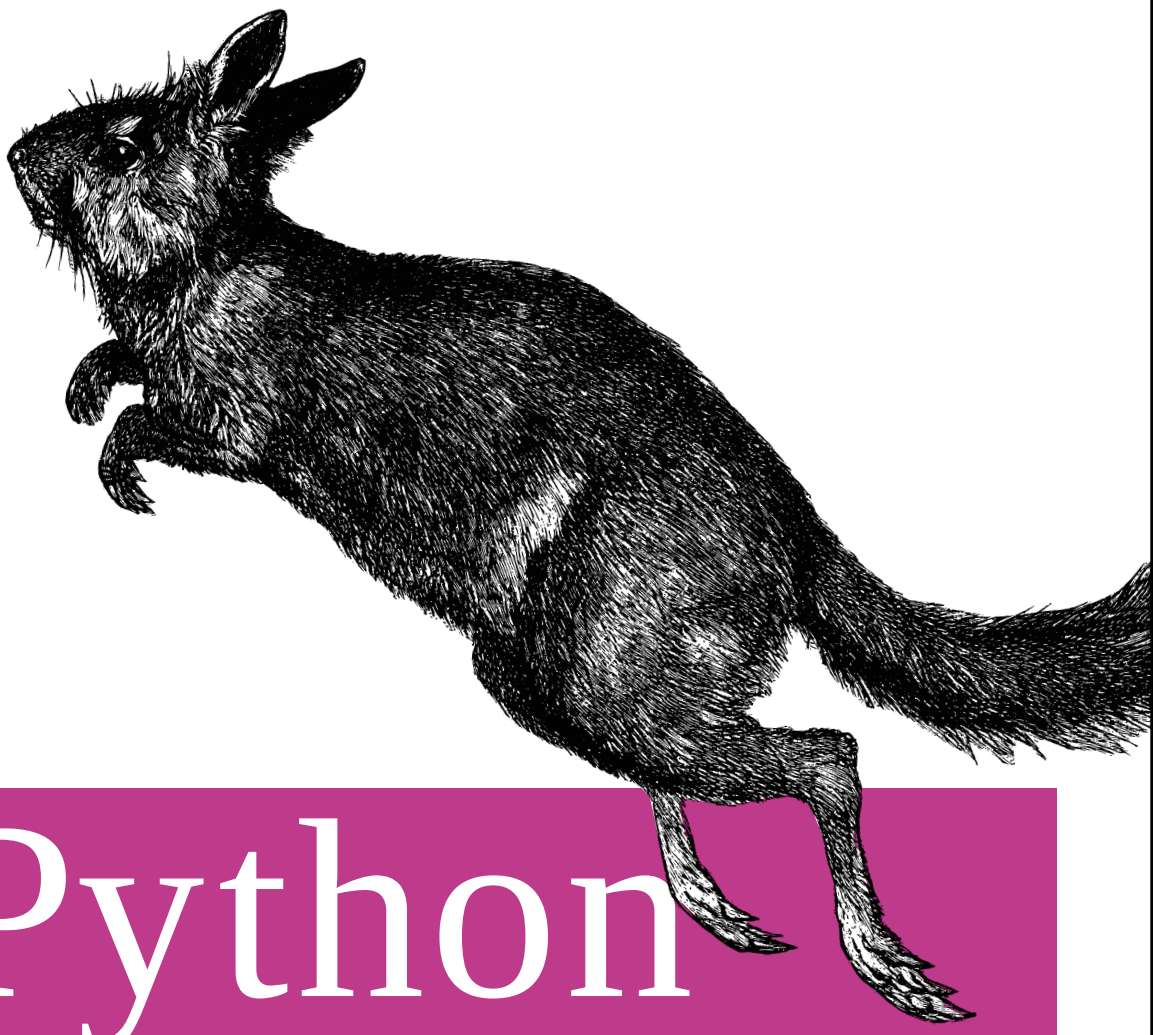
Recipes from the Python Community



Python Cookbook

O'REILLY®

Edited by Alex Martelli & David Ascher

Python Cookbook

Edited by Alex Martelli and David Ascher

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1

Python Shortcuts

1.0 Introduction

Credit: David Ascher, ActiveState, co-author of Learning Python (O'Reilly)

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: unlike the Académie-defined French language, the English language routinely grows words to suit its speakers' needs, such as "carjacking," "earwitness," "snail mail," "email," "googlewhacking," and "blogging." In the world of computer languages, 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. In contrast, as you'll see in many of the discussions of recipes throughout this volume, Python programmers often express their belief in the value of clarity and elegance. As a well-known Perl hacker once said, 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. I still have more fun coding in Python, though.

The reason I bring up these seemingly irrelevant bits at the beginning of this book is that the recipes you see in this first 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. Much like I, a proud resident of Vancouver, will go out of my way to show tourists the really neat things about the city, from the parks to the beaches to the mountains, a Python user will seek out friends and colleagues and say, "You gotta see this!" Programming in Python, in my mind, is a shared social pleasure, not all that competitive. There's 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.

When we identified the recipe categories for this collection, our driving notion was that there would be recipes of various kinds, each aiming to achieve something specific—a souffle recipe, a tart recipe, an osso buco recipe. Those would naturally bunch into fairly typical categories, such as desserts, appetizers, and meat dishes, or their perhaps less appetizing, nonmetaphorical equivalents, such as files, algorithms, and so on. So we picked a list of categories, added the categories to the Zope site used to collect recipes, and opened the floodgates.

Pretty soon, it became clear that some submissions were really hard to fit into the predetermined categories. These recipes are the Pythonic equivalent of making a roux (melted butter or fat combined with flour, used in sauce-making, for those of you without an Italian sauce background), kneading dough, flouring, flipping a pan's contents, blanching, and the myriad other tricks that any accomplished cook knows, but that you won't find in any "straight" recipe book. Many of these tricks and techniques are used in preparing various kinds of meals, but it's hard to pigeonhole them as relevant for a given type of dish. And if you're a novice cook looking up a fancy recipe, you're likely to get frustrated quickly, as these techniques are typically found only in books like *Cooking for Divorced Middle-Aged Men*. We didn't want to exclude this precious category from this book, so a new category was born. That explains why this chapter exists.

This chapter is pretty flimsy, though, in that while the title refers to shortcuts, there is nothing here like what one could have expected had the language in question been Python's venerable cousin, Perl. If this had been a community-authored Perl cookbook, entries in this category would probably have outnumbered those in most other chapters. That is because Perl's syntax provides, proudly, many ways to do pretty much anything. Furthermore, each way is "tricky" in a good way: the writer gets a little thrill out of exploiting an odd corner of the language. That chapter would be impressive, and competitive, and fun. Python programmers just don't get to have that kind of fun on that kind of scale (by which I mean the scale of syntactic shortcuts and semantic-edge cases). No one gives multi-hour talks about tricks of the Python grand masters... Python grand masters simply don't have that many frequently used tricks up their sleeves!

