Intermediate and Advanced Software Carpentry Documentation

Release 1.0

C. Titus Brown

August 23, 2016

Author C Titus Brown

Date June 18, 2007

Welcome! You have stumbled upon the class handouts for a course I taught at Lawrence Livermore National Lab, June 12-June 14, 2007.

These notes are intended to *accompany* my lecture, which was a demonstration of a variety of "intermediate" Python features and packages. Because the demonstration was interactive, these notes are not complete notes of what went on in the course. (Sorry about that; they *have* been updated from my actual handouts to be more complete...)

However, all 70 pages are free to view and print, so enjoy.

All errors are, of course, my own. Note that almost all of the examples starting with '>>>' are doctests, so you can take the source and run doctest on it to make sure I'm being honest. But do me a favor and run the doctests with Python 2.5 ;).

Note that Day 1 of the course ran through the end of "Testing Your Software"; Day 2 ran through the end of "Online Resources for Python"; and Day 3 finished it off.

Example code (mostly from the C extension sections) is available here; see the README for more information.

Contents:

CHAPTER 1

Day 1

Contents:

1.1 Idiomatic Python

Extracts from The Zen of Python by Tim Peters:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Readability counts.

(The whole Zen is worth reading...)

The first step in programming is getting stuff to work at all.

The next step in programming is getting stuff to work regularly.

The step after that is reusing code and designing for reuse.

Somewhere in there you will start writing idiomatic Python.

Idiomatic Python is what you write when the *only* thing you're struggling with is the right way to solve *your* problem, and you're not struggling with the programming language or some weird library error or a nasty data retrieval issue or something else extraneous to your real problem. The idioms you prefer may differ from the idioms I prefer, but with Python there will be a fair amount of overlap, because there is usually at most one obvious way to do every task. (A caveat: "obvious" is unfortunately the eye of the beholder, to some extent.)

For example, let's consider the right way to keep track of the item number while iterating over a list. So, given a list z,

```
>>> z = [ 'a', 'b', 'c', 'd' ]
```

let's try printing out each item along with its index.

You could use a while loop:

```
>>> i = 0
>>> while i < len(z):
...     print i, z[i]
...     i += 1
0 a
1 b
```

```
2 c
3 d
```

or a for loop:

```
>>> for i in range(0, len(z)):
...     print i, z[i]
0 a
1 b
2 c
3 d
```

but I think the clearest option is to use `enumerate`:

```
>>> for i, item in enumerate(z):
...     print i, item
0 a
1 b
2 c
3 d
```

Why is this the clearest option? Well, look at the ZenOfPython extract above: it's explicit (we used `enumerate`); it's simple; it's readable; and I would even argue that it's prettier than the while loop, if not exactly "beatiful".

Python provides this kind of simplicity in as many places as possible, too. Consider file handles; did you know that they were iterable?

```
>>> for line in file('data/listfile.txt'):
...     print line.rstrip()
a
b
c
d
```

Where Python really shines is that this kind of simple idiom – in this case, iterables – is very very easy not only to use but to *construct* in your own code. This will make your own code much more reusable, while improving code readability dramatically. And that's the sort of benefit you will get from writing idiomatic Python.

1.1.1 Some basic data types

I'm sure you're all familiar with tuples, lists, and dictionaries, right? Let's do a quick tour nonetheless.

'tuples' are all over the place. For example, this code for swapping two numbers implicitly uses tuples:

```
>>> a = 5
>>> b = 6
>>> a, b = b, a
>>> print a == 6, b == 5
True True
```

That's about all I have to say about tuples.

I use lists and dictionaries *all the time*. They're the two greatest inventions of mankind, at least as far as Python goes. With lists, it's just easy to keep track of stuff:

```
>>> x = []
>>> x.append(5)
>>> x.extend([6, 7, 8])
>>> x
[5, 6, 7, 8]
```

```
>>> x.reverse()
>>> x
[8, 7, 6, 5]
```

It's also easy to sort. Consider this set of data:

```
>>> y = [ ('IBM', 5), ('Zil', 3), ('DEC', 18) ]
```

The `sort` method will run `cmp` on each of the tuples, which sort on the first element of each tuple:

```
>>> y.sort()
>>> y
[('DEC', 18), ('IBM', 5), ('Zil', 3)]
```

Often it's handy to sort tuples on a different tuple element, and there are several ways to do that. I prefer to provide my own sort method:

```
>>> def sort_on_second(a, b):
...   return cmp(a[1], b[1])
```

```
>>> y.sort(sort_on_second)
>>> y
[('Zil', 3), ('IBM', 5), ('DEC', 18)]
```

Note that here I'm using the builtin `cmp` method (which is what `sort` uses by default: `y.sort()` is equivalent to `y.sort(cmp)`) to do the comparison of the second part of the tuple.

