(file_spec, vtype='copy'):
    import os, shutil
    if os.path.isfile(file_spec):
        # check the 'vtype' parameter
        if vtype not in ('copy', 'rename'):
            raise ValueError, 'Unknown vtype %r' % (vtype,)

        # Determine root filename so the extension doesn't get longer
        n, e = os.path.splitext(file_spec)

        # Is e a three-digits integer preceded by a dot?
        if len(e) == 4 and e[1:].isdigit():
            num = 1 + int(e[1:])
            root = n
        else:
            num = 0
            root = file_spec

        # Find next available file version
```

```
            for i in xrange(num, 1000):
                new_file = '%s.%03d' % (root, i)
                if not os.path.exists(new_file):
                    if vtype == 'copy':
                        shutil.copy(file_spec, new_file)
                    else:
                        os.rename(file_spec, new_file)
                    return True
            raise RuntimeError, "Can't %s %r, all names taken" % (vtype,file_spec)
        return False
```

Time and Money

Even very simple programs may have to deal with timestamps, delays, timeouts, speed gauges, calendars, and so on. As befits a general-purpose language that is proud to come with "batteries included," Python's standard library offers solid support for these application needs, and more support yet comes from third-party modules and packages.

Python 2.4 introduced support for decimal numbers, making Python a good option even for computations where you must avoid using binary floats, as ones involving money so often are.

Python Standard Library's **time** module lets Python applications access a good portion of the time-related functionality offered by the platform Python is running on.

Here is a way to unpack a tuple representing the current local time:

```
year, month, mday, hour, minute, second, wday, yday, isdst = time.localtime()
```

Obtaining the current month then becomes a simple and elegant expression:

```
time.localtime().tm_mon
```

Two very useful functions in module **time** are **strftime**, which lets you build a string from a time tuple, and **strptime**, which goes the other way, parsing a string and producing a time tuple.

One last important function in module **time** is the **time.sleep** function, which lets you introduce delays in Python programs. Even though this function's POSIX counterpart accepts only an integer parameter, the Python equivalent supports a float and allows sub-second delays to be achieved.

```
for i in range(10):
    time.sleep(0.5)
    print "Tick!"
```

While module **time** is quite useful, the Python Standard Library also includes the **datetime** module, which supplies types that provide better abstractions for the concepts of dates and timesnamely, the types **time**, **date**, and **datetime**.

```
today = datetime.date.today()
birthday = datetime.date(1977, 5, 4)     #May 4
currenttime = datetime.datetime.now().time()
lunchtime = datetime.time(12, 00)
```

```
now = datetime.datetime.now()
epoch = datetime.datetime(1970, 1, 1)
meeting = datetime.datetime(2005, 8, 3, 15, 30)

today = datetime.date.today()
next_year = today.replace(year=today.year+1).strftime("%Y.%m.%d")
print next_year
```

Module **datetime** also provides basic support for **time deltas**, through the **timedelta** type.

```
import datetime
NewYearsDay = datetime.date(2005, 01, 01)
NewYearsEve = datetime.date(2004, 12, 31)
oneday = NewYearsDay - NewYearsEve
print oneday
1 day, 0:00:00
```

decimal is a Python Standard Library module, new in Python 2.4, which finally brings decimal arithmetic to Python. Thanks to **decimal**, we now have a decimal numeric data type, with bounded precision and floating point.

The number is not stored in binary, but rather, as a sequence of decimal digits. The number of digits each number stores is fixed. The decimal point does not have a fixed place.

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
>>> d1 = decimal.Decimal("0.1")
>>> d3 = decimal.Decimal("0.3")
>>> d1 + d1 + d1 - d3
Decimal("0.0")

>>> 0.9 / 10
0.089999999999999997
>>> decimal.Decimal("0.9") / decimal.Decimal(10)
Decimal("0.09")
```

decimal.Decimal instances can be constructed from integers, strings, or tuples. To create a **decimal.Decimal** from a **float**, first convert the **float** to a string. Decimal numbers include special values such as **NaN**, positive and negative **Infinity**, and **-0**. Once constructed, a **decimal.Decimal** object is immutable, just like any other number in Python.

The **decimal** module incorporates the notion of **significant digits**, so that, for example, 1.30+1.20 is 2.50. The trailing zero is kept to indicate significance. This is the usual representation for monetary applications.

```
>>> decimal.Decimal("1.3") * decimal.Decimal("1.2")
Decimal("1.56")
>>> decimal.Decimal("1.30") * decimal.Decimal("1.20")
Decimal("1.5600")
```

The **decimal** data type works within a context, where some configuration aspects are set. Each thread has its own current context; the current thread's current context is accessed or changed using functions **getcontext** and **setcontext** from the **decimal** module.

```
>>> decimal.getcontext().prec = 6          # set the precision to 6...
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857")
>>> decimal.getcontext().prec = 60         # ...and to 60 digits
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857142857142857142857142857142857142857142857142857142857")
```

decimal implements the standards for general decimal arithmetic which you can study in detail at http://www2.hursley.ibm.com/decimal/.

If your application must sum many tens of millions of noninteger numbers, you had better stick with **float**! Fortunately, relatively few applications actually need to perform so much arithmetic on non-integers as to give any observable performance problems on today's typical machines. Thus, today, most applications can actually take advantage of **decimal**'s many beneficial aspects.

Whenever you have to deal with a "change" or "difference" in time, think **timedelta**:

```
import datetime
today = datetime.date.today()
yesterday = today - datetime.timedelta(days=1)
tomorrow = today + datetime.timedelta(days=1)
print yesterday, today, tomorrow
#emits: 2004-11-17 2004-11-18 2004-11-19
```

When first confronted with this task, it's quite common for people to try to code it as **yesterday = today - 1**, which gives a **TypeError.** Rather than trying to guess what you mean, Python, as usual, expects you to make your meaning explicit. If you want to subtract a time difference of one day, you code that explicitly.

Keep in mind that, if you want fancier control over date and time arithmetic,

third-party packages, such as **dateutil** and the classic **mx.DateTime**, are available.

```
from dateutil import relativedelta
nextweek = today + relativedelta.relativedelta(weeks=1)
print nextweek
#emits: 2004-11-25
```

You want to find the date of last Friday.

```
import datetime, calendar
lastFriday = datetime.date.today()
oneday = datetime.timedelta(days=1)
while lastFriday.weekday() != calendar.FRIDAY:
    lastFriday -= oneday
print lastFriday.strftime('%A, %d-%b-%Y')
# emits, e.g.: Friday, 10-Dec-2004
```

You need to run a command repeatedly, with arbitrary periodicity.

```
import time, os, sys
def run_repeatedly(cmd, inc=60):
    while True:
        os.system(cmd)
        time.sleep(inc)
```

Binary floating-point arithmetic is the default in Python for very good reasons. You can read all about them in the Python FAQ (Frequently Asked Questions) document at http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate

Many people, however, were unsatisfied with binary floats being the only option they wanted to be able to specify the precision, or wanted to use decimal arithmetic for monetary calculations with predictable results.

The new **decimal** type affords a great deal of control over the **context** for your calculations, allowing you to set the precision and rounding method to use for the results. When all you want is to run simple arithmetical operations that return predictable results, **decimal**'s default context works just fine.

Just keep in mind a few points: you may pass a string, integer, tuple, or other decimal object to create a new **decimal** object, but if you have a float **n** that you want to make into a **decimal**, pass **str(n)**, not bare **n**.

Also, **decimal** objects can interact (arithmetical operations) with integers,

longs, and other **decimal** objects, but not with **float**s.

Decimal numbers have been added to Python exactly to provide the precision and predictability that **float** lacks.

Keep in mind that decimal is still floating point, not fixed point. Also, no **money** data type is yet available in Python.

You want to use Python as a simple adding machine, with accurate decimal.

```
import decimal, re, operator

parse_input = re.compile(r'''(?x)  # allow comments and whitespace in the RE
            (\d+\.?\d*)         # number with optional decimal part
            \s*                # optional whitespace
            ([-+/*])           # operator
            $''')              # end-of-string

oper = { '+': operator.add, '-': operator.sub,
         '*': operator.mul, '/': operator.truediv,
       }

total = decimal.Decimal('0')

def print_total():
    print '=====\n', total

print """Welcome to Adding Machine:
Enter a number and operator,
an empty line to see the current subtotal,
or q to quit: """

while True:
    try:
        tape_line = raw_input().strip()
    except EOFError:
        tape_line = 'q'
    if not tape_line:
        print_total()
        continue
    elif tape_line == 'q':
        print_total()
        break
    try:
        num_text, op = parse_input.match(tape_line).groups()
```

```
        except AttributeError:
            print 'Invalid entry: %r' % tape_line
            print 'Enter number and operator, empty line for total, q to quit'
            continue
        total = oper[op](total, decimal.Decimal(num_text))
```

Shortcuts

Programming languages are like natural languages. Each has a set of qualities that polyglots generally agree on as characteristics of the language. Russian and French are often admired for their lyricism, while English is more often cited for its precision and dynamism.

Perl is well known for its many degrees of freedom: TMTOWTDI (There's More Than One Way To Do It) is one of the mantras of the Perl programmer. Conciseness is also seen as a strong virtue in the Perl and APL communities. As you'll see in many of the discussions of recipes throughout this volume, in contrast, Python programmers often express their belief in the value of clarity and elegance. As a well-known Perl hacker once told me, Python's prettier, but Perl is more fun. I agree with him that Python does have a strong (as in well-defined) aesthetic, while Perl has more of a sense of humor.

The reason I mention these seemingly irrelevant characteristics at the beginning of this chapter is that the recipes you see in this chapter are directly related to Python's aesthetic and social dynamics. In most of the recipes in this chapter, the author presents a single elegant language feature, but one that he feels is underappreciated. For me and most of the programmers I know, programming in Python is a shared social pleasure, not a competitive pursuit. There is great pleasure in learning a new feature and appreciating its design, elegance, and judicious use, and there's a twin pleasure in teaching another or another thousand about that feature.

Module copy in the standard Python library offers two functions to create copies. The one you should generally use is the function named copy, which returns a new object containing exactly the same items and attributes as the object you're copying:

```
import copy
new_list = copy.copy(existing_list)
```

On the rare occasions when you also want every item and attribute in the object to be separately copied, recursively, use deepcopy:

```
import copy
new_list_of_dicts = copy.deepcopy(existing_list_of_dicts)
```

Normally, you use **copy.copy**, which makes a **shallow copy**it copies an object, but for each attribute or item of the object, it continues to share references, which is faster and saves memory. If you do need to copy some container object and also recursively copy all objects it refers to (meaning all items, all attributes, and also items of items, items of attributes, etc.), use **copy.deepcopy**such deep copying may cost you substantial amounts of time and memory.

To copy a list **L**, call **list(L)**; to copy a dict **d**, call **dict(d)**; to copy a set **s**, call **set(s)**. You get the general pattern: to copy a copyable object **o,** which belongs to some built-in Python type **t**, you may generally just call **t(o)**.

You can use other, inferior ways exist to create copies, namely building your own. Given a list **L**, both a "whole-object slice" **L[:]** and a list comprehension **[x for x in L]** do happen to make a (shallow) copy of **L**, as do adding an empty list, **L+[ ]**, and multiplying the list by 1, **L*1** . . . but each of these constructs is just wasted effort and obfuscationcalling. **list(L)** is clearer and faster. You should, however, be familiar with the **L[:]** construct because for historical reasons it's widely used.

Note that you do not need to copy immutable objects (strings, numbers, tuples, etc.) because you don't have to worry about altering them. If you do try to perform such a copy, you'll just get the original right back; no harm done, but it's a waste of time and code.

```
>>> s = 'cat'
>>> t = copy.copy(s)
>>> s is t
True
```

The **is** operator checks whether two objects are not merely equal, but in fact the same object (**is** checks for identity; for checking mere equality, you use the **==** operator). Checking object identity is not particularly useful for immutable objects.