I believe that the recipes in this chapter are among the most time-sensitive of the recipes in this volume. That's because the aspects of the language that people consider shortcuts or noteworthy techniques seem to be relatively straightforward, idiomatic applications of recent language features. List comprehensions, `zip`, and dictionary methods such as `setdefault` are all relatively recent additions to the language, dating from Python 2.0 or later. In fact, many of these newish language features were added to Python to eliminate the need for what used to be fancy recipes.

My favorite recent language features are list comprehensions and the new applicability of the * and ** tokens to function calls as well as to function definitions. List comprehensions have clearly become wildly successful, if the authors of this volume are

representative of the Python community at large, and have largely demoted the `map` and `filter` built-in functions. Less powerful, but equally elegant, are * and **. Since Python 2.0, the oft-quoted recipe:

```
def method(self, argument, *args, **kw):
  # Do something with argument
  apply(callable, args, kw)
```

can now be done much more elegantly as:

```
def method(self, argument, *args, **kw):
  # Do something with argument
  callable(*args, **kw)
```

The `apply` built-in function is still somewhat useful, at least occasionally, but these new syntactic forms are elegant and provably Pythonic. This leads me to my closing comment on language shortcuts: the best source of shortcuts and language tricks is probably the list of language changes that comes with each Python release. Special thanks should be extended to Andrew Kuchling for publishing a list of "What's new with Python 2.x," available at *http://amk.ca/python/*, for each major release since 2.0. It's the place I head for when I want a clear and concise view of Python's recent evolution.

1.1 Swapping Values Without Using a Temporary Variable

Credit: Hamish Lawson

Problem

You want to swap the values of some variables, but you don't want to use a temporary variable.

Solution

Python's automatic tuple packing and unpacking make this a snap:

```
a, b, c = b, c, a
```

Discussion

Most programming languages make you use temporary intermediate variables to swap variable values:

```
temp = a
a = b
b = c
c = temp
```

But Python lets you use tuple packing and unpacking to do a direct assignment:

```
a, b, c = b, c, a
```

In an assignment, Python requires an expression on the righthand side of the `=`. What we wrote there—`b, c, a`—is indeed an expression. Specifically, it is a *tuple*, which is an immutable sequence of three values. Tuples are often surrounded with parentheses, as in `(b, c, a)`, but the parentheses are not necessary, except where the commas would otherwise have some other meaning (e.g., in a function call). The commas are what create a tuple, by *packing* the values that are the tuple's items.

On the lefthand side of the `=` in an assignment statement, you normally use a single *target*. The target can be a simple identifier (also known as a variable), an indexing (such as `alist[i]` or `adict['freep']`), an attribute reference (such as `anobject.someattribute`), and so on. However, Python also lets you use several targets (variables, indexings, etc.), separated by commas, on an assignment's lefthand side. Such a multiple assignment is also called an *unpacking* assignment. When there are two or more comma-separated targets on the lefthand side of an assignment, the value of the righthand side must be a sequence of as many items as there are comma-separated targets on the lefthand side. Each item of the sequence is assigned to the corresponding target, in order, from left to right.

In this recipe, we have three comma-separated targets on the lefthand side, so we need a three-item sequence on the righthand side, the three-item tuple that the packing built. The first target (variable `a`) gets the value of the first item (which used to be the value of variable `b`), the second target (`b`) gets the value of the second item (which used to be the value of `c`), and the third and last target (`c`) gets the value of the third and last item (which used to be the value of `a`). The net result is a swapping of values between the variables (equivalently, you could visualize this particular example as a rotation).

Tuple packing, done using commas, and sequence unpacking, done by placing several comma-separated targets on the lefthand side of a statement, are both useful, simple, general mechanisms. By combining them, you can simply, elegantly, and naturally express any permutation of values among a set of variables.

See Also

The *Reference Manual* section on assignment statements.

1.2 Constructing a Dictionary Without Excessive Quoting

Credit: Brent Burley

Problem

You'd like to construct a dictionary without having to quote the keys.

Solution

Once you get into the swing of Python, you may find yourself constructing a lot of dictionaries. However, the standard way, also known as a *dictionary display*, is just a smidgeon more cluttered than you might like, due to the need to quote the keys. For example:

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

When the keys are identifiers, there's a cleaner way:

```
def makedict(**kwargs):
    return kwargs
data = makedict(red=1, green=2, blue=3)
```

You might also choose to forego some simplicity to gain more power. For example:

```
def dodict(*args, **kwds):
    d = {}
    for k, v in args: d[k] = v
    d.update(kwds)
    return d
tada = dodict(*data.items( ), yellow=2, green=4)
```

Discussion

The syntax for constructing a dictionary can be slightly tedious, due to the amount of quoting required. This recipe presents a technique that avoids having to quote the keys, when they are identifiers that you already know at the time you write the code.

I've often found myself missing Perl's `=>` operator, which is well suited to building hashes (Perl-speak for dictionaries) from a literal list:

```
%data = (red => 1, green => 2, blue => 3);
```

The `=>` operator in Perl is equivalent to Perl's own `,`, except that it implicitly quotes the word to its left.

Perl's syntax is very similar to Python's function-calling syntax for passing keyword arguments. And the fact that Python collects the keyword arguments into a dictionary turned on a light bulb in my head.

When you declare a function in Python, you may optionally conclude the list of formal arguments with *`*args`* or *`**kwds`* (if you want to use both, the one with ** must be last). If you have *`*args`*, your function can be called with any number of extra actual arguments of the positional, or plain, kind. Python collects all the extra positional arguments into a tuple and binds that tuple to the identifier *`args`*. Similarly, if you have *`**kwds`*, your function can be called with any number of extra actual arguments of the named, or keyword, kind. Python collects all the extra named arguments into a dictionary (with the names as the keys and the values as the values) and

binds that dictionary to the identifier *kwds*. This recipe exploits the way that Python knows how to perform the latter task.

The `makedict` function should be very efficient, since the compiler is doing work equivalent to that done with a dictionary literal. It is admittedly idiomatic, but it can make large dictionary literals a lot cleaner and a lot less painful to type. When you need to construct dictionaries from a list of key/item pairs, possibly with explicit override of, or addition to, some specifically named key, the `dodict` function (although less crystal-clear and speedy) can be just as handy. In Python 2.2, the first two lines of `dodict` can be replaced with the more concise and faster equivalent:

```
d = dict(args)
```

See Also

The *Library Reference* section on mapping types.

1.3 Getting a Value from a Dictionary

Credit: Andy McKay

Problem

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.