This kind of function is really handy for sorting dictionaries by value, as I'll show you below.

(For a more in-depth discussion of sorting options, check out the Sorting HowTo.)

On to dictionaries!

Your basic dictionary is just a hash table that takes keys and returns values:

```
>>> d = {}
>>> d['a'] = 5
>>> d['b'] = 4
>>> d['c'] = 18
>>> d
{'a': 5, 'c': 18, 'b': 4}
>>> d['a']
5
```

You can also initialize a dictionary using the `dict` type to create a dict object:

```
>>> e = dict(a=5, b=4, c=18)
>>> e
{'a': 5, 'c': 18, 'b': 4}
```

Dictionaries have a few really neat features that I use pretty frequently. For example, let's collect (key, value) pairs where we potentially have multiple values for each key. That is, given a file containing this data,

```
a 5
b 6
d 7
a 2
c 1
```

suppose we want to keep all the values? If we just did it the simple way,

```
>>> d = {}
>>> for line in file('data/keyvalue.txt'):
...   key, value = line.split()
...   d[key] = int(value)
```

we would lose all but the last value for each key:

```
>>> d
{'a': 2, 'c': 1, 'b': 6, 'd': 7}
```

You can collect *all* the values by using `get`:

```
>>> d = {}
>>> for line in file('data/keyvalue.txt'):
...   key, value = line.split()
...   l = d.get(key, [])
...   l.append(int(value))
...   d[key] = l
>>> d
{'a': [5, 2], 'c': [1], 'b': [6], 'd': [7]}
```

The key point here is that `d.get(k, default)` is equivalent to `d[k]` if `d[k]` already exists; otherwise, it returns `default`. So, the first time each key is used, `l` is set to an empty list; the value is appended to this list, and then the value is set for that key.

(There are tons of little tricks like the ones above, but these are the ones I use the most; see the Python Cookbook for an endless supply!)

Now let's try combining some of the sorting stuff above with dictionaries. This time, our contrived problem is that we'd like to sort the keys in the dictionary `d` that we just loaded, but rather than sorting by key we want to sort by the sum of the values for each key.

First, let's define a sort function:

```
>>> def sort_by_sum_value(a, b):
...     sum_a = sum(a[1])
...     sum_b = sum(b[1])
...     return cmp(sum_a, sum_b)
```

Now apply it to the dictionary items:

```
>>> items = d.items()
>>> items
[('a', [5, 2]), ('c', [1]), ('b', [6]), ('d', [7])]
>>> items.sort(sort_by_sum_value)
>>> items
[('c', [1]), ('b', [6]), ('a', [5, 2]), ('d', [7])]
```

and voila, you have your list of keys sorted by summed values!

As I said, there are tons and tons of cute little tricks that you can do with dictionaries. I think they're incredibly powerful.

1.1.2 List comprehensions

List comprehensions are neat little constructs that will shorten your lines of code considerably. Here's an example that constructs a list of squares between 0 and 4:

```
>>> z = [ i**2 for i in range(0, 5) ]
>>> z
[0, 1, 4, 9, 16]
```

You can also add in conditionals, like requiring only even numbers:

```
>>> z = [ i**2 for i in range(0, 10) if i % 2 == 0 ]
>>> z
[0, 4, 16, 36, 64]
```

The general form is

```
[ expression for var in list if conditional ]
```

so pretty much anything you want can go in `expression` and `conditional`.

I find list comprehensions to be very useful for both file parsing and for simple math. Consider a file containing data and comments:

```
# this is a comment or a header
1
# another comment
2
```

where you want to read in the numbers only:

```
>>> data = [ int(x) for x in open('data/commented-data.txt') if x[0] != '#' ]
>>> data
[1, 2]
```

This is short, simple, and very explicit!

For simple math, suppose you need to calculate the average and stddev of some numbers. Just use a list comprehension:

```
>>> import math
>>> data = [ 1, 2, 3, 4, 5 ]
>>> average = sum(data) / float(len(data))
>>> stddev = sum([ (x - average)**2 for x in data ]) / float(len(data))
>>> stddev = math.sqrt(stddev)
>>> print average, '+/-', stddev
3.0 +/- 1.41421356237
```

Oh, and one rule of thumb: if your list comprehension is longer than one line, change it to a for loop; it will be easier to read, and easier to understand.

1.1.3 Building your own types

Most people should be pretty familiar with basic classes.

```
>>> class A:
...   def __init__(self, item):
...       self.item = item
...   def hello(self):
...       print 'hello,', self.item
```

```
>>> x = A('world')
>>> x.hello()
hello, world
```

There are a bunch of neat things you can do with classes, but one of the neatest is building new types that can be used with standard Python list/dictionary idioms.