Elegance, clarity, and pragmatism, are Python's core values. List comprehensions show how pragmatism can enhance both clarity and elegance. Indeed, list comprehensions are often the best approach even when, instinctively, you're thinking not of constructing a new list but rather of "altering an existing list". For example, if your task is to set all items greater than 100 to 100, in an existing list object **L**, the best solution is:

```
L[:] = [min(x,100) for x in L]
```

Assigning to the "whole-list slice" **L[:]** alters the existing list object in place, rather than just rebinding the name **L.**

You should not use a list comprehension when you simply want to perform a loop. When you want a loop, code a loop.

It's also best not to use a list comprehension when another built-in does what you want even more directly and immediately. For example, to copy a list, use **L1 = list(L)**, not:

```
L1 = [x for x in L]
```

Similarly, when the operation you want to perform on each item is to call a

function on the item and use the function's result, use **L1 = map(f, L)** rather than **L1 = [f(x) for x in L]**. But in most cases, a list comprehension is just right.

In Python 2.4, you should consider using a generator expression, rather than a list comprehension, when the sequence may be long and you only need one item at a time. The syntax of generator expressions is just the same as for list comprehensions, except that generator expressions are surrounded by parentheses, **(** and **)**, not brackets, **[** and **]**.

For example, say that we only need the summation of the list computed in this recipe's Solution, not each item of the list. In Python 2.3, we would code:

```
total = sum([x + 23 for x in theoldlist if x > 5])
```

In Python 2.4, we can code more naturally, omitting the brackets (no need to add additional parenthesesthe parentheses already needed to call the built-in sum suffice):

```
total = sum(x + 23 for x in theoldlist if x > 5)
```

Besides being a little bit cleaner, this method avoids materializing the list as a whole in memory and thus may be slightly faster when the list is extremely long.

Python borrowed list comprehensions from the functional language Haskell (http://www.haskell.org), changing the syntax to use keywords rather than punctuation. If you do know Haskell, though, take care! Haskell's list comprehensions, like the rest of Haskell, use lazy evaluation (also known as normal order or call by need). Each item is computed only when it's needed. Python, like most other languages, uses (for list comprehensions as well as elsewhere) eager evaluation (also known as applicative order, call by value, or strict evaluation). That is, the entire list is computed when the list comprehension executes, and kept in memory afterwards as long as necessary.

If you are translating into Python a Haskell program that uses list comprehensions to represent infinite sequences, or even just long sequences of which only one item at a time must be kept around, Python list comprehensions may not be suitable. Rather, look into Python 2.4's new **generator expressions**, whose semantics are closer to the spirit of Haskell's lazy evaluation. Each item gets computed only when needed.

You have a list **L** and an index **i**, and you want to get **L[i]** when **i** is a valid index into **L**; otherwise, you want to get a default value **v**. If **L** were a dictionary, you'd use **L.get(i, v)**, but lists don't have a **get** method.

Clearly, we need to code a function, and, in this case, the simplest and most direct approach is the best one:

```
def list_get(L, i, v=None):
```

```
    if -len(L) <= i < len(L): return L[i]
    else: return v
```

Python's indexing rule: valid indices are negative ones down to **-len(L)** inclusive, and non-negative ones up to **len(L)** exclusive.

If almost all calls to **list_get** pass a valid index value for **i**, you might prefer an alternative approach:

```
def list_get_egfp(L, i, v=None):
    try: return L[i]
    except IndexError: return v
```

This alternative (as some time-measurements show) can be up to four times slower than the **list_get** function shown in the solution. Therefore, this "easier to get forgiveness than permission" (EGFP) approach, although it is often preferable in Python, cannot be recommended for this specific case.

General principle: when you write Python code, prefer clarity and readability to compactness and terseness, choose simplicity over subtlety. You often will be rewarded with code that runs faster, and invariably, you will end up with code that is less prone to bugs and is easier to maintain, which is far more important than minor speed differences in 99.9% of the cases you encounter in the real world.

You need to loop on a sequence, but at each step you also need to know which index into the sequence you have reached.

That's what built-in function enumerate is for. For example:

```
for index, item in enumerate(sequence):
    if item > 23:
        sequence[index] = transform(item)
```

This is cleaner, more readable, and faster than the alternative of looping over indices and accessing items by indexing:

```
for index in range(len(sequence)):
    if sequence[index] > 23:
        sequence[index] = transform(sequence[index])
```

Looping on a sequence is a very frequent need, and Python strongly encourages you to do just that, looping on the sequence directly. In other words, the Pythonic way to get each item in a sequence is to use:

```
for item in sequence:
    process(item)
```

Looping directly is cleaner, more readable, faster, and more general. However, sometimes you do need to know the index, as well as the corresponding item, within the loop. The most frequent reason for this need is

that, in order to rebind an entry in a list, you must assign the new item to **thelist[index]**. To support this need, Python offers the built-in function **enumerate**, which takes any iterable argument and returns an iterator yielding all the pairs (two-item tuples) of the form (**index**, **item**), one pair at a time.

You want to create a multidimensional list but want to avoid implicit reference sharing. To build a list and avoid implicit reference sharing, use a list comprehension. For example, to build a 5 x 10 array of zeros:

```
multilist = [[0 for col in range(5)] for row in range(10)]
multilist = [[0]*5 for row in range(10)]
```

With list comprehensions, no sharing of references occursyou have a truly nested computation.

You need to loop over a "flattened" sequence, "expanding" each sub-sequence into a single, flat sequence of scalar items. We need to be able to tell which of the elements we're handling are "subsequences" to be "expanded" and which are "scalars" to be yielded as is. For generality, we can take an argument that's a **predicate** to tell us what items we are to expand.

```
def list_or_tuple(x):
    return isinstance(x, (list, tuple))

def nonstring_iterable(obj):
    try: iter(obj)
    except TypeError: return False
    else: return not isinstance(obj, basestring)

def flatten(sequence, to_expand=list_or_tuple):
    for item in sequence:
        if to_expand(item):
            for subitem in flatten(item, to_expand):
                yield subitem
        else:

for x in flatten([1, 2, [3, [  ], 4, (5, 6), 7, [8,], ], 9]):
    print x,
```

You have a list of lists (rows) and need to get another list of the same rows but with some columns removed and/or reordered.

An alternative way of coding, that is at least as practical and arguably a bit more elegant, is to use an auxiliary sequence (meaning a list or tuple) that has the column indices you desire in their proper order. Then, you can nest an inner

list comprehension that loops on the auxiliary sequence inside the outer list comprehension that loops on **listOfRows**:

```
listOfRows = [ [1,2,3,4], [5,6,7,8], [9,10,11,12] ]
newList = [ [row[ci] for ci in (0, 3, 2)] for row in listofRows ]
```

The possibility of using an auxiliary sequence to hold the column indices you desire, in the order in which you desire them, rather than explicitly hard-coding the list.

A list comprehension builds a new list, rather than altering an existing one. But even when you do need to alter the existing list in place, the best approach is to write a list comprehension and assign it to the existing list's contents.

```
listOfRows[:] = [ [row[0], row[3], row[2]] for row in listOfRows ]
```

If you adopt this approach, you gain some potential generality, because you can choose to give a name to the auxiliary sequence of indices, use it to reorder several lists of rows in the same fashion, pass it as an argument to a function.

***args** is Python syntax for accepting or passing arbitrary positional arguments. When you receive arguments with this syntax, Python binds the identifier to a tuple that holds all positional arguments not "explicitly" received. When you pass arguments with this syntax, the identifier can be bound to any iterable.

****kwds** is Python syntax for accepting or passing arbitrary named arguments. When you receive arguments with this syntax, Python binds the identifier to a **dict**, which holds all named arguments not "explicitly" received. When you pass arguments with this syntax, the identifier must be bound to a **dict**.

Whether in defining a function or in calling it, make sure that both ***a** and ****k** come after any other parameters or arguments. If both forms appear, then place the ****k** after the ***a**.

You need to transpose a list of lists, turning rows into columns and vice versa.

A faster though more obscure alternative (with exactly the same output) can be obtained by exploiting built-in function **zip** in a slightly strange way:

```
arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
print map(list, zip(*arr))
--> [[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

If you're transposing large arrays of numbers, consider Numeric Python and other third-party packages. Numeric Python defines transposition and other axis-swinging routines that will make your head spin.

You need to obtain a value from a dictionary, without having to handle an exception if the key you seek is not in the dictionary.

Say you have a dictionary such as **d = {'key':'value',}**. To get the value corresponding to **key** in **d** in an exception-safe way, code:

```
print d.get('key', 'not found')
```

Another similar method is **pop**, which is mostly like **get**, except that, if the key was in the dictionary, **pop** also removes it. **d.pop(x)** does raise **KeyError** if **x** is not a key in **d**; to get exactly the same effect as **d.get(x)**, plus the entry removal, call **d.pop(x,None)** instead.

Working with a dictionary **d**, you need to use the entry **d[k]** when it's already present, or add a new value as **d[k]** when **k** isn't yet a key in **d**. This is what the **setdefault** method of dictionaries is for.

```
def addword(theIndex, word, pagenumber):
    theIndex.setdefault(word, []).append(pagenumber)
```

This code is equivalent to more verbose approaches such as:

```
def addword(theIndex, word, pagenumber):
    if word in theIndex:
        theIndex[word].append(pagenumber)
    else:
        theIndex[word] = [pagenumber]
```

and:

```
def addword(theIndex, word, pagenumber):
    try:
        theIndex[word].append(pagenumber)
    except KeyError:
        theIndex[word] = [pagenumber]
```

setdefault is not all that useful for immutable values, such as numbers. If you just want to count words, for example, the right way to code is to use, not **setdefault**, but rather **get**:

```
theIndex[word] = theIndex.get(word, 0) + 1
```

You want to construct a dictionary whose keys are literal strings, without having to quote each key.

When the keys are identifiers, you can avoid quoting them by calling **dict** with named-argument syntax:

```
data = dict(red=1, green=2, blue=3)
```

This is neater than the equivalent use of dictionary-display syntax:

```
data = {'red': 1, 'green': 2, 'blue': 3}
```

One powerful way to build a dictionary is to call the built-in type **dict**. It's often a good alternative to the dictionary-display syntax with braces and colons.

A common dictionary-building idiom is:

```
d = dict(zip(the_keys, the_values))
```

where **the_keys** is a sequence of keys and **the_values** a "parallel" sequence of corresponding values. Built-in function zip builds and returns a list of (key, value) pairs, and built-in type dict accepts that list as its argument and constructs a dictionary accordingly.

If the sequences are long, it's faster to use module itertools from the standard Python library:

```
import itertools
d = dict(itertools.izip(the_keys, the_values))
```

Built-in function **zip** constructs the whole list of pairs in memory, while **itertools.izip** yields only one pair at a time.

If you need to build a dictionary where the same value corresponds to each key, call **dict.fromkeys(keys_sequence, value)** (if you omit the **value**, it defaults to **None**).