Solution

That's what the `get` method of dictionaries is for. Say you have a dictionary:

```
d = {'key':'value'}
```

You can write a test to pull out the value of `'key'` from `d` in an exception-safe way:

```
if d.has_key('key'):      # or, in Python 2.2 or later: if 'key' in d:
  print d['key']
else:
  print 'not found'
```

However, there is a much simpler syntax:

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

Discussion

Want to get a value from a dictionary but first make sure that the value exists in the dictionary? Use the simple and useful `get` method.

If you try to get a value with a syntax such as `d[x]`, and the value of `x` is not a key in dictionary `d`, your attempt raises a `KeyError` exception. This is often okay. If you

expected the value of x to be a key in d, an exception is just the right way to inform you that you're wrong (i.e., that you need to debug your program).

However, you often need to be more tentative about it: as far as you know, the value of x may or may not be a key in d. In this case, don't start messing with the has_key method or with try/except statements. Instead, use the get method. If you call d.get(x), no exception is thrown: you get d[x] if x is a key in d, and if it's not, you get None (which you can check for or propagate). If None is not what you want to get when x is not a key of d, call d.get(x, *somethingelse*) instead. In this case, if x is not a key, you will get the value of *somethingelse*.

get is a simple, useful mechanism that is well explained in the Python documentation, but a surprising number of people don't know about it. This idiom is also quite common in Zope, for example, when pulling variables out of the REQUEST dictionary.

See Also

The *Library Reference* section on mapping types.

1.4 Adding an Entry to a Dictionary

Credit: Alex Martelli

Problem

Working with a dictionary D, you need to use the entry D[k] if it's already present, or add a new D[k] if k isn't yet a key in D.

Solution

This is what the setdefault method of dictionary objects is for. Say we're building a word-to-page numbers index. A key piece of code might be:

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

Good Pythonic instincts suggest substituting this "look before you leap" pattern with an "easier to get permission" pattern (see Recipe 5.3 for a detailed discussion of these phrases):

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

This is just a minor simplification, but it satisfies the pattern of "use the entry if it is already present; otherwise, add a new entry." Here's how using `setdefault` simplifies this further:

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

Discussion

The `setdefault` method of a dictionary is a handy shortcut for this task that is especially useful when the new entry you want to add is mutable. Basically, `dict.setdefault(k, v)` is much like `dict.get(k, v)`, except that if `k` is not a key in the dictionary, the `setdefault` method assigns `dict[k]=v` as a side effect, in addition to returning `v`. (`get` would just return `v`, without affecting `dict` in any way.) Therefore, `setdefault` is appropriate any time you have `get`-like needs but also want to produce this specific side effect on the dictionary.

`setdefault` is particularly useful in a dictionary with values that are lists, as detailed in Recipe 1.5. The single most typical usage form for `setdefault` is:

```
somedict.setdefault(somekey, []).append(somevalue)
```

Note that `setdefault` is normally not very useful if the values are immutable. If you just want to count words, for example, something like the following is no use:

```
theIndex.setdefault(word, 1)
```

In this case, you want:

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

since you will be rebinding the dictionary entry at `theIndex[word]` anyway (because numbers are immutable).

See Also

Recipe 5.3; the *Library Reference* section on mapping types.

1.5 Associating Multiple Values with Each Key in a Dictionary

Credit: Michael Chermside

Problem

You need a dictionary that maps each key to multiple values.

Solution

By nature, a dictionary is a one-to-one mapping, but it's not hard to make it one-to-many—in other words, to make one key map to multiple values. There are two possible approaches, depending on how you want to treat duplications in the set of values for a key. The following approach allows such duplications:

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

while this approach automatically eliminates duplications:

```
d2 = {}
d2.setdefault(key, {})[value] = 1
```

Discussion

A normal dictionary performs a simple mapping of a key to a value. This recipe shows two easy, efficient ways to achieve a mapping of each key to multiple values. The semantics of the two approaches differ slightly but importantly in how they deal with duplication. Each approach relies on the `setdefault` method of a dictionary to initialize the entry for a key in the dictionary, if needed, and in any case to return said entry.

Of course, you need to be able to do more than just add values for a key. With the first approach, which allows duplications, here's how to retrieve the list of values for a key:

```
list_of_values = d1[key]
```

Here's how to remove one value for a key, if you don't mind leaving empty lists as items of `d1` when the last value for a key is removed:

```
d1[key].remove(value)
```

Despite the empty lists, it's still easy to test for the existence of a key with at least one value:

```
def has_key_with_some_values(d, key):
    return d.has_key(key) and d[key]
```

This returns either `0` or a list, which may be empty. In most cases, it is easier to use a function that always returns a list (maybe an empty one), such as:

```
def get_values_if_any(d, key):
    return d.get(key, [])
```

You can use either of these functions in a statement. For example:

```
if get_values_if_any(d1, somekey):
if has_key_with_some_values(d1, somekey):
```

However, `get_values_if_any` is generally handier. For example, you can use it to check if `'freep'` is among the values for `somekey`:

```
if 'freep' in get_values_if_any(d1, somekey):
```

This extra handiness comes from `get_values_if_any` always returning a list, rather than sometimes a list and sometimes `0`.

The first approach allows each value to be present multiple times for each given key. For example:

```
example = {}
example.setdefault('a', []).append('apple')
example.setdefault('b', []).append('boots')
example.setdefault('c', []).append('cat')
example.setdefault('a', []).append('ant')
example.setdefault('a', []).append('apple')
```

Now `example['a']` is `['apple', 'ant', 'apple']`. If we now execute:

```
example['a'].remove('apple')
```

the following test is still satisfied:

```
if 'apple' in example['a']
```

`'apple'` was present twice, and we removed it only once. (Testing for `'apple'` with `get_values_if_any(example, 'a')` would be more general, although equivalent in this case.)

The second approach, which eliminates duplications, requires rather similar idioms. Here's how to retrieve the list of the values for a key:

```
list_of_values = d2[key].keys( )
```

Here's how to remove a key/value pair, leaving empty dictionaries as items of `d2` when the last value for a key is removed:

```
del d2[key][value]
```

The `has_key_with_some_values` function shown earlier also works for the second approach, and you also have analogous alternatives, such as:

```
def get_values_if_any(d, key):
    return d.get(key, {}).keys( )
```

The second approach doesn't allow duplication. For example:

```
example = {}
example.setdefault('a', {})['apple']=1
example.setdefault('b', {})['boots']=1
example.setdefault('c', {})['cat']=1
example.setdefault('a', {})['ant']=1
example.setdefault('a', {})['apple']=1
```

Now `example['a']` is `{'apple':1, 'ant':1}`. Now, if we execute:

```
del example['a']['apple']
```

the following test is not satisfied:

```
if 'apple' in example['a']
```

`'apple'` was present, but we just removed it.