For example, let's consider a basic binning class.

```
>>> class Binner:
...   def __init__(self, binwidth, binmax):
...     self.binwidth, self.binmax = binwidth, binmax
...     nbins = int(binmax / float(binwidth) + 1)
...     self.bins = [0] * nbins
...
...   def add(self, value):
...     bin = value / self.binwidth
...     self.bins[bin] += 1
```

This behaves as you'd expect:

```
>>> binner = Binner(5, 20)
>>> for i in range(0,20):
...   binner.add(i)
>>> binner.bins
[5, 5, 5, 5, 0]
```

...but wouldn't it be nice to be able to write this?

```
for i in range(0, len(binner)):
   print i, binner[i]
```

or even this?

```
for i, bin in enumerate(binner):
   print i, bin
```

This is actually quite easy, if you make the `Binner` class look like a list by adding two special functions:

```
>>> class Binner:
...   def __init__(self, binwidth, binmax):
...     self.binwidth, self.binmax = binwidth, binmax
...     nbins = int(binmax / float(binwidth) + 1)
...     self.bins = [0] * nbins
...
...   def add(self, value):
...     bin = value / self.binwidth
...     self.bins[bin] += 1
...
...   def __getitem__(self, index):
...     return self.bins[index]
...
...   def __len__(self):
...     return len(self.bins)
```

```
>>> binner = Binner(5, 20)
>>> for i in range(0,20):
...   binner.add(i)
```

and now we can treat `Binner` objects as normal lists:

```
>>> for i in range(0, len(binner)):
...   print i, binner[i]
0 5
1 5
```

```
2 5
3 5
4 0
```

```
>>> for n in binner:
...   print n
5
5
5
5
0
```

In the case of `len(binner)`, Python knows to use the special method `__len__`, and likewise `binner[i]` just calls `__getitem__(i)`.

The second case involves a bit more implicit magic. Here, Python figures out that `Binner` can act like a list and simply calls the right functions to retrieve the information.

Note that making your own read-only dictionaries is pretty simple, too: just provide the `__getitem__` function, which is called for non-integer values as well:

```
>>> class SillyDict:
...    def __getitem__(self, key):
...        print 'key is', key
...        return key
>>> sd = SillyDict()
>>> x = sd['hello, world']
key is hello, world
>>> x
'hello, world'
```

You can also write your own mutable types, e.g.

```
>>> class SillyDict:
...   def __setitem__(self, key, value):
...       print 'setting', key, 'to', value
>>> sd = SillyDict()
>>> sd[5] = 'world'
setting 5 to world
```

but I have found this to be less useful in my own code, where I'm usually writing special objects like the `Binner` type above: I prefer to specify my own methods for putting information *into* the object type, because it reminds me that it is not a generic Python list or dictionary. However, the use of `__getitem__` (and some of the iterator and generator features I discuss below) can make code *much* more readable, and so I use them whenever I think the meaning will be unambiguous. For example, with the `Binner` type, the purpose of `__getitem__` and `__len__` is not very ambiguous, while the purpose of a `__setitem__` function (to support `binner[x] = y`) would be unclear.

Overall, the creation of your own custom list and dict types is one way to make reusable code that will fit nicely into Python's natural idioms. In turn, this can make your code look much simpler and feel much cleaner. The risk, of course, is that you will also make your code harder to understand and (if you're not careful) harder to debug. Mediating between these options is mostly a matter of experience.

1.1.4 Iterators

Iterators are another built-in Python feature; unlike the list and dict types we discussed above, an iterator isn't really a *type*, but a *protocol*. This just means that Python agrees to respect anything that supports a particular set of methods as if it were an iterator. (These protocols appear everywhere in Python; we were taking advantage of the mapping and sequence protocols above, when we defined `__getitem__` and `__len__`, respectively.)

Iterators are more general versions of the sequence protocol; here's an example:

```
>>> class SillyIter:
...    i = 0
...    n = 5
...    def __iter__(self):
...       return self
...    def next(self):
...       self.i += 1
...       if self.i > self.n:
...          raise StopIteration
...       return self.i
```

```
>>> si = SillyIter()
>>> for i in si:
...    print i
1
2
3
4
5
```

Here, `__iter__` just returns `self`, an object that has the function `next()`, which (when called) either returns a value or raises a StopIteration exception.

We've actually already met several iterators in disguise; in particular, `enumerate` is an iterator. To drive home the point, here's a simple reimplementation of `enumerate`:

```
>>> class my_enumerate:
...    def __init__(self, some_iter):
...       self.some_iter = iter(some_iter)
...       self.count = -1
...
...    def __iter__(self):
...       return self
...
...    def next(self):
...       val = self.some_iter.next()
...       self.count += 1
...       return self.count, val
>>> for n, val in my_enumerate(['a', 'b', 'c']):
...    print n, val
0 a
1 b
2 c
```

You can also iterate through an iterator the "old-fashioned" way:

```
>>> some_iter = iter(['a', 'b', 'c'])
>>> while 1:
...    try:
...       print some_iter.next()
...    except StopIteration:
...       break
a
b
c
```

but that would be silly in most situations! I use this if I just want to get the first value or two from an iterator.