```
import string
count_by_letter = dict.fromkeys(string.ascii_lowercase, 0)
```

If you want to leave the original dictionary intact:

```
def sub_dict(somedict, somekeys, default=None):
   return dict([ (k, somedict.get(k, default)) for k in somekeys ])
```

If you want to remove from the original the items you're extracting:

```
def sub_dict_remove(somedict, somekeys, default=None):
   return dict([ (k, somedict.pop(k, default)) for k in somekeys ])
```

Alternatively, you might prefer missing keys to be simply omitted from the result. This, too, requires just minor modifications:

```
def sub_dict_select(somedict, somekeys):
   return dict([ (k, somedict[k]) for k in somekeys if k in somedict])
def sub_dict_remove_select(somedict, somekeys):
   return dict([ (k, somedict.pop(k)) for k in somekeys if k in somedict])
```

In Python 2.4, you can use generator expressions, instead of list comprehensions, as the arguments to **dict** in each of the functions shown in this recipe. Just change the syntax of the calls to **dict**, from **dict([. . .])** to **dict(. . .)** (removing the brackets adjacent to the parentheses) and enjoy the resulting slight simplification and acceleration.

An existing **dict** maps keys to unique values, and you want to build the inverse **dict**, mapping each value to its key.

You can write a function that passes a list comprehension as dict's argument to build the new requested dictionary:

```
def invert_dict(d):
   return dict([ (v, k) for k, v in d.iteritems() ])
```

For large dictionaries, though, it's faster to use the generator izip from the itertools module in the Python Standard Library:

```
from itertools import izip
def invert_dict_fast(d):
   return dict(izip(d.itervalues(), d.iterkeys()))
```

When we program in Python, we normally "disregard minor optimizations," as Donald Knuth suggested over thirty years ago: we place a premium on clarity and correctness and care relatively little about speed. However, it can't hurt to know about faster possibilities: when we decide to code in a certain way because it's simpler or clearer than another, it's best if we are taking the decision deliberately, not out of ignorance.

function **invert_dict_fast**, also in this recipe's Solution, isn't really any more complicated: it just operates more abstractly, by getting all keys and all values as two separate iterators and zipping them up (into an iterator whose items are the needed, swapped **(value, key)** pairs) via a call to generator **izip**, supplied by the **itertools** module of the Python Standard Library. If you get used to such higher abstraction levels, they will soon come to feel simpler than lower-level code!

Performance gain is an extra incentive for getting familiar with working at higher abstraction levels, a familiarity that has conceptual and productivity pluses, too.

By nature, a dictionary is a one-to-one mapping, but it's not hard to make it one-to-manyin other words, to make one key map to multiple values.

```
d1 = {}
d.setdefault(key, []).append(value)

d3 = {}
d3.setdefault(key, set()).add(value)
```

Given two dictionaries, you need to find the set of keys that are in both dictionaries (the intersection) or the set of keys that are in either dictionary (the

union).

The fastest way to compute the **dict** that is the set-union is:

```
union = dict(a, **b)
```

The fastest concise way to compute the **dict** that is the set-intersection is:

```
inter = dict.fromkeys([x for x in a if x in b])
```

The clearest and simplest way to print all names for which you know both address and phone number, and their associated data, is:

```
for name in set(phones) & set(addresses):
    print name, phones[name], addresses[name]

for name in set(phones).intersection(addresses):
    print name, phones[name], addresses[name]
```

If you use the named **intersection** method, rather than the **&** intersection operator, you don't need to turn both **dict**s into **set**s: pass the other **dict** as the argument to the **intersection** method.

In Python, assignment is a statement, not an expression. Thus, you cannot assign the result that you are also testing, for example, in the condition of an **if**, **elif**, or **while** statement. This is usually fine: just structure your code to avoid the need to assign while testing (in fact, your code will often become clearer as a result).

It's easy to code a **printf** function in Python:

```
def printf(format, *args):
    print format % args
```

You want to pick an item at random from a list, just about as **random.choice** does, but you need to pick the various items with different probabilities given in another list, rather than picking any item with equal probability as **random.choice** does.

```
import random
def random_pick(some_list, probabilities):
    x = random.uniform(0, 1)
    cumulative_probability = 0.0
    for item, item_probability in zip(some_list, probabilities):
        cumulative_probability += item_probability
```

```
            if x < cumulative_probability: break
        return item
```

Module **random** in the standard Python library does not have the weighted choice functionality that is sometimes needed in games, simulations, and random tests, so I wrote this recipe to supply this functionality. The recipe uses module **random**'s function **uniform** to get a uniformly distributed pseudo-random number between 0.0 and 1.0, then loops in parallel on items and their probabilities, computing the increasing cumulative probability, until the latter becomes greater than the pseudo-random number. **probabilities** is a sequence with just as many items as **some_list**, which are probabilitiesthat is, numbers between 0.0 and 1.0, summing up to 1.0; if these assumptions are violated, you may still get some random picks, but they will not follow the (inconsistent) specifications encoded in the function's arguments.

Searching and Sorting

Professor Knuth's masterful work on the topics of sorting and searching spans nearly 800 pages of sophisticated technical text. In Python practice, we reduce it to two imperatives (we read Knuth so you don't have to):

1. When you need to sort, find a way to use the built-in **sort** method of Python lists.
2. When you need to search, find a way to use built-in dictionaries.

You want to sort a dictionary. This probably means that you want to sort the keys and then get the values in that same sorted order.

```
def sortedDictValues(adict):
    keys = adict.keys()
    keys.sort()
    return [adict[key] for key in keys]
```

You want to sort a list of strings, ignoring case differences. The decorate-sort-undecorate (DSU) idiom is simple and fast:

```
def case_insensitive_sort(string_list):
    auxiliary_list = [(x.lower(), x) for x in string_list]  # decorate
    auxiliary_list.sort()                                   # sort
    return [x[1] for x in auxiliary_list]                   # undecorate
```

In Python 2.4, DSU is natively supported; you can use the following even shorter and faster approach:

```
def case_insensitive_sort(string_list):
    return sorted(string_list, key=str.lower)
```

The DSU idiom builds an auxiliary list, whose items are tuples where each item of the original list is preceded by a "key". The sort then takes place on the key, because Python compares tuples lexicographically (i.e., it compares the tuples' first items first). With DSU, the **lower** method gets called only **n** times to sort a list of **n** strings, which saves enough time to cover the small costs of the first, decorate step and the final, undecorate step, with a big net increase in speed.

DSU is also sometimes known, not quite correctly, as the Schwartzian Transform, by somewhat imprecise analogy with a well-known idiom of the Perl language.

DSU is so important that Python 2.4 supports it directly: you can optionally pass to the **sort** method of a list an argument named **key**, which is the callable to

use on each item of the list to obtain the key for the sort. If you pass such an argument, the sorting internally uses DSU. So, in Python 2.4, **string_list.sort(key=str.lower)** is essentially equivalent to function case_insensitive_sort, except the **sort** method sorts the list in-place (and returns **None**) instead of returning a sorted copy and leaving the original list alone.

You want to maintain a sequence, to which items are added, in a sorted state, so that at any time, you can easily examine or remove the smallest item currently present in the sequence. You can use the heapq module of the Python Standard Library:

```
import heapq
heapq.heapify(the_list)
```

Now the list is not necessarily fully sorted, but it does satisfy the heap property (meaning if all indices involved are valid, the_list[i]<=the_list[2*i+1] and the_list[i]<=the_list[2*i+2])so, in particular, the_list[0] is the smallest item. To keep the heap property valid, use result=heapq.heappop(the_list) to get and remove the smallest item and heapq.heappush(the_list, newitem) to add a new item.

When you need to retrieve data in an ordered way (at each retrieval getting the smallest item among those you currently have at hand), you can pay the runtime cost for the sorting when you retrieve the data, or you can pay for it when you add the data. One approach is to collect your data into a list and sort the list. Now it's easy to get your data in order, smallest to largest.

An alternative approach is to use a data organization known as a heap, a type of binary tree implemented compactly, yet ensuring that each "parent" is always less than its "children". The best way to maintain a heap in Python is to use a **list** and have it managed by the **heapq** library module. The list does not get fully sorted, yet you can be sure that, whenever you **heappop** an item from the list, you always get the lowest item currently present, and all others will be adjusted to ensure the heap property is still valid. Each addition with **heappush**, and each removal with **heappop**, takes a short time proportional to the logarithm of the current length of the list (**O(log N)**).

A good occasion to use this heap approach is when you have a long-running queue with new data periodically arriving, and you always want to be able to get the most important item off the queue without having to constantly re-sort your data or perform full searches. This concept is known as a **priority queue**, and a heap is an excellent way to implement it. The **heapq** module supplies you with the smallest item at each **heappop**, so make sure to arrange the way you encode your items' priority values to reflect this.

You need to get just a few of the smallest items from a sequence. You could sort the sequence and just use **seq[:n]**, but is there any way you can do better?

sort is very fast, but it still takes O(n log n) time, while we can get the first n smallest elements in time O(n) if n is small. Here is a simple and practical generator for this purpose, which works equally well in Python 2.3 and 2.4:

```
import heapq
def isorted(data):
    data = list(data)
    heapq.heapify(data)
    while data:
        yield heapq.heappop(data)
```

In Python 2.4 only, you can use an even simpler and faster way to get the smallest n items of data when you know n in advance:

```
import heapq
def smallest(n, data):
    return heapq.nsmallest(n, data)
```

In Python 2.4, module **heapq** has also grown two new functions. **heapq.nlargest(n, data)** returns a list of the **n** largest items of data; **heapq.nsmallest(n, data)** returns a list of the **n** smallest items. These functions do not require that **data** satisfy the heap condition; indeed, they do not even require **data** to be a list. Any bounded iterable whose items are comparable will do.

Lesson to retain: **whenever you optimize, measure**. You shouldn't choose optimizations based on first principles, since the performance numbers can vary so widely, even between vastly compatible "point releases". A secondary point can be made: if you care about performance, move to Python 2.4 as soon as you can.

Looking for an item **x** in a list **L** is very easy in Python: to check whether the item is there at all, **if x in L**; to find out where exactly it is, **L.index(x)**. However, if **L** has length **n**, these operations take time proportional to **n**essentially, they just loop over the list's items, checking each for equality to **x**. If **L** is sorted, we can do better.

The classic algorithm to look for an item in a sorted sequence is known as binary search, because at each step it roughly halves the range it's still searching onit generally takes about $\log_2 n$ steps. It's worth considering when you're going to look for items many times, so you can amortize the cost of sorting over many searches. Once you've decided to use binary search for **x** in **L**, after calling **L.sort()**, module **bisect** from the Python Standard Library makes the job easy.