This recipe focuses on how to code the raw functionality, but if you want to use this functionality in a systematic way, you'll want to wrap it up in a class. For that purpose, you need to make some of the design decisions that the recipe highlights. Do you want a value to be in the entry for a key multiple times? (Is the entry a bag rather than a set, in mathematical terms?) If so, should `remove` just reduce the number of occurrences by 1, or should it wipe out all of them? This is just the beginning of the choices you have to make, and the right choices depend on the specifics of your application.

See Also

The *Library Reference* section on mapping types.

1.6 Dispatching Using a Dictionary

Credit: Dick Wall

Problem

You need to execute appropriate pieces of code in correspondence with the value of some control variable—the kind of problem that in some other languages you might approach with a `case`, `switch`, or `select` statement.

Solution

Object-oriented programming, thanks to its elegant concept of dispatching, does away with many (but not all) such needs. But dictionaries, and the fact that in Python functions are first-class values (in particular, they can be values in a dictionary), conspire to make the problem quite easy to solve:

```
animals = []
number_of_felines = 0

def deal_with_a_cat( ):
    global number_of_felines
    print "meow"
    animals.append('feline')
    number_of_felines += 1
```

```
def deal_with_a_dog( ):
    print "bark"
    animals.append('canine')

def deal_with_a_bear( ):
    print "watch out for the *HUG*!"
    animals.append('ursine')

tokenDict = {
    "cat": deal_with_a_cat,
    "dog": deal_with_a_dog,
    "bear": deal_with_a_bear,
    }

# Simulate, say, some words read from a file
words = ["cat", "bear", "cat", "dog"]

for word in words:
    # Look up the function to call for each word, then call it
    functionToCall = tokenDict[word]
    functionToCall( )
    # You could also do it in one step, tokenDict[word]( )
```

Discussion

The basic idea behind this recipe is to construct a dictionary with string (or other) keys and with bound methods, functions, or other callables as values. During execution, at each step, use the string keys to select which method or function to execute. This can be used, for example, for simple parsing of tokens from a file through a kind of generalized `case` statement.

It's embarrassingly simple, but I use this technique often. Instead of functions, you can also use bound methods (such as `self.method1`) or other callables. If you use unbound methods (such as `class.method`), you need to pass an appropriate object as the first actual argument when you do call them. More generally, you can also store tuples, including both callables and arguments, as the dictionary's values, with diverse possibilities.

I primarily use this in places where in other languages I might want a `case`, `switch`, or `select` statement. I also use it to provide a poor man's way to parse command files (e.g., an X10 macro control file).

See Also

The *Library Reference* section on mapping types; the *Reference Manual* section on bound and unbound methods.

1.7 Collecting a Bunch of Named Items

Credit: Alex Martelli

Problem

You want to collect a bunch of items together, naming each item of the bunch, and you find dictionary syntax a bit heavyweight for the purpose.

Solution

Any (classic) class inherently wraps a dictionary, and we take advantage of this:

```
class Bunch:
    def __init__(self, **kwds):
        self.__dict__.update(kwds)
```

Now, to group a few variables, create a `Bunch` instance:

```
point = Bunch(datum=y, squared=y*y, coord=x)
```

You can access and rebind the named attributes just created, add others, remove some, and so on. For example:

```
if point.squared > threshold:
    point.isok = 1
```

Discussion

Often, we just want to collect a bunch of stuff together, naming each item of the bunch; a dictionary's okay for that, but a small do-nothing class is even handier and is prettier to use.

A dictionary is fine for collecting a few items in which each item has a name (the item's key in the dictionary can be thought of as the item's name, in this context). However, when all names are identifiers, to be used just like variables, the dictionary-access syntax is not maximally clear:

```
if point['squared'] > threshold
```

It takes minimal effort to build a little class, as in this recipe, to ease the initialization task and provide elegant attribute-access syntax:

```
if bunch.squared > threshold
```

An equally attractive alternative implementation to the one used in the solution is:

```
class EvenSimplerBunch:
    def __init__(self, **kwds): self.__dict__ = kwds
```

The alternative presented in the `Bunch` class has the advantage of not rebinding `self.__dict__` (it uses the dictionary's `update` method to modify it instead), so it

will keep working even if, in some hypothetical far-future dialect of Python, this specific dictionary became nonrebindable (as long, of course, as it remains mutable). But this `EvenSimplerBunch` is indeed even simpler, and marginally speedier, as it just rebinds the dictionary.

It is not difficult to add special methods to allow attributes to be accessed as `bunch['squared']` and so on. In Python 2.1 or earlier, for example, the simplest way is:

```
import operator

class MurkierBunch:
    def __init__(self, **kwds):
        self.__dict__ = kwds
    def __getitem__(self, key):
        return operator.getitem(self.__dict__, key)
    def __setitem__(self, key, value):
        return operator.setitem(self.__dict__, key, value)
    def __delitem__(self, key):
        return operator.delitem(self.__dict__, key)
```

In Python 2.2, we can get the same effect by inheriting from the `dict` built-in type and delegating the other way around:

```
class MurkierBunch22(dict):
    def __init__(self, **kwds): dict.__init__(self, kwds)
    __getattr__ = dict.__getitem__
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__
```

Neither approach makes these `Bunch` variants into fully fledged dictionaries. There are problems with each—for example, what is `someBunch.keys` supposed to mean? Does it refer to the method returning the list of keys, or is it just the same thing as `someBunch['keys']`? It's definitely better to avoid such confusion: Python distinguishes between attributes and items for clarity and simplicity. However, many newcomers to Python do believe they desire such confusion, generally because of previous experience with JavaScript, in which attributes and items are regularly confused. Such idioms, however, seem to have little usefulness in Python. For occasional access to an attribute whose name is held in a variable (or otherwise runtime-computed), the built-in functions `getattr`, `setattr`, and `delattr` are quite adequate, and they are definitely preferable to complicating the delightfully simple little `Bunch` class with the semantically murky approaches shown in the previous paragraph.

See Also

The *Tutorial* section on classes.

1.8 Finding the Intersection of Two Dictionaries

Credit: Andy McKay, Chris Perkins, Sami Hangaslammi

Problem

Given two dictionaries, you need to find the set of keys that are in both dictionaries.

Solution

Dictionaries are a good concrete representation for sets in Python, so operations such as intersections are common. Say you have two dictionaries (but pretend that they each contain thousands of items):

```
some_dict = { 'zope':'zzz', 'python':'rocks' }
another_dict = { 'python':'rocks', 'perl':'$' }
```

Here's a bad way to find their intersection that is very slow:

```
intersect = []
for item in some_dict.keys( ):
    if item in another_dict.keys( ):
        intersect.append(item)
print "Intersects:", intersect
```

And here's a good way that is simple and fast:

```
intersect = []
for item in some_dict.keys( ):
    if another_dict.has_key(item):
        intersect.append(item)
print "Intersects:", intersect
```

In Python 2.2, the following is elegant and even faster:

```
print "Intersects:", [k for k in some_dict if k in another_dict]
```

And here's an alternate approach that wins hands down in speed, for Python 1.5.2 and later:

```
print "Intersects:", filter(another_dict.has_key, some_dict.keys())
```

Discussion

The `keys` method produces a list of all the keys of a dictionary. It can be pretty tempting to fall into the trap of just using `in`, with this list as the righthand side, to test for membership. However, in the first example, you're looping through all of `some_dict`, then each time looping through all of `another_dict`. If `some_dict` has $N1$ items, and `another_dict` has $N2$ items, your intersection operation will have a compute time proportional to the product of $N1 \times N2$. ($O(N1 \times N2)$ is the common computer-science notation to indicate this.)