With iterators, one thing to watch out for is the return of `self` from the `__iter__` function. You can all too easily write an iterator that isn't as re-usable as you think it is. For example, suppose you had the following class:

```
>>> class MyTrickyIter:
...   def __init__(self, thelist):
...      self.thelist = thelist
...      self.index = -1
...
...   def __iter__(self):
...      return self
...
...   def next(self):
...      self.index += 1
...      if self.index < len(self.thelist):
...         return self.thelist[self.index]
...      raise StopIteration
```

This works just like you'd expect as long as you create a new object each time:

```
>>> for i in MyTrickyIter(['a', 'b']):
...   for j in MyTrickyIter(['a', 'b']):
...      print i, j
a a
a b
b a
b b
```

but it will break if you create the object just once:

```
>>> mi = MyTrickyIter(['a', 'b'])
>>> for i in mi:
...   for j in mi:
...      print i, j
a b
```

because self.index is incremented in each loop.

1.1.5 Generators

Generators are a Python implementation of coroutines. Essentially, they're functions that let you suspend execution and return a result:

```
>>> def g():
...   for i in range(0, 5):
...      yield i**2
>>> for i in g():
...     print i
0
1
4
9
16
```

You could do this with a list just as easily, of course:

```
>>> def h():
...   return [ x ** 2 for x in range(0, 5) ]
>>> for i in h():
...     print i
0
```

```
1
4
9
16
```

But you can do things with generators that you couldn't do with finite lists. Consider two full implementation of Eratosthenes' Sieve for finding prime numbers, below.

First, let's define some boilerplate code that can be used by either implementation:

```
>>> def divides(primes, n):
...   for trial in primes:
...      if n % trial == 0: return True
...   return False
```

Now, let's write a simple sieve with a generator:

```
>>> def prime_sieve():
...    p, current = [], 1
...    while 1:
...        current += 1
...        if not divides(p, current): # if any previous primes divide, cancel
...            p.append(current)           # this is prime! save & return
...            yield current
```

This implementation will find (within the limitations of Python's math functions) all prime numbers; the programmer has to stop it herself:

```
>>> for i in prime_sieve():
...    print i
...    if i > 10:
...        break
2
3
5
7
11
```

So, here we're using a generator to implement the generation of an infinite series with a single function definition. To do the equivalent with an iterator would require a class, so that the object instance can hold the variables:

```
>>> class iterator_sieve:
...    def __init__(self):
...       self.p, self.current = [], 1
...    def __iter__(self):
...       return self
...    def next(self):
...       while 1:
...          self.current = self.current + 1
...          if not divides(self.p, self.current):
...             self.p.append(self.current)
...             return self.current
```

```
>>> for i in iterator_sieve():
...    print i
...    if i > 10:
...        break
2
3
5
```

```
7
11
```

It is also *much* easier to write routines like `enumerate` as a generator than as an iterator:

```
>>> def gen_enumerate(some_iter):
...    count = 0
...    for val in some_iter:
...       yield count, val
...       count += 1
```

```
>>> for n, val in gen_enumerate(['a', 'b', 'c']):
...    print n, val
0 a
1 b
2 c
```

Abstruse note: we don't even have to catch `StopIteration` here, because the for loop simply ends when `some_iter` is done!

1.1.6 assert

One of the most underused keywords in Python is `assert`. Assert is pretty simple: it takes a boolean, and if the boolean evaluates to False, it fails (by raising an AssertionError exception). `assert True` is a no-op.

```
>>> assert True
>>> assert False
Traceback (most recent call last):
   ...
AssertionError
```

You can also put an optional message in:

```
>>> assert False, "you can't do that here!"
Traceback (most recent call last):
   ...
AssertionError: you can't do that here!
```

`assert` is very, very useful for making sure that code is behaving according to your expectations during development. Worried that you're getting an empty list? `assert len(x)`. Want to make sure that a particular return value is not None? `assert retval is not None`.

Also note that 'assert' statements are removed from optimized code, so only use them to conditions related to actual development, and make sure that the statement you're evaluating has no side effects. For example,

```
>>> a = 1
>>> def check_something():
...    global a
...    a = 5
...    return True
>>> assert check_something()
```

will behave differently when run under optimization than when run without optimization, because the `assert` line will be removed completely from optimized code.

If you need to raise an exception in production code, see below. The quickest and dirtiest way is to just "raise Exception", but that's kind of non-specific ;).

1.1.7 Conclusions

Use of common Python idioms – both in your python code and for your new types – leads to short, sweet programs.