```
import bisect
x_insert_point = bisect.bisect_right(L, x)
x_is_present = L[x_insert_point-1:x_insert_point] == [x]
```

We need function **bisect.bisect_right**, which returns the index where an item should be inserted, to keep the sorted list sorted, but doesn't alter the list; moreover, if the item already appears in the list, **bisect_right** returns an index that's just to the right of any items with the same value. So, after getting this "insert point" by calling **bisect.bisect_right(L, x)**, we need only to check the list immediately before the insert point, to see if an item equal to **x** is already there.

You need to get from a sequence the **n**th item in rank order (e.g., the middle item, known as the median). If the sequence was sorted, you would just use **seq[n]**. But the sequence isn't sorted, and you wonder if you can do better than just sorting it first. If the sequence is big, has been shuffled enough, and comparisons between its items are costly. Sort is very fast, but in the end (when applied to a thoroughly shuffled sequence of length **n**) it always takes **O(n log n)** time, while there exist algorithms that can be used to get the **n**th smallest element in time **O(n)**. Here is a function with a solid implementation of such an algorithm:

```
import random
def select(data, n):
    " Find the nth rank ordered element (the least value has rank 0). "

    # make a new list, deal with < 0 indices, check for valid index
    data = list(data)
    if n < 0:
        n += len(data)
    if not 0 <= n < len(data):
        raise ValueError, "can't get rank %d out of %d" % (n, len(data))

    # main loop, quicksort-like but with no need for recursion
    while True:
        pivot = random.choice(data)
        pcount = 0
        under, over = [], []
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
```

```
        numunder = len(under)
        if n < numunder:
            data = under
        elif n < numunder + pcount:
            return pivot
        else:
            data = over
            n -= numunder + pcount
```

The algorithm then loops, implementing at each leg a few key ideas: randomly choosing a pivot element; slicing up the list into two parts, made up of the items that are "under" and "over" the pivot respectively; continuing work for the next leg on just one of the two parts, since we can tell which one of them the **n**th element will be in, and the other part can safely be ignored. The ideas are very close to that in the classic algorithm known as quicksort.

The random choice of pivot makes the algorithm robust against unfavorable data orderings; this implementation decision costs about **log$_2$N** calls to **random.choice**. Another implementation issue worth pointing out is that the recipe counts the number of occurrences of the pivot: this precaution ensures good performance even in the anomalous case where **data** contains a high number of repetitions of identical values.

Extracting the bound methods **.append** of lists **under** and **over** as local variables **uappend** and **oappend** may look like a pointless, if tiny, complication, but it is, in fact, a very important optimization technique in Python. To keep the compiler simple, straightforward, unsurprising, and robust, Python does not hoist constant computations out of loops, nor does it "cache" the results of method lookup. If you call **under.append** and **over.append** in the inner loop, you pay the cost of lookup each and every time. If you want something hoisted, hoist it yourself. When you're considering an optimization, you should always measure the code's performance with and without that optimization, to check that the optimization does indeed make an important difference.

The break-even point will be smaller if the items in the sequence have costly comparison methods, since the key difference between the two approaches is in the number of comparisons performed **select** takes **O(n)**, **selsor** takes **O(n log n)**. Although **select** has more general overhead, when compared to the wondrously efficient coding of lists' **sort** method, nevertheless, if **n** is large enough and each comparison is costly enough, **select** is still well worth considering.

Functional programming languages, of which **Haskell** is a great example, are splendid animals, but Python can hold its own in such company:

```
   def qsort(L):
      if len(L) <= 1: return L
```

```
    return qsort([lt for lt in L[1:] if lt < L[0]]) + L[0:1] + \
          qsort([ge for ge in L[1:] if ge >= L[0]])

def q(x):
    if len(x) > 1:
        lt = [i for i in x if cmp(i,x[0]) == -1 ]
        eq = [i for i in x if cmp(i,x[0]) == 0 ]
        gt = [i for i in x if cmp(i,x[0]) == 1 ]
        return q(lt) + eq + q(gt)
    else:
        return x
```

This rather naive implementation of quicksort illustrates the expressive power of **list comprehensions**. Do not use this approach in real code! Python lists have an in-place **sort** method that is much faster and should always be preferred; in Python 2.4, the new built-in function **sorted** accepts any finite sequence and returns a new sorted list with the sequence's items. The only proper use of this recipe is for impressing friends, particularly ones who (quite understandably) are enthusiastic about **functional programming**, and particularly about the **Haskell** language.

Both implementations pivot on the first element of the list and thus have worst-case **O(n)** performance for the very common case of sorting an already sorted list. You would never want to do so in production code!

You can write a less compact version with similar architecture in order to use named local variables and functions for enhanced clarity:

```
def qsort(L):
    if not L: return L
    pivot = L[0]
    def lt(x): return x < pivot
    def ge(x): return x >= pivot
    return qsort(filter(lt, L[1:])) + [pivot] + qsort(filter(ge, L[1:]))
```

You can easily move to a slightly less naive version, using random pivot selection to make worst-case performance less likely and counting pivots to handle degenerate case with many equal elements:

```
import random
def qsort(L):
    if not L: return L
    pivot = random.choice(L)
    def lt(x): return x < pivot
    def gt(x): return x > pivot
    return qsort(filter(lt, L))+[pivot]* L.count(pivot)+qsort(filter(gt, L))
```

Despite the enhancements, they are meant essentially for fun and demonstration purposes. Production-quality sorting code is quite another thing:

these little jewels, no matter how much we dwell on them, will never match the performance and solidity of Python's own built-in sorting approaches.

Fortunately, in the real world, Pythonists are much too sensible to write convoluted, **lambda**-filled horrors such as this. In fact, many of us feel enough aversion to **lambda** itself that we go out of our way to use readable **def** statements instead.

You need to perform frequent tests for membership in a sequence. The **O(n)** behavior of repeated **in** operators hurts performance, but you can't switch to using just a dictionary or set instead of the sequence, because you also need to keep the sequence's order.

Say you need to append items to a list only if they're not already in the list. One sound approach to this task is the following function:

```
def addUnique(baseList, otherList):
    auxDict = dict.fromkeys(baseList)
    for item in otherList:
        if item not in auxDict:
            baseList.append(item)
            auxDict[item] = None
```

The **in**-operator inside the loop checks for membership in a **dict**a step that makes all the difference because checking for membership in a **dict** takes roughly constant time, independent of the number of items in the **dict**! So, the **for** loop takes time proportional to **len(otherList)**, and the entire function takes time proportional to the sum of the lengths of the two lists.

An important requisite for any of these membership-test optimizations is that the values in the sequence must be hashable.

You need to find occurrences of a subsequence in a larger sequence. If the sequences are strings (plain or Unicode), Python strings' **find** method and the standard library's **re** module are the best approach. Otherwise, use the Knuth-Morris-Pratt algorithm (KMP):

```
def KnuthMorrisPratt(text, pattern):
    ''' Yields all starting positions of copies of subsequence 'pattern'
        in sequence 'text' -- each argument can be any iterable.
        At the time of each yield, 'text' has been read exactly up to and
        including the match with 'pattern' that is causing the yield. '''
    # ensure we can index into pattern, and also make a copy to protect
    # against changes to 'pattern' while we're suspended by `yield'
    pattern = list(pattern)
    length = len(pattern)
```

```
        # build the KMP "table of shift amounts" and name it 'shifts'
        shifts = [1] * (length + 1)
        shift = 1
        for pos, pat in enumerate(pattern):
            while shift <= pos and pat != pattern[pos-shift]:
                shift += shifts[pos-shift]
            shifts[pos+1] = shift
        # perform the actual search
        startPos = 0
        matchLen = 0
        for c in text:
            while matchLen == length or matchLen >= 0 and pattern[matchLen] != c:
                startPos += shifts[matchLen]
                matchLen -= shifts[matchLen]
            matchLen += 1
            if matchLen == length: yield startPos
```

Since KMP accesses the text sequentially, it is natural to implement it in a way that allows the text to be an arbitrary iterator. After a preprocessing stage that builds a table of shift amounts and takes time that's directly proportional to the length of the pattern, each text symbol is processed in constant amortized time.

Object-Oriented Programming

Object-oriented programming (OOP) is among Python's greatest strengths. Python's OOP features continue to improve steadily and gradually, just like Python in general. I am constantly amazed at the systematic progress Python achieves without sacrificing solidity, stability, and backwards-compatibility.

If you can meet your goals with simplicity (and most often, in Python, you can), then keep your code simple. Simplicity pays off in readability, maintainability, and, more often than not, performance, too. To describe something as clever is not considered a compliment in the Python culture.

Python supports multiple paradigms. While everything in Python is an object, you package things as OOP objects only when you want to. Other languages try to force your programming style into a predefined mold for your own good, while Python empowers you to make and express your own design choices.

You define methods with the same **def** statement Python uses to define functions, exactly because methods are essentially functions. However, a method is an attribute of a class object, and its first formal argument is (by universal convention) named **self**. **self** always refers to the instance on which you call the method.

class object is a **first-class** object, like every Python object, meaning that you can put it in lists and dictionaries, pass it as an argument to functions, and so on.

The method with the special name **__init__** is also known as the **constructor** (or more properly the **initializer**) for instances of the class. Python calls this special method to initialize each newly created instance with the arguments that you passed when calling the class (except for **self**, which you do not pass explicitly since Python supplies it automatically). The body of **__init__** typically binds attributes on the newly created **self** instance to appropriately initialize the instance's state.

In addition to the constructor (**__init__**), your class may have other special methods, meaning methods with names that start and end with two underscores. Python calls the special methods of a class when instances of the class are used in various operations and built-in functions. For example, **len(x)** returns **x.__len__()**; **a+b** normally returns **a.__add__(b)**; **a[b]** returns **a.__getitem__(b)**. Therefore, by defining special methods in a class, you can make instances of that class interchangeable with objects of built-in types, such as numbers, lists, and dictionaries.

The ability to handle different objects in similar ways, known as **polymorphism**, is a major advantage of OOP. Thanks to polymorphism, you can call the same method on various objects, and each object can implement the

method appropriately.

Other languages require inheritance, or the formal definition and implementation of interfaces, in order to enable such polymorphism. In Python, all you need is to have methods with the same **signature**. This **signature-based polymorphism** allows a style of programming that's quite similar to **generic programming** (e.g., as supported by C++'s **template** classes and functions), without syntax cruft and without conceptual complications.

Python also uses inheritance, which is mostly a handy, elegant, structured way to reuse code.

The method of a subclass often overrides a method from the superclass, but also needs to call the method of the superclass as part of its own operation.

An overriding method often is best implemented by delegating some of the work to the same method in the superclass. In Python, the syntax **Classname.method(self**, . . .**)** delegates to **Classname**'s version of the method. A vastly preferable way to perform superclass delegation, however, is to use Python's built-in **super**.

```
class Behave(object):
    def __init__(self, name):
        self.name = name
    def repeat(self, N): print N * "*-*"

class OneMore(Behave):
    #def repeat(self, N): Behave.repeat(self, N + 1)
    def repeat(self, N): super(OneMore, self).repeat(N + 1)
```

This **super** construct is equivalent to the explicit use of **Behave.repeat** in this simple case, but it also allows class **OneMore** to be used smoothly with **multiple inheritance**.

Python does fully support multiple inheritance: one class can inherit from several other classes. In terms of coding, this feature is sometimes just a minor one that lets you use the mix-in class idiom, a convenient way to supply functionality across a broad range of classes.

Multiple inheritance is particularly important because of its implications for object-oriented analysis the way you conceptualize your problem and your solution in the first place. Single inheritance pushes you to frame your problem space via taxonomy (i.e., mutually exclusive classification). The real world doesn't work like that.

Modeling aspects of the real world in your programs is hard enough without buying into artificial constraints such as taxonomy. Multiple inheritance frees you from these constraints.

Now that you've seen Python's notation for inheritance, you realize that

writing **class X(object)** means that class **X** inherits from class **object**. If you just write **class Y**:, you're saying that **Y** doesn't inherit from anything**Y**, so to speak, "stands on its own". For backwards compatibility, Python allows you to request such a rootless class, and, if you do, then Python makes class **Y** an "old-style" class, also known as a **classic** class, meaning a class that works just like all classes used to work in the Python versions of old. Python is very keen on backwards-compatibility.

For many elementary uses, you won't notice the difference between classic classes and the new-style classes that are recommended for all new Python code you write. However, it's important to underscore that classic classes are a legacy feature, not recommended for new code. You cannot use **super** within classic classes, and in practice, you should not do any serious use of multiple inheritance with them. Many important features of today's Python OOP, such as the **property** built-in, can't work completely.

In practice, even if you're maintaining a large body of legacy Python code, the next time you need to do any substantial maintenance on that code, you should take the little effort required to ensure all classes are new style: it's a small job, and it will ease your future maintenance burden quite a bit. Instead of explicitly having all your classes inherit from object, an equivalent alternative is to add the following assignment statement close to the start of every module that defines any classes:

```
__metaclass__ = type
```

The built-in type is the metaclass of object and of every other new-style class and built-in type. That's why inheriting from object or any built-in type makes a class new style: the class you're coding gets the same metaclass as its base. A class without bases can get its metaclass from the module-global __metaclass__ variable, which is why the "state"ment I suggest suffices to ensure that any classes without explicit bases are made new-style.

You need to define module-level variables (i.e., named constants) that client code cannot accidentally rebind.

You can install any object as if it were a module. Save the following code as module **const.py** on some directory on your Python sys.path:

```
import sys
class _const(object):
    class ConstError(TypeError): pass
    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise self.ConstError, "Can't rebind const(%s)" % name
        self.__dict__[name] = value
    def __delattr__(self, name):
```

```
        if name in self.__dict__:
            raise self.ConstError, "Can't unbind const(%s)" % name
        raise NameError, name
sys.modules[__name__] = _const()
```

Now, any client code can import **const**, then bind an attribute on the **const** module just once, as follows:

```
const.magic = 23
```

Once the attribute is bound, the program cannot accidentally rebind or unbind it:

```
const.magic = 88     # raises const.ConstError
del const.magic      # raises const.ConstError
```

Numbers, strings, and tuples are immutable: if you bind a name in **const** to such an object, not only will the name always be bound to that object, but the object's contents also will always be the same since the object is immutable. However, other objects, such as lists and dictionaries, are mutable: if you bind a name in **const** to, say, a list object, the name will always remain bound to that list object, but the contents of the list may change.

Persistence and Databases

marshal has the advantage of being able to retrieve large data structures with blinding speed. The **pickle** and **cPickle** modules allow general storage of objects, including classes, class instances, and circular structures. **cPickle** is so named because it is implemented in C and is consequently quite fast, but it remains slower than **marshal**. For access to structured data in a somewhat human-readable form, it is also worth considering storing and retrieving data in XML format.

While **marshal** and **pickle** provide basic serialization and deserialization of structures, the application programmer will frequently desire more functionality, such as transaction support and concurrency control. When the relational model doesn't fit the application, a direct object database implementation such as the Z-Object Database (ZODB) might be appropriate.

You want to serialize and reconstruct a Python data structure whose items are fundamental Python objects (e.g., lists, tuples, numbers, and strings but no classes, instances, etc.) as fast as possible.

Algorithms

When you're trying to solve a problem that may not have been solved before, you may have some intuitions about how to proceed, but you rarely know in advance exactly what needs to be done. The only way to proceed is to try things, many things, everything you can think of, just to see what happens. Python makes such exploration easier by minimizing the time and pain from conception to code: if your colleagues are using, for example, C or Java, it's not unusual for you to try (and discard) six different approaches in Python while they're still getting the bugs out of their first attempt.

I've used many languages in my computer career, and I know of none more productive than Python for prototyping. Best of all, while being an expert is often helpful, moderate skill in Python is much easier to obtain than for many other languages, yet much more productive for research and prototyping than merely moderate skill in any other language I've used. You don't have to be an expert to start!

Python 2.3 introduced a new **timeit** module, which captures best practice and is perfectly suited to timing small programs with a minimum of fuss and pitfalls. Everyone should learn how to use **timeit**, and basic usage is very easy to learn.

timeit picks the best way of measuring time on your platform and runs your code in a loop. The module tries a few times first to determine how many iterations to use in the loop, aiming at a total loop time between 0.2 and 2 seconds. When it determines a suitable number of iterations for the loop, it then runs the loop three times, reports the shortest time, and computes the time per loop iteration. The iterations per loop, and number of loops to run, can be forced to specific values with command-line options.

Measuring itself will consume more of your time than you expect. As noted innumerable times by innumerable authors, the speed of most of your code doesn't matter at all. Find the 10% that consumes most of the time before worrying about any of it. When you find the true bottlenecks, **timeit** can help you measure the speed of alternatives objectively and you may be surprised by what you find.

You have a sequence that may include duplicates, and you want to remove the duplicates in the fastest possible way, without knowing much about the properties of the items in the sequence. You do not care about the "or"der of items in the resulting sequence.

The key is to try several approaches, fastest first, and use **try**/**except** to handle the failing cases of the faster approaches by falling back to slower approaches. Here's a function that implements exactly this strategy:

```
    # support 2.3 as well as 2.4
    try: set
    except NameError: from sets import Set as set

    def unique(s):
        """ Return a list of the elements in s in arbitrary order, but without
            duplicates. """

        # Try using a set first, because it's the fastest and will usually work
        try:
            return list(set(s))
        except TypeError:
            pass  # Move on to the next method

        # Since you can't hash all elements, try sorting, to bring equal items
        # together and then weed them out in a single pass
        t = list(s)
        try:
            t.sort()
        except TypeError:
            del t  # Move on to the next method
        else:
            # the sort worked, so we're fine -- do the weeding
            return [x for i, x in enumerate(t) if not i or x != t[i-1]]

        # Brute force is all that's left
        u = []
        for x in s:
            if x not in u:
                u.append(x)
        return u
```

unique function is to take a sequence **s** as an argument and return a list of the items in **s** in arbitrary order, but without duplicates.

The fastest way to remove duplicates from a sequence depends on fairly subtle properties of the sequence elements, such as whether they're hashable and whether they support full comparisons. The **unique** function shown in this recipe tries three methods, from fastest to slowest, letting runtime exceptions pick the best method for the sequence at hand.

For fastest speed, all sequence elements must be hashable. When they are, the **unique** function will usually work in linear time.

If it turns out that hashing the elements (e.g., using them as dictionary keys, or, as in this case, **set** elements) is not possible, the next best situation is when the elements enjoy a total ordering, meaning that each element can be

compared to each other element with the **<** operator. If **list(s).sort()** doesn't raise a **TypeError**, we can assume that **s**' elements can be sorted and therefore enjoy a total ordering. Then **unique** will usually work in **O(n log(n))** time. Python lists' **sort** method is particularly efficient in the presence of partially ordered data.

If sorting also turns out to be impossible, the sequence items must at least support equality testing, or else the very concept of duplicates can't really be meaningful for them. In this case, **unique** works in quadratic timethat is, **O(n^2)**.

You could split this recipe's function into three distinct functions and directly call the one that best meets your needs. In practice, however, the brute-force method is so slow for large sequences that nothing measurable is lost by simply letting the function as written try the faster methods first.

You have a sequence that may include duplicates, and you want to remove the duplicates in the fastest possible way. Moreover, the output sequence must respect the item ordering of the input sequence.

This requirement often arises in conjunction with a function **f** that defines an equivalence relation among items: **x** is equivalent to **y** if and only if **f(x)==f(y)**.

```
# f defines an equivalence relation among items of sequence seq, and
# f(x) must be hashable for each item x of seq
def uniquer(seq, f=None):
    """ Keeps earliest occurring item of each f-defined equivalence class """

    if f is None:    # f's default is the identity function f(x) -> x
        def f(x): return x
    already_seen = set()
    result = []
    for item in seq:
        marker = f(item)
        if marker not in already_seen:
            already_seen.add(marker)
            result.append(item)
    return result
```

If the items are hashable, it's not hard to maintain sequence order, keeping only the first occurrence of each value. More generally, we may want uniqueness within equivalence classes, as shown in this recipe's Solution: the **uniquer** function accepts as an argument a function **f** that must return hashable objects, such that **f(x)==f(y)** if and only if items **x** and **y** are equivalent.

If you need to keep the last occurrence, rather than the earliest occurrence, of an item in each equivalence class, the simplest approach is to **reverse** the input sequence (or, rather, a copy thereof into a local list, since the input might be immutable or at any rate not support reversing), then, after processing with

uniquer, **reverse** the resulting list:

```
def uniquer_last(seq, f=None):
    seq = list(seq)
    seq.reverse()
    result = uniquer(seq, f)
    result.reverse()
    return result
```

If you must deal with nonhashable items, the simplest fallback approach is to use a set-like container that supports the **add** method and membership testing without requiring items to be hashable. Unfortunately, performance will be much worse than with a real **set**.

```
def uniquer_with_simplest_fallback(seq, f=None):
    if f is None:
        def f(x): return x
    already_seen = set()
    result = []
    for item in seq:
        marker = f(item)
        try:
            new_marker = marker not in already_seen
        except TypeError:
            class TotallyFakeSet(list):
                add = list.append
            already_seen = TotallyFakeSet(already_seen)
            new_marker = marker not in already_seen
         if new_marker:
             already_seen.add(marker)
             result.append(item)
    return result
```

However, remember that you can often use an **f** that gives you hashable markers for nonhashable items. The built-in function **repr** can often be useful for this purpose.

```
lol = [ [1, 2], [], [1, 2], [3], [], [3, 4], [1, 2], [], [2, 1] ]
print uniquer(lol, repr)
```

While the items of **lol** are lists, and thus are not hashable, the built-in function **repr** produces representations of each of the items as a string, which is hashable. This enables use of the fast function **uniquer**. Unfortunately, **repr** is not useful for nonhashable items of other types, including **dict** and **set**. Because of the workings of hash-collision resolution, it's quite possible to have **d1 == d2**

and yet **repr(d1) != repr(d2)** for two dictionaries **d1** and **d2**, depending on the exact sequences of adds that built each **dict**. Whether **repr** can help for instances of a certain user-defined type depends on how accurately and usefully that specific type defines special method **__repr__**, which **repr** calls.

Out of all the general programming techniques presented in the various functions of this recipe, the idea of writing higher-order functions, which organize a computation and appropriately call back to functions that they receive as arguments, is easily the most precious and widely applicable concept. This idea is well worth keeping in mind in several circumstances not just for old Haskell-heads, because it works just as well in Python.

You need to generate random samples with replacement out of a "population" of items that are held in a sequence.

A generator for the purpose is quintessentially simple:

```
import random
def sample_wr(population, _choose=random.choice):
   while True: yield _choose(population)
```

The **sample_wr** generator shown in this recipe is unbounded: used on its own, it will keep looping forever. To bound the output of an intrinsically unbounded generator, you can use it in a **for** statement that at some point executes a **break.**

```
from operator import itemgetter
from heapq import nlargest

class bag(object):
   def __init__(self, iterable=()):
      # start empty, then add the `iterable' if any
      self._data = {}
      self.update(iterable)

   def update(self, iterable):
      # update from an element->count mapping, or from any iterable
      if isinstance(iterable, dict):
         for elem, n in iterable.iteritems():
            self[elem] += n
      else:
         for elem in iterable:
            self[elem] += 1

   def __contains__(self, elem):
```

```
        # delegate membership test
        return elem in self._data

    def __getitem__(self, elem):
        # default all missing items to a count of 0
        return self._data.get(elem, 0)

    def __setitem__(self, elem, n):
        # setting an item to a count of 0 removes the item
        self._data[elem] = n
        if n == 0:
            del self._data[elem]

    def __delitem__(self, elem):
        # delegate to __setitem__ to allow deleting missing items
        self[elem] = 0

    def __len__(self):
        # length is computed on-the-fly
        return sum(self._data.itervalues())

    def __nonzero__(self):
        # avoid truth tests using __len__, as it's relatively slow
        return bool(self._data)

    def __eq__(self, other):
        # a bag can only equal another bag
        if not isinstance(other, bag):
            return False
        return self._data == other._data

    def __ne__(self, other):
        # a bag always differs from any non-bag
        return not (self == other)

    def __hash__(self):
        # a bag can't be a dict key nor an element in a set
        raise TypeError

    def __repr__(self):
        # typical string-representation
        return '%s(%r)' % (self.__class__.__name__, self._data)

    def copy(self):
```

```
            # make and return a shallow copy
            return self.__class__(self._data)
        __copy__ = copy  # For the copy module

        def clear(self):
            # remove all items
            self._data.clear()

        def __iter__(self):
            # yield each element the # of times given by its count
            for elem, cnt in self._data.iteritems():
                for i in xrange(cnt):
                    yield elem

        def iterunique(self):
            # yield each element only once
            return self._data.iterkeys()