By using the has_key method, you are not looping on another_dict any more, but rather checking the key in the dictionary's hash table. The processing time for has_key is basically independent of dictionary size, so the second approach is $O(N1)$. The difference is quite substantial for large dictionaries! If the two dictionaries are very different in size, it becomes important to use the smaller one in the role of some_dict, while the larger one takes on the role of another_dict (i.e., loop on the keys of the smaller dictionary, thus picking the smaller $N1$).

Python 2.2 lets you iterate on a dictionary's keys directly, with the statement:

```
for key in dict
```

You can test membership with the equally elegant:

```
if key in dict
```

rather than the equivalent but syntactically less nice dict.has_key(key). Combining these two small but nice innovations of Python 2.2 with the list-comprehension notation introduced in Python 2.0, we end up with a very elegant approach, which is at the same time concise, clear, and quite speedy.

However, the fastest approach is the one that uses filter with the bound method another_dict.has_key on the list some_dict.keys. A typical intersection of two 500-item dictionaries with 50% overlap, on a typical cheap machine of today (AMD Athlon 1.4GHz, DDR2100 RAM, Mandrake Linux 8.1), took 710 microseconds using has_key, 450 microseconds using the Python 2.2 technique, and 280 microseconds using the filter-based way. While these speed differences are almost substantial, they pale in comparison with the timing of the bad way, for which a typical intersection took 22,600 microseconds—30 times longer than the simple way and 80 times longer than the filter-based way! Here's the timing code, which shows a typical example of how one goes about measuring relative speeds of equivalent Python constructs:

```
import time

def timeo(fun, n=1000):
    def void( ): pass
    start = time.clock( )
    for i in range(n): void( )
    stend = time.clock( )
    overhead = stend - start
    start = time.clock( )
    for i in range(n): fun( )
    stend = time.clock( )
    thetime = stend-start
    return fun.__name__, thetime-overhead

to500 = {}
for i in range(500): to500[i] = 1
evens = {}
for i in range(0, 1000, 2): evens[i] = 1
```

```
def simpleway( ):
    result = []
    for k in to500.keys( ):
        if evens.has_key(k):
            result.append(k)
    return result

def pyth22way( ):
    return [k for k in to500 if k in evens]

def filterway( ):
    return filter(evens.has_key, to500.keys( ))

def badsloway( ):
    result = []
    for k in to500.keys( ):
        if k in evens.keys( ):
            result.append(k)
    return result

for f in simpleway, pyth22way, filterway, badsloway:
    print "%s: %.2f"%timeo(f)
```

You can save this code into a *.py* file and run it (a few times, on an otherwise quiescent machine, of course) with `python -O` to check how the timings of the various constructs compare on any specific machine in which you're interested. (Note that this script requires Python 2.2 or later.) Timing different code snippets to find out how their relative speeds compare is an important Python technique, since intuition is a notoriously unreliable guide to such relative-speed comparisons. For detailed and general instruction on how to time things, see the introduction to Chapter 17.

When applicable without having to use a `lambda` form or a specially written function, `filter`, `map`, and `reduce` often offer the fastest solution to any given problem. Of course, a clever Pythonista cares about speed only for those very, very few operations where speed really matters more than clarity, simplicity, and elegance! But these built-ins are pretty elegant in their own way, too.

We don't have a separate recipe for the union of the keys of two dictionaries, but that's because the task is even easier, thanks to a dictionary's `update` method:

```
def union_keys(some_dict, another_dict):
    temp_dict = some_dict.copy( )
    temp_dict.update(another_dict)
    return temp_dict.keys( )
```

See Also

The *Library Reference* section on mapping types.

1.9 Assigning and Testing with One Statement

Credit: Alex Martelli

Problem

You are transliterating C or Perl code to Python, and, to keep close to the original's structure, you need an expression's result to be both assigned and tested (as in `if((x=foo( ))` or `while((x=foo( ))` in such other languages).

Solution

In Python, you can't code:

```
if x=foo( ):
```

Assignment is a statement, so it cannot fit into an expression, which is necessary for conditions of `if` and `while` statements. Normally this isn't a problem, as you can just structure your code around it. For example, this is quite Pythonic:

```
while 1:
    line = file.readline( )
    if not line: break
    process(line)
```

In modern Python, this is far better, but it's even farther from C-like idioms:

```
for line in file.xreadlines( ):
    process(line)
```

In Python 2.2, you can be even simpler and more elegant:

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

But sometimes you're transliterating C, Perl, or some other language, and you'd like your transliteration to be structurally close to the original.

One simple utility class makes this easy:

```
class DataHolder:
    def __init__(self, value=None):
        self.value = value
    def set(self, value):
        self.value = value
        return value
    def get(self):
        return self.value
# optional and strongly discouraged, but handy at times:
import __builtin__
__builtin__.DataHolder = DataHolder
__builtin__.data = DataHolder( )
```

With the help of the DataHolder class and its data instance, you can keep your C-like code structure intact in transliteration:

```
while data.set(file.readline( )):
    process(data.get( ))
```

Discussion

In Python, assignment is not an expression. Thus, you cannot assign the result that you are testing in, for example, an if, elif, or while statement. This is usually okay: you just structure your code to avoid the need to assign while testing (in fact, your code will often become clearer as a result). However, sometimes you may be writing Python code that is the transliteration of code originally written in C, Perl, or another language that supports assignment-as-expression. For example, such transliteration often occurs in the first Python version of an algorithm for which a reference implementation is supplied, an algorithm taken from a book, and so on. In such cases, having the structure of your initial transliteration be close to that of the code you're transcribing is often preferable. Fortunately, Python offers enough power to make it pretty trivial to satisfy this requirement.

We can't redefine assignment, but we can have a method (or function) that saves its argument somewhere and returns that argument so it can be tested. That "somewhere" is most naturally an attribute of an object, so a method is a more natural choice than a function. Of course, we could just retrieve the attribute directly (i.e., the get method is redundant), but it looks nicer to have symmetry between data.set and data.get.