1.2 Structuring, Testing, and Maintaining Python Programs

Python is really the first programming language in which I started re-using code significantly. In part, this is because it is rather easy to compartmentalize functions and classes in Python. Something else that Python makes relatively easy is building testing into your program structure. Combined, reusability and testing can have a huge effect on maintenance.

1.2.1 Programming for reusability

It's difficult to come up with any hard and fast rules for programming for reusability, but my main rules of thumb are: don't plan too much, and don't hesitate to refactor your code. [1].

In any project, you will write code that you want to re-use in a slightly different context. It will often be easiest to cut and paste this code rather than to copy the module it's in – but try to resist this temptation a bit, and see if you can make the code work for both uses, and then use it in both places.

1.2.2 Modules and scripts

The organization of your code source files can help or hurt you with code re-use.

Most people start their Python programming out by putting everything in a script:

```
calc-squares.py:
  #! /usr/bin/env python
  for i in range(0, 10):
     print i**2
```

This is great for experimenting, but you can't re-use this code at all!

(UNIX folk: note the use of `#!  /usr/bin/env python`, which tells UNIX to execute this script using whatever `python` program is first in your path. This is more portable than putting `#!  /usr/local/bin/python` or `#! /usr/bin/python` in your code, because not everyone puts python in the same place.)

Back to reuse. What about this?

```
calc-squares.py:
  #! /usr/bin/env python
  def squares(start, stop):
     for i in range(0, 10):
        print i**2

  squares(0, 10)
```

I think that's a bit better for re-use – you've made `squares` flexible and re-usable – but there are two mechanistic problems. First, it's named `calc-squares.py`, which means it can't readily be imported. (Import filenames have to be valid Python names, of course!) And, second, were it importable, it would execute `squares(0, 10)` on import - hardly what you want!

To fix the first, just change the name:

[1] If you haven't read Martin Fowler's **Refactoring**, do so – it describes how to incrementally make your code better. I'll discuss it some more in the context of testing, below.

```
calc_squares.py:
  #! /usr/bin/env python
  def squares(start, stop):
    for i in range(0, 10):
        print i**2

  squares(0, 10)
```

Good, but now if you do `import calc_squares`, the `squares(0, 10)` code will still get run! There are a couple of ways to deal with this. The first is to look at the module name: if it's `calc_squares`, then the module is being imported, while if it's `__main__`, then the module is being run as a script:

```
calc_squares.py:
  #! /usr/bin/env python
  def squares(start, stop):
    for i in range(0, 10):
       print i**2

  if __name__ == '__main__':
    squares(0, 10)
```

Now, if you run `calc_squares.py` directly, it will run `squares(0, 10)`; if you import it, it will simply define the `squares` function and leave it at that. This is probably the most standard way of doing it.

I actually prefer a different technique, because of my fondness for testing. (I also think this technique lends itself to reusability, though.) I would actually write two files:

```
squares.py:
  def squares(start, stop):
    for i in range(0, 10):
       print i**2

  if __name__ == `__main__`:
    # ...run automated tests...

calc-squares:
  #! /usr/bin/env python
  import squares
  squares.squares(0, 10)
```

A few notes – first, this is eminently reusable code, because `squares.py` is completely separate from the context-specific call. Second, you can look at the directory listing in an instant and see that `squares.py` is probably a library, while `calc-squares` must be a script, because the latter cannot be imported. Third, you can add automated tests to `squares.py` (as described below), and run them simply by running `python squares.py`. Fourth, you can add script-specific code such as command-line argument handling to the script, and keep it separate from your data handling and algorithm code.

1.2.3 Packages

A Python package is a directory full of Python modules containing a special file, `__init__.py`, that tells Python that the directory is a package. Packages are for collections of library code that are too big to fit into single files, or that have some logical substructure (e.g. a central library along with various utility functions that all interact with the central library).

For an example, look at this directory tree:

```
package/
  __init__.py         -- contains functions a(), b()
  other.py            -- contains function c()
  subdir/
     __init__.py      -- contains function d()
```

From this directory tree, you would be able to access the functions like so:

```
import package
package.a()
package.b()

import package.other
package.other.c()

import package.subdir
package.subdir.d()
```

Note that `__init__.py` is just another Python file; there's nothing special about it except for the name, which tells Python that the directory is a package directory. `__init__.py` is the only code executed on import, so if you want names and symbols from other modules to be accessible at the package top level, you have to import or create them in `__init__.py`.