        def itercounts(self):
            # yield element-count pairs
            return self._data.iteritems()

        def mostcommon(self, n=None):
            # return the n (default: all) most common elements, each as an
            # element-count pair, as a list sorted by descending counts
            if n is None:
                return sorted(self.itercounts(), key=itemgetter(1), reverse=True)
            it = enumerate(self.itercounts())
            nl = nlargest(n, ((cnt, i, elem) for (i, (elem, cnt)) in it))
            return [(elem, cnt) for cnt, i, elem in nl]

    b = bag('banana')
    print b['b']
```

The **bag**, (i.e., multiset), is widely useful, since counting the numbers of occurrences of different objects is a frequent task useful in many applications.

Iterators and Generators

After namespaces, iterators and generators emerged as the next "honking great ideas" in Python. Since their introduction in Python 2.2, they have come to pervade and unify the language. They encourage a loosely coupled programming style that is simple to write, easy to read, flexible, and extendable.

Simply put, the iterator protocol has two halves, a producer and a consumer. An iterable object says, "I know how to supply data one element at a time," and the consumer says "please give me data one element at a time and say Stop when you're done."

The producer/consumer connection can appear in a number of guises. The simplest is where a function or constructor wraps around an iterable object. **sorted(set('simsalabim'))** has the set constructor looping over the elements of the iterable string and a **sorted** function wrapping around the resulting iterable set object.

In addition to functions and constructors, regular Python statements can use the **in** operator to loop over iterable objects. **for line in myfile: print line** loops over lines of an iterable **file** object. Likewise, **if token in sequence** loops over elements of a sequence until it finds a match until it reaches the end with no matches.

Both guises of the consumer side of the iterator protocol use the protocol implicitly. In addition, an explicit form is more flexible but used less often. The iterator object is saved as a variable, **it = iter(mystring)**. Then, the iterator's **next** method is called to retrieve a data element, **elem = it.next()**. Such calls are usually wrapped in **try**/**except** statements to catch the **StopIteration** exception that the iterator raises when the data stream is exhausted.

All of these approaches provide the full range of iterator benefits, including loose coupling and memory friendliness. The independently created and maintained **sorted** function, **set** data type, and **string** objects were readily combined. The memory friendliness derives from the one-at-a-time structure of the iterator protocol. Programs using iterators are likely to be less resource intensive and more scalable than their **list**-based counterparts.

Any iterator object must implement a **next** method and an **__iter__** method. The **next** method should raise **StopIteration** when the iteration is complete. To be useful, most iterators have a stored state that enables them to return a new data element on each call.

Instead of writing classes, two alternate approaches dominate. Starting with the observation that many functions and types both accept iterable inputs

and return iterable outputs, an obvious approach is to link them together in a "pipes and filters" style to create new tools. **def uniq(seq): return sorted(set(seq))** is a way to create a new tool directly from existing functions and types. Like functional programming, the resulting code is terse, readable, trivial to debug, and often runs at the speed of compiled C code. The economy of this approach motivated the creation of an entire module of iterator building blocks, the **itertools** module.

If no combination of building blocks solves the problem, the next best approach is to write a generator. By introducing a **yield** keyword, the responsibilities of creating an iterator are handled automatically. The iterator objects obtained by calling a generator are distinct, save their state, have an idempotent **__iter__** method, and have a **next** method that raises **StopIteration** when complete and stays stopped if called again afterwards. Python internally takes care of all of these details.

Starting with version 2.4, Python continued its evolution toward using iterators everywhere by introducing generator expressions (**genexps** for short). Genexps can be likened to a memory-efficient, scalable form of list comprehensions. Simply by replacing brackets with parentheses, an expression will yield one element at a time rather than filling memory all at once.

Paradoxically, the simpler and more general an idea, the more likely that people will find extraordinary and unexpected ways of using it. Here is a brief sampling of the ways that iterators and generators have been pushed to their outer limits.

Observing that the **yield** keyword has the unique capability of stopping execution, saving state, and later resuming, it is not surprising that techniques have been discovered for using generators to simulate co-routines and continuations. The core idea is to implement each routine as a generator and having a **dispatch** function launch the routines in orderly succession. Whenever a task switch is needed, the routines yield back to the dispatcher, which then launches or resumes the next routine by calling its **next** method. Small complications are involved for startup, termination, and data sharing, but they each are solvable without much ado and present fewer challenges than equivalent thread-based solutions.

Observing that some tools can be both producers and consumers, it is natural to want to stack them together like pipes and filters. While that analogy can lead to useful decoupling, be aware that underlying models are different. Iterators do not run independently from start to finish; instead, an outermost layer is always in control, requesting data elements one at a time, so that nothing runs until the outer layer starts making requests.

When stacking tools together (as in the first example with **sorted**, **set**, and a string), the code takes on the appearance of a functional programming

language. The resemblance is not shallow: iterators do fulfill some of the promise of lazy languages. So, it is natural to borrow some of the most successful techniques from those languages, such as Haskell and SML.

One such technique is to write innermost iterators to yield infinite streams and concentrate the control logic in an outermost driver function. For instance, in numerical programming, write a generator that yields successively better approximations to a desired result and call it from a function that stops whenever two successive approximations fall within a tolerance value. Separating the control logic from the calculation decouples the two, making them easier to write, test, and debug, and makes them more reusable in other contexts.

Just because iterators cannot be restarted doesn't mean they cannot be abandoned in mid-stream. The lazy, just-in-time style of production is a key feature of iterators. Take advantage of it. That's why the **for** statement supports a **break** keyword, after all.

You need an arithmetic progression, like the built-in **xrange** but with float values (**xrange** works only on integers).

```
import itertools
def frange(start, end=None, inc=1.0):
    "An xrange-like generator which yields float values"

    # imitate range/xrange strange assignment of argument meanings
    if end is None:
        end = start + 0.0     # Ensure a float value for 'end'
        start = 0.0

    assert inc              # sanity check

    for i in itertools.count():
        next = start + i * inc
        if (inc > 0.0 and next >= end) or (inc < 0.0 and next <= end):
            break
        yield next
```

Precision would suffer in a potentially dangerous way if we "simplified" the first statement in the loop body to something like:

```
    next += inc
```

Talking about speedbelieve it or not, looping with

```
for i in itertools.count()
```

is measurably faster than apparently obvious lower-level alternatives such

as:

```
    i = 0
    while True:
        ...loop body unchanged...
        yield next
        i += 1
```

Do consider using **itertools** any time you want speed, and you may be in for more of these pleasant surprises.

If you work with floating-point numbers, you should definitely take a look at **Numeric** and other third-party extension packages that make Python such a powerful language for floating-point computations.

When you know that iterable object **x** is bounded, building the list object you require is trivial:

```
y = list(x)
```

However, when you know that **x** is unbounded, or when you are not sure, then you must ensure termination before you call list. In particular, if you want to make a list with no more than **n** items from **x**, then standard library module itertools' function islice does exactly what you need:

```
import itertools
y = list(itertools.islice(x, N))
```

The powerful and generic concept of **iterable** is a great way to represent all sort of sequences, including unbounded ones, in ways that can potentially save you huge (and even infinite!) amounts of memory. With the standard library module **itertools**, generators you can code yourself, and, in Python 2.4, generator expressions, you can perform many manipulations on completely general iterables.

As long as you know for sure that the iterable is **bounded** (i.e., has a finite number of items), just call **list** with the iterable as the argument, as the "Solution" points out. In particular, avoid the goofiness of misusing a list comprehension such as **[i for i in x]**, when **list(x)** is faster, cleaner, and more readable!

The programming language Haskell, from which Python took many of the ideas underlying its list comprehensions and generator expression functionalities, has a built-in **take** function to cater to this rather frequent need, and **itertools.islice** is most often used as an equivalent to Haskell's built-in **take**.

In some cases, you cannot specify a maximum number of items, but you are able to specify a generic condition that you know will eventually be satisfied by

the items of iterable **x** and can terminate the proceedings. **itertools.takewhile** lets you deal with such cases in a very general way, since it accepts the controlling predicate as a callable argument. For example:

```
y = list(itertools.takewhile((11).__cmp__, x))
```

The parentheses are included to force the tokenization we mean, with **11** as an integer literal and the **period** indicating an access to its attribute.

For the special and frequent case in which the terminating condition is the equality of an item to some given value, a useful alternative is to use the two-arguments variant of the built-in function **iter**:

```
y = list(iter(iter(x).next, 11))
```

iter in its two-arguments form requires a callable as its first argument. The second argument is the **sentinel** value, meaning the value that terminates the iteration as soon as an item equal to it appears.

Generators are particularly suitable for implementing infinite sequences, given their intrinsically "lazy evaluation" semantics:

```
def fib():
    ''' Unbounded generator for Fibonacci numbers '''
    x, y = 0, 1
    while True:
        yield x
        x, y = y, x + y
if __name__ == "__main__":
    import itertools
    print list(itertools.islice(fib(), 10))
# outputs: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

It's worth reflecting on why a generator is so perfectly suitable for implementing an unbounded sequence and letting you work with it. Syntactically, a generator is "just" a function containing the keyword **yield**. When you call a generator, however, the function body does not yet execute. Rather, calling the generator gives you a special anonymous iterator object that wraps the function's body, the function's local variables, and the current point of execution, which is initially the start of the function.

When you call this anonymous iterator object's **next** method, the function body executes up to the next **yield** statement. **yield**'s argument is returned as the result of the iterator's **next** method, and the function is "frozen", with its execution state intact. When you call **next** again on the same iterator object, the function "thaws" and continues from where it left off, again up to the next **yield** statement.

If the function body "falls off the end", or executes a **return**, the iterator object raises **StopIteration** to indicate the end of the sequence. But, of course, if the sequence that the generator is producing is not bounded, the iterator never raises **StopIteration**. That's okay, as long as you don't rely on such an exception as the only way to terminate a loop.

The main point to retain is that it's all right to have infinite sequences represented by generators, since generators are computed lazily, as long as some control structure ensures that only a finite number of items are required from the generator. The answer to our curiosity as to why generators are so excellently suitable for this use is in the anonymous iterator object which a generator returns when we call it: that anonymous iterator wraps some code (the generator's function body) and some state (the function's local variables, and, crucially, the point at which the function's execution is to resume) in just the way that's most convenient for the computation of most sequences, be they bounded or unbounded.

You have an iterable **p** and need to get the **n** non-overlapping extended slices of stride **n**, which, if the iterable was a sequence supporting extended slicing, would be **p[0::n]**, **p[1::n]**, and so on up to **p[n-1::n]**.

```
def strider(p, n):
    """ Split an iterable p into a list of n sublists, repeatedly taking
        the next element of p and adding it to the next sublist.  Example:
        >>> strider('abcde', 3)
        [['a', 'd'], ['b', 'e'], ['c']]
        In other words, strider's result is equal to:
            [list(p[i::n]) for i in xrange(n)]
        if iterable p is a sequence supporting extended-slicing syntax.
    """

    # First, prepare the result, a list of n separate lists
    result = [ [] for x in xrange(n) ]

    # Loop over the input, appending each item to one of
    # result's lists, in "round robin" fashion
    for i, item in enumerate(p):
        result[i % n].append(item)
    return result
```

If we were willing to sacrifice generality, forcing argument **p** to be a sequence supporting extended slicing, rather than a generic iterable, we could use a very different approach, as the docstring of **strider** indicates:

```
def strider1(p, n):
    return [list(p[i::n]) for i in xrange(n)]
```

we might omit the **list** call to make each subsequence into a list, and/or code a generator to avoid consuming extra memory to materialize the whole list of results at once:

```
def strider2(p, n):
    for i in xrange(n):
        yield p[i::n]
```

or, in Python 2.4, with a generator expression:

```
def strider4(p, n):
    return (p[i::n] for i in xrange(n))
```

The best way to enhance the recipe is to recode it to avoid low-level fiddling with indices. While doing arithmetic on indices is conceptually quite simple, it can get messy and indeed is notoriously error prone. We can do better by a generous application of module **itertools** from the Python Standard Library:

```
import itertools
def strider5(p, n):
    result = [ [] for x in itertools.repeat(0, n) ]
    resiter = itertools.cycle(result)
    for item, sublist in itertools.izip(p, resiter):
        sublist.append(item)
    return result
```

Function **repeat** yields an object, repeatedly, a given number of times, and here we use it instead of the built-in function **xxrange** to control the list comprehension that builds the initial value for **result**. Function **cycle** takes an iterable object and returns an iterator that walks over that iterable object repeatedly and cyclically. Function **izip** is essentially like the built-in function **zip**, except that it returns an iterator and thus avoids the memory-consumption overhead that **zip** incurs by building its whole result list in memory at once.