Special-purpose solutions, such as the xreadlines method of file objects, the similar decorator function in the xreadlines module, and (not so special-purpose) Python 2.2 iterators, are obviously preferable for the purposes for which they've been designed. However, such constructs can imply even wider deviation from the structure of the algorithm being transliterated. Thus, while they're great in themselves, they don't really address the problem presented here.

data.set(whatever) can be seen as little more than syntactic sugar for data.value=whatever, with the added value of being acceptable as an expression. Therefore, it's the one obviously right way to satisfy the requirement for a reasonably faithful transliteration. The only difference is the syntactic sugar variation needed, and that's a minor issue.

Importing __builtin__ and assigning to its attributes is a trick that basically defines a new built-in object at runtime. All other modules will automatically be able to access these new built-ins without having to do an import. It's not good practice, though, since readers of those modules should not need to know about the strange side

effects of other modules in the application. Nevertheless, it's a trick worth knowing about in case you encounter it.

Not recommended, in any case, is the following abuse of list format as comprehension syntax:

```
while [line for line in (file.readline(),) if line]:
    process(line)
```

It works, but it is unreadable and error-prone.

See Also

The *Tutorial* section on classes; the documentation for the `builtin` module in the *Library Reference*.

1.10 Using List Comprehensions Instead of map and filter

Credit: Luther Blissett

Problem

You want to perform an operation on all the elements of a list, but you'd like to avoid using `map` and `filter` because they can be hard to read and understand, particularly when they need `lambda`.

Solution

Say you want to create a new list by adding 23 to each item of some other list. In Python 1.5.2, the solution is:

```
thenewlist = map(lambda x: x + 23, theoldlist)
```

This is hardly the clearest code. Fortunately, since Python 2.0, we can use a list comprehension instead:

```
thenewlist = [x + 23 for x in theoldlist]
```

This is much clearer and more elegant.

Similarly, say you want the new list to comprise all items in the other list that are larger than 5. In Python 1.5.2, the solution is:

```
thenewlist = filter(lambda x: x > 5, theoldlist)
```

But in modern Python, we can use the following list comprehension:

```
thenewlist = [x for x in theoldlist if x > 5]
```

Now say you want to combine both list operations. In Python 1.5.2, the solution is quite complex:

```
thenewlist = map(lambda x: x+23, filter(lambda x: x>5, theoldlist))
```

A list comprehension affords far greater clarity, as we can both perform selection with the `if` clause and use some expression, such as adding 23, on the selected items:

```
thenewlist = [x + 23 for x in theoldlist if x > 5]
```

Discussion

Elegance and clarity, within a generally pragmatic attitude, are Python's core values. List comprehensions, added in Python 2.0, delightfully display how pragmatism can enhance both clarity and elegance. The built-in `map` and `filter` functions still have their uses, since they're arguably of equal elegance and clarity as list comprehensions when the `lambda` construct is not necessary. In fact, when their first argument is another built-in function (i.e., when `lambda` is not involved and there is no need to write a function just for the purpose of using it within a `map` or `filter`), they can be even faster than list comprehensions.

All in all, Python programs optimally written for 2.0 or later use far fewer `map` and `filter` calls than similar programs written for 1.5.2. Most of the `map` and `filter` calls (and quite a few explicit loops) are replaced with list comprehensions (which Python borrowed, after some prettying of the syntax, from Haskell, described at *http://www.haskell.org*). It's not an issue of wanting to play with a shiny new toy (although that desire, too, has its place in a programmer's heart)—the point is that the toy, when used well, is a wonderfully useful instrument, further enhancing your Python programs' clarity, simplicity, and elegance.

See Also

The *Reference Manual* section on list displays (the other name for list comprehensions).

1.11 Unzipping Simple List-Like Objects

Credit: gyro funch

Problem

You have a sequence and need to pull it apart into a number of pieces.

Solution

There's no built-in `unzip` counterpart to `zip`, but it's not hard to code our own:

```
def unzip(p, n):
    """ Split a sequence p into a list of n tuples, repeatedly taking the
```

```
next unused element of p and adding it to the next tuple.  Each of the
resulting tuples is of the same length; if p%n != 0, the shorter tuples
are padded with None (closer to the behavior of map than to that of zip).
    Example:
    >>> unzip(['a','b','c','d','e'], 3)
    [('a', 'd'), ('b', 'e'), ('c', None)]
"""
# First, find the length for the longest sublist
mlen, lft = divmod(len(p), n)
if lft != 0: mlen += 1

# Then, initialize a list of lists with suitable lengths
lst = [[None]*mlen for i in range(n)]

# Loop over all items of the input sequence (index-wise), and
# Copy a reference to each into the appropriate place
for i in range(len(p)):
    j, k = divmod(i, n)    # Find sublist-index and index-within-sublist
    lst[k][j] = p[i]       # Copy a reference appropriately

# Finally, turn each sublist into a tuple, since the unzip function
# is specified to return a list of tuples, not a list of lists
return map(tuple, lst)
```

Discussion

The function in this recipe takes a list and pulls it apart into a user-defined number of pieces. It acts like a sort of reverse `zip` function (although it deals with only the very simplest cases). This recipe was useful to me recently when I had to take a Python list and break it down into a number of different pieces, putting each consecutive item of the list into a separate sublist.

Preallocating the result as a list of lists of `None` is generally more efficient than building up each sublist by repeated calls to `append`. Also, in this case, it already ensures the padding with `None` that we would need anyway (unless `length(p)` just happens to be a multiple of `n`).

The algorithm that `unzip` uses is quite simple: a reference to each item of the input sequence is placed into the appropriate item of the appropriate sublist. The built-in function `divmod` computes the quotient and remainder of a division, which just happen to be the indexes we need for the appropriate sublist and item in it.

Although we specified that `unzip` must return a list of tuples, we actually build a list of sublists, and we turn each sublist into a tuple as late in the process as possible by applying the built-in function `tuple` over each sublist with a single call to `map`. It is much simpler to build sublists first. Lists are mutable, so we can bind specific items separately; tuples are immutable, so we would have a harder time working with them in our `unzip` function's main loop.

See Also

Documentation for the `zip` and `divmod` built-ins in the *Library Reference*.

1.12 Flattening a Nested Sequence

Credit: Luther Blissett

Problem

You have a sequence, such as a list, some of whose items may in turn be lists, and so on. You need to flatten it out into a sequence of its scalar items (the leaves, if you think of the nested sequence as a tree).