There are two ways to use packages: you can treat them as a convenient code organization technique, and make most of the functions or classes available at the top level; or you can use them as a library hierarchy. In the first case you would make all of the names above available at the top level:

```
package/__init__.py:
  from other import c
  from subdir import d
  ...
```

which would let you do this:

```
import package
package.a()
package.b()
package.c()
package.d()
```

That is, the names of the functions would all be immediately available at the top level of the package, but the implementations would be spread out among the different files and directories. I personally prefer this because I don't have to remember as much ;). The down side is that everything gets imported all at once, which (especially for large bodies of code) may be slow and memory intensive if you only need a few of the functions.

Alternatively, if you wanted to keep the library hierarchy, just leave out the top-level imports. The advantage here is that you only import the names you need; however, you need to remember more.

Some people are fond of package trees, but I've found that hierarchies of packages more than two deep are annoying to develop on: you spend a lot of your time browsing around between directories, trying to figure out *exactly* which function you need to use and what it's named. (Your mileage may vary.) I think this is one of the main reasons why the Python stdlib looks so big, because most of the packages are top-level.

One final note: you can restrict what objects are exported from a module or package by listing the names in the `__all__` variable. So, if you had a module `some_mod.py` that contained this code:

```
some_mod.py:
  __all__ = ['fn1']

  def fn1(...):
      ...

  def fn2(...):
      ...
```

then only 'some_mod.fn1()' would be available on import. This is a good way to cut down on "namespace pollution" – the presence of "private" objects and code in imported modules – which in turn makes introspection useful.

1.2.4 A short digression: naming and formatting

You may have noticed that a lot of Python code looks pretty similar – this is because there's an "official" style guide for Python, called PEP 8. It's worth a quick skim, and an occasional deeper read for some sections.

Here are a few tips that will make your code look internally consistent, if you don't already have a coding style of your own:

- use four spaces (NOT a tab) for each indentation level;
- **use lowercase, _-separated names for module and function names, e.g.** `my_module`;
- use CapsWord style to name classes, e.g. `MySpecialClass`;
- **use '_'-prefixed names to indicate a "private" variable that should** not be used outside this module, , e.g. `_some_private_variable`;

1.2.5 Another short digression: docstrings

Docstrings are strings of text attached to Python objects like modules, classes, and methods/functions. They can be used to provide human-readable help when building a library of code. "Good" docstring coding is used to provide additional information about functionality beyond what can be discovered automatically by introspection; compare

```
def is_prime(x):
    """
    is_prime(x) -> true/false.  Determines whether or not x is prime,
    and return true or false.
    """
```

versus

```
def is_prime(x):
    """
    Returns true if x is prime, false otherwise.

    is_prime() uses the Bernoulli-Schmidt formalism for figuring out
    if x is prime.  Because the BS form is stochastic and hysteretic,
    multiple calls to this function will be increasingly accurate.
    """
```

The top example is good (documentation is good!), but the bottom example is better, for a few reasons. First, it is not redundant: the arguments to `is_prime` are discoverable by introspection and don't need to be specified. Second, it's summarizable: the first line stands on its own, and people who are interested in more detail can read on. This enables certain document extraction tools to do a better job.

For more on docstrings, see PEP 257.

1.2.6 Sharing data between code

There are three levels at which data can be shared between Python code: module globals, class attributes, and object attributes. You can also sneak data into functions by dynamically defining a function within another scope, and/or binding them to keyword arguments.

1.2.7 Scoping: a digression

Just to make sure we're clear on scoping, here are a few simple examples. In this first example, f() gets x from the module namespace.

```
>>> x = 1
>>> def f():
...   print x
>>> f()
1
```

In this second example, f() overrides x, but only within the namespace in f().

```
>>> x = 1
>>> def f():
...   x = 2
...   print x
>>> f()
2
>>> print x
1
```

In this third example, g() overrides x, and h() obtains x from within g(), because h() was *defined* within g():

```
>>> x = 1
```

```
>>> def outer():
...    x = 2
...
...    def inner():
...       print x
...
...    return inner
```

```
>>> inner = outer()
>>> inner()
2
```

In all cases, without a `global` declaration, assignments will simply create a new local variable of that name, and not modify the value in any other scope:

```
>>> x = 1
>>> def outer():
...    x = 2
...
...    def inner():
...       x = 3
...
...    inner()
...
...    print x
>>> outer()
2
```

However, *with* a `global` definition, the outermost scope is used:

```
>>> x = 1
>>> def outer():
...     x = 2
...
...     def inner():
...         global x
...         x = 3
...
...     inner()
...
...     print x
>>> outer()
2
>>> print x
3
```

I generally suggest avoiding scope trickery as much as possible, in the interests of readability. There are two common patterns that I use when I *have* to deal with scope issues.