You need to loop through every item of multiple iterables in parallel, meaning that you first want to get a tuple with all of the first items of each iterable, next, a tuple with all of the "second items", and so forth.

Say you have two iterables (lists, in this case) such as:

```
a = ['a1', 'a2', 'a3']
b = ['b1', 'b2']
```

If you want to loop "in parallel" over them, the most general and effective approach is:

```
import itertools
for x, y in itertools.izip(a, b):
    print x, y
```

This snippet outputs two lines:

```
a1 b1
a2 b2
```

The most general and effective way to loop "in parallel" over multiple iterables is to use function **izip** of standard library module **itertools**, as shown in the "Solution". The built-in function **zip is** an alternative that is almost as good:

```
for x, y in zip(a, b):
    print x, y
```

However, **zip** has one downside that can hurt your performance if you're dealing with long sequences: it builds the list of tuples in memory all at once (preparing and returning a list), while you need only one tuple at a time for pure looping purposes.

Both **zip** and **itertools.izip**, when you iterate in parallel over iterables of different lengths, stop as soon as the "shortest" such iterable is exhausted. This approach to termination is normally what you want.

In some cases, when iterating in parallel over iterables of different lengths, you may want shorter iterables to be conceptually "padded" with **None** up to the length of the longest iterable in the zipping. For this special need, you can use the built-in function **map** with a first argument of **None**:

```
for x, y in map(None, a, b):
    print x, y
```

You need to loop through every item of multiple iterables cross-productwise, meaning that you first want to get the first item of the first iterable paired with all the others, next, the second item of the first iterable paired with all the others, and so forth.

The simplest approach is often just a couple of nested **for** loops:

```
for x in a:
    for y in b:
        print x, y
```

However, in many cases, you'd rather get all items in the "cross-product" of multiple iterables as a single, linear sequence, suitable for using in a single **for** or for passing onwards to other sequence manipulation functions. For such needs, you may put the nested **for**s in a list comprehension:

```
for x, y in [(x,y) for x in a for y in b]: print x, y
```

A list comprehension lets you easily generate (as a single, linear sequence) all the pairings of several iterables (also known as the **cross-product**, **product set**, or **Cartesian product** of these iterables). However, the number of items in such a cross-product is the arithmetic product (multiplication) of the lengths of all the iterables involved, a number that may easily get quite large. A list comprehension, by definition, builds the entire list at once, which means that it may consume substantial amounts of memory. Also, you get to start iterating only when the whole cross-product list is entirely built.

Python 2.4 offers one obvious way to solve this problem: the newly introduced construct of **generator expressions**:

```
for x, y in ((x,y) for x in a for y in b): print x, y
```

A generator expression looks just like a list comprehension, except that it uses parentheses rather than brackets: it returns an iterator, suitable for looping on, rather than building and returning a list. Thus, a generator expression can save substantial amounts of memory, if you are iterating over a very long sequence. Also, you start executing the loop's body very soon, since each successive element gets generated iteratively, before each iteration of the loop's body. If your loop's body contains conditional **break**s, so that execution terminates as soon as some conditions are met, using a generator expression rather than a list comprehension can mean a potentially substantial improvement in performance.

If you need to support Python 2.3, and yet you want to achieve the kind of advantages that generator expressions can afford over list comprehensions, the best approach may be to code your own generator.

```
def cross_two(a, b):
    for x in a:
        for y in b:
            yield a, b
```

In many cases, getting all items in the "cross-product" as a single sequence is preferable, so it's worth knowing how to do that. However, do keep in mind that simplicity is an important virtue, and do not lose sight of it in pursuit of a cool (but complicated) solution. All the cool tools, constructs, and library modules that Python offers exist strictly to serve you, to let you build and maintain your applications with minimal effort. Don't go out of your way to use the new shiny tools if you can solve your application's problems with less effort in simpler ways!

You have a file that includes long logical lines split over two or more physical lines, with backslashes to indicate that a continuation line follows.

```
def logical_lines(physical_lines, joiner=''.join):
```

```
        logical_line = []
        for line in physical_lines:
            stripped = line.rstrip()
            if stripped.endswith('\\'):
                # a line which continues w/the next physical line
                logical_line.append(stripped[:-1])
            else:
                # a line which does not continue, end of logical line
                logical_line.append(line)
                yield joiner(logical_line)
                logical_line = []
        if logical_line:
            # end of sequence implies end of last logical line
            yield joiner(logical_line)

    text = ['some\\\n', 'lines\\\n', 'get\n', 'joined\\\n', 'up\n']
    for line in logical_lines(text, ' '.join):
        print line,

    some lines get
    joined up
```

Today, in Python, sequences are often processed most simply and effectively by means of generators. The generator can encompass just a small amount of generality without introducing extra complexity.

The need for "merging" sorted subsequences into a larger sorted sequence is reasonably frequent. If the amount of data is small enough to fit entirely in memory without problems, then the best approach is to build a list by concatenating all subsequences, then sort the list:

```
    def smallmerge(*subsequences):
        result = []
        for subseq in subsequences: result.extend(subseq)
        result.sort()
        return result
```

The **sort** method of list objects is based on a sophisticated natural merge algorithm, able to take advantage of existing sorted subsequences in the list you're sorting; therefore, this approach is quite fast, as well as simple. If you can choose this approach, it has many other advantages. For example, **smallmerge** works fine even if one of the **subsequences** isn't perfectly sorted to start with; and in Python 2.4, you may add a generic keywords argument ****kwds** to **smallmerge** and pass it right along to the **result.sort()** step, to achieve the flexibility afforded in that version by the **cmp=**, **key=**, and **reverse=** arguments to

list's **sort** method.

However, you sometimes deal with large sequences, which might not comfortably fit in memory all at the same time (e.g., your sequences might come from files on disk, or be computed on the fly, item by item, by other generators). Generator will enable you to perform your sequence merging while consuming a very moderate amount of extra memory.

The recipe's implementation uses a classic sequence-merging algorithm based on a priority queue, which, in turn, lets it take advantage of the useful **heapq** module in the Python Standard Library. **heapq** offers functions to implement a priority queue through the data structure known as a **heap**. As its first item, each pair has the "current item" in the corresponding subsequence and, as its second item, an iterator over that subsequence. At each iteration step, we yield the smallest "current item", then we advance the corresponding iterator and re-establish the "heap" property; when an iterator is exhausted, we remove the corresponding pair from the "heap".

```
import heapq

def merge(*subsequences):

    # prepare a priority queue whose items are pairs of the form
    # (current-value, iterator), one each per (non-empty) subsequence
    heap = []

    for subseq in subsequences:
        iterator = iter(subseq)
        for current_value in iterator:
            # subseq is not empty, therefore add this subseq's pair
            # (current-value, iterator) to the list
            heap.append((current_value, iterator))
            break

    # make the priority queue into a heap
    heapq.heapify(heap)

    while heap:
        # get and yield lowest current value (and corresponding iterator)
        current_value, iterator = heap[0]
        yield current_value

        for current_value in iterator:
            # subseq is not finished, therefore add this subseq's pair
            # (current-value, iterator) back into the priority queue
            heapq.heapreplace(heap, (current_value, iterator))
```

```
            break
        else:
            # subseq has been exhausted, therefore remove it from the queue
            heapq.heappop(heap)
```

Note the idiom that we use to advance an iterator by one step, dealing with the possibility that the iterator is exhausted:

```
for current_value in iterator:
    # if we get here the iterator was not empty, current_value was
    # its first value, and the iterator has been advanced one step
    ...use pair (current_value, iterator)...
    # we break at once as we only wanted the first item of iterator
    break
else:
    # if we get here the break did not execute, so the iterator
    # was empty (exhausted)
    # deal with the case of iterator being exhausted...
```

If you find this idiom confusing or tricky, you may prefer a different approach:

```
try:
    current_value = iterator.next()
except StopIteration:
    # if we get here the iterator was empty (exhausted)
    #    deal with the case of iterator being exhausted...
else:
    # if we get here the iterator was not empty, current_value was
    # its first value, and the iterator has been advanced one step
    #   use pair (current_value, iterator)...
```

You have an iterator (or other iterable) object **x**, and need to iterate twice over **x**'s sequence of values.

The need to iterate repeatedly over the same sequence of values is a reasonably common one. In Python 2.4, solving this problem is the job of function **tee** in the standard library module **itertools**:

```
import itertools
x1, x2 = itertools.tee(x)
# you can now iterate on x1 and x2 separately
```

In Python 2.3, you can code **tee** yourself:

```
import itertools
def tee(iterable):
    def yield_with_cache(next, cache={}):
        pop = cache.pop
```

```
        for i in itertools.count():
            try:
                yield pop(i)
            except KeyError:
                cache[i] = next()
                yield cache[i]
    it = iter(iterable)
    return yield_with_cache(it.next), yield_with_cache(it.next)
```

the nested generator **yield_with_cache** makes use of the fact that the default values of arguments get computed just once, at the time the **def** statement executes. Thus, both calls to the nested generator in the **return** statement of **tee** implicitly share the same initially empty **dict** as the value of the **cache** argument.

In general, if one iterator is going to walk over most or all of the data from the original before the "other" one starts advancing, you should consider using **list** instead of **tee**.

There is no guarantee of thread safety: to access the tee'd iterators from different threads, you need to guard those iterators with a single lock!

You are using an iterator for some task such as parsing, which requires you to be able to "look ahead" at the next item the iterator is going to yield, without disturbing the iterator state.

```
import collections

class peekable(object):
    """ An iterator that supports a peek operation.  Example usage:
    >>> p = peekable(range(4))
    >>> p.peek()
    0
    >>> p.next(1)
    [0]
    >>> p.peek(3)
    [1, 2, 3]
    >>> p.next(2)
    [1, 2]
    >>> p.peek(2)
    Traceback (most recent call last):
      ...
    StopIteration
    >>> p.peek(1)
    [3]
    >>> p.next(2)
    Traceback (most recent call last):
```

```
        ...
      StopIteration
      >>> p.next()
      3
      """
      def __init__(self, iterable):
          self._iterable = iter(iterable)
          self._cache = collections.deque()

      def __iter__(self):
          return self

      def _fillcache(self, n):
          if n is None:
              n = 1
          while len(self._cache) < n:
              self._cache.append(self._iterable.next())

      def next(self, n=None):
          self._fillcache(n)
          if n is None:
              result = self._cache.popleft()
          else:
              result = [self._cache.popleft() for i in range(n)]
          return result

      def peek(self, n=None):
          self._fillcache(n)
          if n is None:
              result = self._cache[0]
          else:
              result = [self._cache[i] for i in range(n)]
          return result
```

Many iterator-related tasks, such as parsing, require the ability to "peek ahead" (once or a few times) into the sequence of items that an iterator is yielding, in a way that does not alter the iterator's observable state. One approach is to use the new Python 2.4 function **iterator.tee** to get two independent copies of the iterator, one to be advanced for peeking purposes and the "other" one to be used as the "main" iterator. It's actually handier to wrap the incoming iterator once for all, at the start, with the class **peekable** presented in this recipe; afterwards, a **peek** method, which is safe and effective, can be counted on.