Solution

Of course, we need to be able to tell which of the elements we're handling are to be deemed scalar. For generality, say we're passed as an argument a predicate that defines what is scalar—a function that we can call on any element and that returns `1` if the element is scalar or `0` otherwise. Given this, one approach is:

```
def flatten(sequence, scalarp, result=None):
    if result is None: result = []
    for item in sequence:
        if scalarp(item): result.append(item)
        else: flatten(item, scalarp, result)
    return result
```

In Python 2.2, a simple generator is an interesting alternative, and, if all the caller needs to do is loop over the flattened sequence, may save the memory needed for the `result` list:

```
from __future__ import generators
def flatten22(sequence, scalarp):
    for item in sequence:
        if scalarp(item):
            yield item
        else:
            for subitem in flatten22(item, scalarp):
                yield subitem
```

Discussion

The only problem with this recipe is that determining what is a scalar is not as obvious as it might seem, which is why I delegated that decision to a callable predicate argument that the caller is supposed to pass to `flatten`. Of course, we must be able to loop over the items of any non-scalar with a `for` statement, or `flatten` will raise an

exception (since it does, via a recursive call, attempt a `for` statement over any non-scalar item). In Python 2.2, that's easy to check:

```
def canLoopOver(maybeIterable):
    try: iter(maybeIterable)
    except: return 0
    else: return 1
```

The built-in function `iter`, new in Python 2.2, returns an iterator, if possible. `for x in s` implicitly calls the `iter` function, so the `canLoopOver` function can easily check if `for` is applicable by calling `iter` explicitly and seeing if that raises an exception.

In Python 2.1 and earlier, there is no `iter` function, so we have to try more directly:

```
def canLoopOver(maybeIterable):
    try:
        for x in maybeIterable:
            return 1
        else:
            return 1
    except:
        return 0
```

Here we have to rely on the `for` statement itself raising an exception if `maybeIterable` is not iterable after all. Note that this approach is not fully suitable for Python 2.2: if `maybeIterable` is an iterator object, the `for` in this approach consumes its first item.

Neither of these implementations of `canLoopOver` is entirely satisfactory, by itself, as our scalar-testing predicate. The problem is with strings, Unicode strings, and other string-like objects. These objects are perfectly good sequences, and we could loop on them with a `for` statement, but we typically want to treat them as scalars. And even if we didn't, we would at least have to treat any string-like objects with a length of 1 as scalars. Otherwise, since such strings are iterable and yield themselves as their only items, our `flatten` function would not cease recursion until it exhausted the call stack and raised a `RuntimeError` due to "maximum recursion depth exceeded."

Fortunately, we can easily distinguish string-like objects by attempting a typical string operation on them:

```
def isStringLike(obj):
    try: obj+''
    except TypeError: return 0
    else: return 1
```

Now, we finally have a good implementation for the scalar-checking predicate:

```
def isScalar(obj):
    return isStringLike(obj) or not canLoopOver(obj)
```

By simply placing this `isScalar` function and the appropriate implementation of `canLoopOver` in our module, before the recipe's functions, we can change the signatures of these functions to make them easier to call in most cases. For example:

```
def flatten22(sequence, scalarp=isScalar):
```

Now the caller needs to pass the `scalarp` argument only in those (hopefully rare) cases where our definition of what is scalar does not quite meet the caller's application-specific needs.

See Also

The *Library Reference* section on sequence types.

1.13 Looping in Parallel over Index and Sequence Items

Credit: Alex Martelli

Problem

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

Solution

Together, the built-in functions `xrange` and `zip` make this easy. You need only this one instance of `xrange`, as it is fully reusable:

```
indices = xrange(sys.maxint)
```

Here's how you use the `indices` instance:

```
for item, index in zip(sequence, indices):
    something(item, index)
```

This gives the same semantics as:

```
for index in range(len(sequence)):
    something(sequence[index], index)
```

but the change of emphasis allows greater clarity in many usage contexts.

Another alternative is to use class wrappers:

```
class Indexed:
    def __init__(self, seq):
        self.seq = seq
    def __getitem__(self, i):
        return self.seq[i], i
```

For example:

```
for item, index in Indexed(sequence):
    something(item, index)
```

In Python 2.2, with `from __future__ import generators`, you can also use:

```
def Indexed(sequence):
    iterator = iter(sequence)
```

```
        for index in indices:
            yield iterator.next( ), index
        # Note that we exit by propagating StopIteration when .next raises it!
```

However, the simplest roughly equivalent way remains the good old:

```
def Indexed(sequence):
    return zip(sequence, indices)
```

Discussion

We often want to loop on a sequence but also need the current index in the loop body. The canonical Pydiom for this is:

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

using `sequence[i]` as the item reference in the loop's body. However, in many contexts, it is clearer to emphasize the loop on the sequence items rather than on the indexes. `zip` provides an easy alternative, looping on indexes and items in parallel, since it truncates at the shortest of its arguments. Thus, it's okay for some arguments to be unbounded sequences, as long as not all the arguments are unbounded. An unbounded sequence of indexes is trivial to write (`xrange` is handy for this), and a reusable instance of that sequence can be passed to `zip`, in parallel to the sequence being indexed.

The same `zip` usage also affords a client code–transparent alternative to the use of a wrapper class `Indexed`, as demonstrated by the `Indexed` class, generator, and function shown in the solution. Of these, when applicable, `zip` is simplest.

The performance of each of these solutions is roughly equivalent. They're all $O(N)$ (i.e., they execute in time proportional to the number of elements in the sequence), they all take $O(1)$ extra memory, and none is anything close to twice as fast or as slow as another.

Note that `zip` is not lazy (i.e., it cannot accept all argument sequences being unbounded). Therefore, in certain cases in which `zip` cannot be used (albeit not the typical one in which `range(len(sequence))` is the alternative), other kinds of loop might be usable. See Recipe 17.12 for lazy, iterator-based alternatives, including an `xzip` function (Python 2.2 only).

See Also

Recipe 17.12; the *Library Reference* section on sequence types.

1.14 Looping Through Multiple Lists

Credit: Andy McKay

Problem

You need to loop through every item of multiple lists.

Solution

There are basically three approaches. Say you have:

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

Using the built-in function `map`, with a first argument of `None`, you can iterate on both lists in parallel:

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

The loop runs three times. On the last iteration, `y` will be `None`.

Using the built-in function `zip` also lets you iterate in parallel:

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

The loop runs two times; the third iteration simply is not done.

A list comprehension affords a very different iteration:

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

The loop runs six times, over each item of `b` for each item of `a`.

Discussion

Using `map` with `None` as the first argument is a subtle variation of the standard `map` call, which typically takes a function as the first argument. As the documentation indicates, if the first argument is `None`, the identity function is used as the function through which the arguments are mapped. If there are multiple list arguments, `map` returns a list consisting of tuples that contain the corresponding items from all lists (in other words, it's a kind of transpose operation). The list arguments may be any kind of sequence, and the result is always a list.