First, module globals are sometimes necessary. For one such case, imagine that you have a centralized resource that you must initialize precisely once, and you have a number of functions that depend on that resource. Then you can use a module global to keep track of the initialization state. Here's a (contrived!) example for a random number generator that initializes the random number seed precisely once:

```
_initialized = False
def init():
  global _initialized
  if not _initialized:
      import time
      random.seed(time.time())
      _initialized = True

def randint(start, stop):
  init()
  ...
```

This code ensures that the random number seed is initialized only once by making use of the `_initialized` module global. A few points, however:

- this code is not threadsafe. If it was really important that the resource be initialized precisely once, you'd need to use thread locking. Otherwise two functions could call `randint()` at the same time and both could get past the `if` statement.
- the module global code is very isolated and its use is very clear. Generally I recommend having only one or two functions that access the module global, so that if I need to change its use I don't have to understand a lot of code.

The other "scope trickery" that I sometimes engage in is passing data into dynamically generated functions. Consider a situation where you have to use a callback API: that is, someone has given you a library function that will call your own code in certain situations. For our example, let's look at the `re.sub` function that comes with Python, which takes a callback function to apply to each match.

Here's a callback function that uppercases words:

```
>>> def replace(m):
...   match = m.group()
...   print 'replace is processing:', match
...   return match.upper()
>>> s = "some string"
```

```
>>> import re
>>> print re.sub('\\S+', replace, s)
replace is processing: some
replace is processing: string
SOME STRING
```

What's happening here is that the `replace` function is called each time the regular expression '\S+' (a set of non-whitespace characters) is matched. The matching substring is replaced by whatever the function returns.

Now let's imagine a situation where we want to pass information into `replace`; for example, we want to process only words that match in a dictionary. (I *told* you it was contrived!) We could simply rely on scoping:

```
>>> d = { 'some' : True, 'string' : False }
>>> def replace(m):
...   match = m.group()
...   if match in d and d[match]:
...      return match.upper()
...   return match
```

```
>>> print re.sub('\\S+', replace, s)
SOME string
```

but I would argue against it on the grounds of readability: passing information implicitly between scopes is bad. (At this point advanced Pythoneers might sneer at me, because scoping is natural to Python, but nuts to them: readability and transparency is also very important.) You *could* also do it this way:

```
>>> d = { 'some' : True, 'string' : False }
>>> def replace(m, replace_dict=d):             # <-- explicit declaration
...   match = m.group()
...   if match in replace_dict and replace_dict[match]:
...      return match.upper()
...   return match
```

```
>>> print re.sub('\\S+', replace, s)
SOME string
```

The idea is to use keyword arguments on the function to pass in required information, thus making the information passing explicit.

1.2.8 Back to sharing data

I started discussing scope in the context of sharing data, but we got a bit sidetracked from data sharing. Let's get back to that now.

The key to thinking about data sharing in the context of code reuse is to think about how that data will be used.

If you use a module global, then any code in that module has access to that global.

If you use a class attribute, then any object of that class type (including inherited classes) shares that data.

And, if you use an object attribute, then every object of that class type will have its own version of that data.

How do you choose which one to use? My ground rule is to minimize the use of more widely shared data. If it's possible to use an object variable, do so; otherwise, use either a module or class attribute. (In practice I almost never use class attributes, and infrequently use module globals.)

1.2.9 How modules are loaded (and when code is executed)

Something that has been implicit in the discussion of scope and data sharing, above, is the order in which module code is executed. There shouldn't be any surprises here if you've been using Python for a while, so I'll be brief: in general, the code at the top level of a module is executed at *first* import, and all other code is executed in the order you specify when you start calling functions or methods.

Note that because the top level of a module is executed precisely once, at *first* import, the following code prints "hello, world" only once:

```
mod_a.py:
  def f():
     print 'hello, world'

  f()

mod_b.py:
  import mod_a
```

The `reload` function will reload the module and force re-execution at the top level:

```
reload(sys.modules['mod_a'])
```

It is also worth noting that the module name is bound to the local namespace *prior* to the execution of the code in the module, so not all symbols in the module are immediately available. This really only impacts you if you have interdependencies between modules: for example, this will work if `mod_a` is imported before `mod_b`:

```
mod_a.py:
  import mod_b

mod_b.py:
  import mod_a
```

while this will not:

```
mod_a.py:
  import mod_b
  x = 5

mod_b.py:
  import mod_a
  y = mod_a.x
```

To see why, let's put in some print statements:

```
mod_a.py:
  print 'at top of mod_a'
  import mod_b
  print 'mod_a: defining x'
  x = 5

mod_b.py:
  print 'at top of mod_b'
  import mod_a
  print 'mod_b: defining y'
  y = mod_a.x
```

Now try `import mod_a` and `import mod_b`, each time in a new interpreter:

```
>> import mod_a
at top of mod_a
at top of mod_b
mod_b: defining y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mod_a.py", line 2, in <module>
    import mod_b
  File "mod_b.py", line 4, in <module>
    y = mod_a.x
AttributeError: 'module' object has no attribute 'x'

>> import mod_b
at top of mod_b
at top of mod_a
mod_a: defining x
mod_b: defining y
```

1.2.10 PYTHONPATH, and finding packages & modules during development

So, you've got your re-usable code nicely defined in modules, and now you want to ... use it. How can you import code from multiple locations?