A subtle point is that the **n** argument to methods **peek** and **next** defaults to **None**, not to **1**. This gives you two distinct ways to peek at a single item: the

default way, calling **p.peek()**, just gives you that item, while calling **p.peek(1)** gives you a list with that single item in it.

You have a list of data grouped by a key value, typically read from a spreadsheet or the like, and want to generate a summary of that information for reporting purposes.

```
from itertools import groupby
from operator import itemgetter

def summary(data, key=itemgetter(0), field=itemgetter(1)):
    """ Summarise the given data (a sequence of rows), grouped by the
        given key (default: the first item of each row), giving totals
        of the given field (default: the second item of each row).
        The key and field arguments should be functions which, given a
        data record, return the relevant value.
    """
    for k, group in groupby(data, key):
        yield k, sum(field(row) for row in group)

if __name__ == "__main__":
    # Example: given a sequence of sales data for city within region,
    # _sorted on region_, produce a sales report by region
    sales = [('Scotland', 'Edinburgh', 20000),
            ('Scotland', 'Glasgow', 12500),
            ('Wales', 'Cardiff', 29700),
            ('Wales', 'Bangor', 12800),
            ('England', 'London', 90000),
            ('England', 'Manchester', 45600),
            ('England', 'Liverpool', 29700)]
    for region, total in summary(sales, field=itemgetter(2)):
        print "%10s: %d" % (region, total)
```

In many situations, data is available in tabular form, with the information naturally grouped by a subset of the data values (e.g., recordsets obtained from database queries and data read from spreadsheets typically with the **csv** module of the Python Standard Library). It is often useful to be able to produce summaries of the detail data.

The new **groupby** function (added in Python 2.4 to the **itertools** module of the Python Standard Library) is designed exactly for the purpose of handling such grouped data. It takes as arguments an iterator, whose items are to be thought of as records, along with a function to extract the **key** value from each record.

For a summary report, two extraction functions are required: one function

to extract the key, which is the function that you pass to the **groupby** function, and another function to extract the values to be summarized. The recipe uses another innovation of Python 2.4 for these purposes: the **operator.itemgetter** higher-order function: called with an index **i** as its argument. **itemgetter** produces a function **f** such that **f(x)** extracts the **i**th item from **x**, operating just like an **indexing x[i]**.

The input records must be sorted by the given key; if you're uncertain about that condition, you can use **groubpy(sorted(data, key=key), key)** to ensure it, exploiting the built-in function **sorted**, also new in Python 2.4. It's quite convenient that the same key-extraction function can be passed to both **sorted** and **groupby** in this idiom.

Overall, this recipe provides a vivid illustration of how the new Python 2.4 features work well together: in addition to the **groupby** function, the **operator.itemgetter** used to provide field extraction functions, and the potential use of the built-in function **sorted**, the recipe also uses a generator expression as the argument to the **sum** built-in function.

If you need to implement this recipe's functionality in Python 2.3, you can start by implementing your own approximate version of **groupby.**

```
class groupby(dict):
    def __init__(self, seq, key):
        for value in seq:
            k = key(value)
            self.setdefault(k, []).append(value)
    __iter__ = dict.iteritems
```

Similarly, you can write your own simplified versions of functions **itemgetter** and **sorted.**

```
def itemgetter(i):
    def getter(x): return x[i]
    return getter

def sorted(seq, key):
    aux = [(key(x), i, x) for i, x in enumerate(seq)]
    aux.sort()
    return [x for k, i, x in aux]
```

Descriptors, Decorators,and Metaclasses

Though easy to use, the power tools can be considered advanced for several reasons. First, the need for them rarely arises in simple programs. Second, most involve introspection, wrapping, and forwarding techniques available only in a dynamic language like Python. Third, the tools seem advanced because when you learn them, you also develop a deep understanding of how Python works internally.

The concept of **descriptors** is easy enough. Whenever an attribute is looked up, an action takes place. By default, the action is a get, set, or delete. However, someday you'll be working on an application with some subtle need and wish that more complex actions could be programmed. Perhaps you would like to create a log entry every time a certain attribute is accessed. Perhaps you would like to redirect a method lookup to another method. The solution is to write a function with the needed action and then specify that it be run whenever the attribute is accessed. An object with such functions is called a **descriptor.**

While the concept of a descriptor is straightforward, there seems to be no limit to what can be done with them. Descriptors underlie Python's implementation of methods, bound methods, **super**, **property**, **classmethod**, and **staticmethod**. Learning about the various applications of descriptors is key to mastering the language.

Decorators are even simpler than descriptors. Writing **myfunc=wrapper(myfunc)** was the common way to modify or log something about another function, which took place somewhere after **myfunc** was defined. Starting with Python 2.4, we now write **@wrapper** just before the **def** statement that performs the definition of **myfunc**. Common examples include **@staticmethod** and **@classmethod**. Unlike Java declarations, these wrappers are higher-order functions that can modify the original function or take some other action. Their uses are limitless. Some ideas that have been advanced include **@make_constants** for bytecode optimization, **@atexit** to register a function to be run before Python exits, **@synchronized** to automatically add mutual exclusion locking to a function or method, and **@log** to create a log entry every time a function is called. Such wrapper functions are called **decorators.**

The concept of a **metaclass** sounds strange only because it is so familiar. Whenever you write a class definition, a mechanism uses the name, bases, and class dictionary to create a class object. For old-style classes that mechanism is **types.ClassType**. For new-style classes, the mechanism is just **type**. The former implements the familiar actions of a classic class, including attribute lookup and showing the name of the class when **repr** is called. The latter adds a few bells and whistles including support for **__slots__** and **__getattribute__**. If only that mechanism were programmable, what you could do in Python would be limitless. Well, the mechanism is programmable, and, of course, it has an intimidating

name, **metaclasses**.

Most metaclasses subclass **type** and simply extend or override the desired behavior. Some are as simple as altering the class dictionary and then forwarding the arguments to **type** to finish the job.

```
class M(type):
    def __new__(cls, name, bases, classdict):
        for attr in classdict.get('__slots__', ()):
            if attr.startswith('_'):
                def getter(self, attr=attr):
                    return getattr(self, attr)
                classdict['get' + attr[1:]] = getter
        return type.__new__(cls, name, bases, classdict)

class Point(object):
    __metaclass__ = M
    __slots__ = ['_x', '_y']
```

If you now **print dir(Point)**, you will see the two accessor methods as if you had written them out the long way:

```
class Point(object):
    __slots__ = ['_x', '_y']
    def getx(self):
        return self._x
    def gety(self):
        return self._y
```

Python computes the default values for a function's optional arguments just once, when the function's **def** statement executes. However, for some of your functions, you'd like to ensure that the default values are fresh ones (i.e., new and independent copies) each time a function gets called.

A Python 2.4 decorator offers an elegant solution:

```
import copy

def freshdefaults(f):
    "a decorator to wrap f and keep its default values fresh between calls"
    fdefaults = f.func_defaults
    def refresher(*args, **kwds):
        f.func_defaults = copy.deepcopy(fdefaults)
        return f(*args, **kwds)
    refresher.__name__ = f.__name__
    return refresher
```

```
# usage as a decorator, in python 2.4:
@freshdefaults
def packitem(item, pkg=[]):
    pkg.append(item)
    return pkg
```

A function's default values are evaluated once, and only once, at the time the function is defined. Beginning Python programmers are sometimes surprised by this fact; they try to use mutable default values and yet expect that the values will somehow be regenerated afresh each time they're needed.

Recommended Python practice is to not use mutable default values. Instead, you should use idioms such as:

```
def packitem(item, pkg=None):
    if pkg is None:
        pkg = []
    pkg.append(item)
    return pkg
```

The **freshdefaults** decorator provides another way to accomplish the same task. It eliminates the need to set as your default value anything but the value you intend that optional argument to have by default.

freshdefaults also removes the need to test each argument against the stand-in value before assigning the intended value: this could be an important simplification in your code.

On the other hand, the implementation of **freshdefaults** needs several reasonably advanced concepts: decorators, closures, function attributes, and deep copying. All in all, this implementation is no doubt more difficult to explain to beginning Python programmers than the recommended idiom.

If an outer function just returns an inner function (often a closure), the name of the returned function object is fixed, which can be confusing when the name is shown during introspection or debugging.

```
def make_adder(addend):
    def adder(augend): return augend + addend
    return adder
```

In Python 2.4, you can solve the problem by setting the **__name__** attribute of the inner function right after the end of the inner function's **def** statement, and before the **return** statement from the outer function.

```
def make_adder(addend):
    def adder(augend): return augend + addend
    adder.__name__ = 'add_%s' % (addend,)
    return adder
```

Unfortunately, in Python 2.3, you cannot assign to the **__name__** attribute of a function object; in that release, the attribute is read-only.

You want to use an attribute name as an alias for another one, either just as a default value (when the attribute was not explicitly set), or with full setting and deleting abilities too.

Custom descriptors are the right tools for this task:

```
class DefaultAlias(object):
    ''' unless explicitly assigned, this attribute aliases to another. '''
    def __init__(self, name):
        self.name = name
    def __get__(self, inst, cls):
        if inst is None:
            # attribute accessed on class, return `self' descriptor
            return self
        return getattr(inst, self.name)

class Alias(DefaultAlias):
    ''' this attribute unconditionally aliases to another. '''
    def __set__(self, inst, value):
        setattr(inst, self.name, value)
    def __delete__(self, inst):
        delattr(inst, self.name)

class Book(object):
    def __init__(self, title, shortTitle=None):
        self.title = title
        if shortTitle is not None:
            self.shortTitle = shortTitle
    shortTitle = DefaultAlias('title')
    #shortTitle = Alias('title')
```

Custom descriptor class **Alias** is a simple variant of class **DefaultAlias**, easily obtained by inheritance. **Alias** aliases one attribute to another, not just upon accesses to the attribute's value, but also upon all operations of value setting and deletion.

Alias can be quite useful when you want to evolve a class, which you made publicly available in a previous version, to use more appropriate names for methods and other attributes, while still keeping the old names available for backwards compatibility.

Debugging and Testing

While developing or debugging, you want certain conditional or looping sections of code to be temporarily omitted from execution. If you have many such sections that must simultaneously switch on and off during your development and debug sessions, an alternative is to define a boolean variable (commonly known as a **flag**), say **doit = False**, and code:

```
if doit and i < 1:
  doSomething()
while doit and j < k:
  j = fleep(j, k)
```

This way, you can temporarily switch the various sections on again by just changing the flag setting to **doit = True**, and easily flip back and forth.

One Python-specific technique you can use is the **__debug__** read-only global boolean variable. **__debug__** is **true** when Python is running without the **-O** (optimize) command-line option, **False** when Python is running with that option. Moreover, the Python compiler knows about **__debug__** and can completely remove any block guarded by **if __debug__** when Python is running with the command-line optimization option, thus saving memory as well as execution time.

Documentation strings (**docstrings**) are an important feature that Python offers to help you document your code. Any module, class, function or method can have a string literal as its very first "statement". If so, then Python considers that string to be the docstring for the module, class, function, or method in question and saves it as the **__doc__** attribute of the respective object. Modules, classes, functions, and methods that lack docstrings have **None** as the value of their **__doc__** attribute.

In Python's interactive interpreter, you can examine the "docstring" of an object, as well as other helpful information about the object, with the command **help(theobject)**.