Note that the first technique returns None for sequences in which there are no more elements. Therefore, the output of the first loop is:

```
Map:
a1 b1
a2 b2
a3 None
```

zip lets you iterate over the lists in a similar way, but only up to the number of elements of the smallest list. Therefore, the output of the second technique is:

```
Zip:
a1 b1
a2 b2
```

Python 2.0 introduced list comprehensions, with a syntax that some found a bit strange:

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

This iterates over list b for every element in a. These elements are put into a tuple (x, y). We then iterate through the resulting list of tuples in the outermost for loop. The output of the third technique, therefore, is quite different:

```
List comprehension:
a1 b1
a1 b2
a2 b1
a2 b2
a3 b1
a3 b2
```

See Also

The *Library Reference* section on sequence types; documentation for the zip and map built-ins in the *Library Reference*.

1.15 Spanning a Range Defined by Floats

Credit: Dinu C. Gherman, Paul M. Winkler

Problem

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

Solution

Although this functionality is not available as a built-in, it's not hard to code it with a loop:

```
def frange(start, end=None, inc=1.0):
    "A range-like function that does accept float increments..."
```

```
    if end == None:
        end = start + 0.0     # Ensure a float value for 'end'
        start = 0.0
    assert inc                # sanity check

    L = []
    while 1:
        next = start + len(L) * inc
        if inc > 0 and next >= end:
            break
        elif inc < 0 and next <= end:
            break
        L.append(next)

    return L
```

Discussion

Sadly missing in the Python standard library, the function in this recipe lets you use ranges, just as with the built-in function `range`, but with float arguments.

Many theoretical restrictions apply, but this function is more useful in practice than in theory. People who work with floating-point numbers all the time have many war stories about billion-dollar projects that failed because someone did not take into consideration the strange things that modern hardware does when comparing floating-point numbers. But for pedestrian cases, simple approaches like this recipe generally work.

You can get a substantial speed boost by preallocating the list instead of calling `append` repeatedly. This also allows you to get rid of the conditionals in the inner loop. For one element, this version is barely faster, but with more than 10 elements it's consistently about 5 times faster—the kind of performance ratio that is worth caring about. I get identical output for every test case I can think of:

```
def frange2(start, end=None, inc=1.0):
    "A faster range-like function that does accept float increments..."
    if end == None:
        end = start + 0.0
        start = 0.0
    else: start += 0.0 # force it to be a float

    count = int((end - start) / inc)
    if start + count * inc != end:
        # Need to adjust the count. AFAICT, it always comes up one short.
        count += 1

    L = [start] * count
    for i in xrange(1, count):
        L[i] = start + i * inc

    return L
```

Both versions rely on a single multiplication and one addition to compute each item, to avoid accumulating error by repeated additions. This is why, for example, the body of the `for` loop in `frange2` is not:

```
L[i] = L[i-1] + inc
```

In Python 2.2, if all you need to do is loop on the result of `frange`, you can save some memory by turning this function into a simple generator, yielding an iterator when you call it:

```
from __future__ import generators

def frangei(start, end=None, inc=1.0):
    "An xrange-like simple generator that does accept float increments..."

    if end == None:
        end = start + 0.0
        start = 0.0
    assert inc                 # sanity check

    i = 0
    while 1:
        next = start + i * inc
        if inc > 0 and next >= end:
            break
        elif inc < 0 and next <= end:
            break
        yield next
        i += 1
```

If you use this recipe a lot, you should probably take a look at Numeric Python and other third-party packages that take computing with floating-point numbers seriously. This recipe, for example, will not scale well to very large ranges, while those defined in Numeric Python will.

See Also

Documentation for the `range` built-in function in the *Library Reference*; Numeric Python (*http://www.pfdubois.com/numpy/*).

1.16 Transposing Two-Dimensional Arrays

Credit: Steve Holden

Problem

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

Solution

You must start with a list whose items are lists all of the same length:

```
arr = [[1,2,3], [4,5,6], [7,8,9], [10,11,12]]
```

A list comprehension offers a simple, handy way to transpose it:

```
print [[r[col] for r in arr] for col in range(len(arr[0]))]
[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

Discussion

This recipe shows a concise way (although not necessarily the fastest way) to turn rows into columns. List comprehensions are known for being concise.

Sometimes data just comes at you the wrong way. For instance, if you use Microsoft's ADO database interface, due to array element ordering differences between Python and Microsoft's preferred implementation language (Visual Basic), the `GetRows` method actually appears to return database columns in Python, despite its name. This recipe's solution to this common problem was chosen to demonstrate nested list comprehensions.

Notice that the inner comprehension varies what is selected from (the row), while the outer comprehension varies the selector (the column). This process achieves the required transposition.

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.

See Also

The *Reference Manual* section on list displays (the other name for list comprehensions); Numeric Python (*http://www.pfdubois.com/numpy/*).

1.17 Creating Lists of Lists Without Sharing References

Credit: David Ascher

Problem

You want to create a multidimensional list, but the apparently simplest solution is fraught with surprises.

Solution

Use list comprehensions (also known as list displays) to avoid implicit reference sharing:

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

Discussion

When a newcomer to Python is shown the power of the multiplication operation on lists, he often gets quite excited about it, since it is such an elegant notation. For example:

```
>>> [0] * 5
[0, 0, 0, 0, 0]
```

The problem is that one-dimensional problems often grow a second dimension, so there is a natural progression to:

```
>>> multi = [[0] * 5] * 3
>>> print multi
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

This appears to have worked, but the same newcomer is then often puzzled by bugs, which typically can be boiled down to the following test:

```
>>> multi[0][0] = 'Changed!'
>>> print multi
[['Changed!', 0, 0, 0, 0], ['Changed!', 0, 0, 0, 0], ['Changed!', 0, 0, 0, 0]]
```

This problem definitely confuses most programmers at least once, if not a few times (see the FAQ entry at *http://www.python.org/doc/FAQ.html#4.50*). To understand it, it helps to decompose the creation of the multidimensional list into two steps:

```
>>> row = [0] * 5          # a list with five references to 0
>>> multi = [row] * 3      # a list with three references to the row object
```

The problem still exists in this version (Python is not that magical). The comments are key to understanding the source of the confusion. The process of multiplying a sequence by a number creates a new sequence with the specified number of new references to the original contents. In the case of the creation of `row`, it doesn't matter whether references are being duplicated or not, since the referent (the object being referred to) is immutable. In other words, there is no difference between an object and a reference to an object if that object is immutable. In the second line, however, what is created is a new list containing three references to the contents of the `[row]` list, which is a single reference to a list. Thus, `multi` contains three references to a single object. So when the first element of the first element of `multi` is changed, you are actually modifying the first element of the shared list. Hence the surprise.

List comprehensions, added in Python 2.2, provide a nice syntax that avoids the problem, as illustrated in the solution. With list comprehensions, there is no sharing of references—it's a truly nested computation. Note that the performance

characteristics of the solution are $O(M \times N)$, meaning that it will scale with each dimension. The list-multiplication idiom, however, is an $O(M)$ computation, as it doesn't really do duplications.

See Also

Documentation for the `range` built-in function in the *Library Reference*.