The simplest way is to set the PYTHONPATH environment variable to contain a list of directories from which you want to import code; e.g. in UNIX bash,

```
% export PYTHONPATH=/path/to/directory/one:/path/to/directory/two
```

or in csh,

```
% setenv PYTHONPATH /path/to/directory/one:/path/to/directory/two
```

Under Windows,

```
> set PYTHONPATH directory1;directory2
```

should work.

However, setting the PYTHONPATH explicitly can make your code less movable in practice, because you will forget (and fail to document) the modules and packages that your code depends on. I prefer to modify sys.path directly:

```python
import sys
sys.path.insert(0, '/path/to/directory/one')
sys.path.insert(0, '/path/to/directory/two')
```

which has the advantage that you are explicitly specifying the location of packages that you depend upon in the dependent code.

Note also that you can put modules and packages in zip files and Python will be able to import directly from the zip file; just place the path to the zip file in either `sys.path` or your PYTHONPATH.

Now, I tend to organize my projects into several directories, with a `bin/` directory that contains my scripts, and a `lib/` directory that contains modules and packages. If I want to to deploy this code in multiple locations, I can't rely on inserting absolute paths into sys.path; instead, I want to use relative paths. Here's the trick I use

In my script directory, I write a file `_mypath.py`.

```
_mypath.py:
   import os, sys
   thisdir = os.path.dirname(__file__)
   libdir = os.path.join(thisdir, '../relative/path/to/lib/from/bin')

   if libdir not in sys.path:
      sys.path.insert(0, libdir)
```

Now, in each script I put `import _mypath` at the top of the script. When running scripts, Python automatically enters the script's directory into sys.path, so the script can import _mypath. Then _mypath uses the special attribute __file__ to calculate its own location, from which it can calculate the absolute path to the library directory and insert the library directory into `sys.path`.

1.2.11 setup.py and distutils: the old fashioned way of installing Python packages

While developing code, it's easy to simply work out of the development directory. However, if you want to pass the code onto others as a finished module, or provide it to systems admins, you might want to consider writing a `setup.py` file that can be used to install your code in a more standard way. setup.py lets you use distutils to install the software by running

```
python setup.py install
```

Writing a setup.py is simple, especially if your package is pure Python and doesn't include any extension files. A setup.py file for a pure Python install looks like this:

```
from distutils.core import setup
setup(name='your_package_name',
      py_modules = ['module1', 'module2']
      packages = ['package1', 'package2']
      scripts = ['script1', 'script2'])
```

One this script is written, just drop it into the top-level directory and type `python setup.py build`. This will make sure that distutils can find all the files.

Once your setup.py works for building, you can package up the entire directory with tar or zip and anyone should be able to install it by unpacking the package and typing

```
% python setup.py install
```

This will copy the packages and modules into Python's `site-packages` directory, and install the scripts into Python's script directory.

1.2.12 setup.py, eggs, and easy_install: the new fangled way of installing Python packages

A somewhat newer (and better) way of distributing Python software is to use easy_install, a system developed by Phillip Eby as part of the setuptools package. Many of the capabilities of easy_install/setuptools are probably unnecessary for scientific Python developers (although it's an excellent way to install Python packages from other sources), so I will focus on three capabilities that I think are most useful for "in-house" development: versioning, user installs, and binary eggs.

First, install easy_install/setuptools. You can do this by downloading

```
http://peak.telecommunity.com/dist/ez_setup.py
```

and running `python ez_setup.py`. (If you can't do this as the superuser, see the note below about user installs.) Once you've installed setuptools, you should be able to run the script `easy_install`.

The first thing this lets you do is easily install any software that is distutils-compatible. You can do this from a number of sources: from an unpackaged directory (as with `python setup.py install`); from a tar or zip file; from the project's URL or Web page; from an egg (see below); or from PyPI, the Python Package Index (see http://cheeseshop.python.org/pypi/).

Let's try installing `nose`, a unit test discovery package we'll be looking at in the testing section (below). Type:

```
easy_install --install-dir=~/.packages nose
```

This will go to the Python Package Index, find the URL for nose, download it, and install it in your ~/.packages directory. We're specifying an install-dir so that you can install it for your use only; if you were the superuser, you could install it for everyone by omitting '–install-dir'.

(Note that you need to add ~/.packages to your PATH and your PYTHONPATH, something I've already done for you.)

So, now, you can go do 'import nose' and it will work. Neat, eh? Moreover, the nose-related scripts (`nosetests`, in this case) have been installed for your use as well.

You can also install specific versions of software; right now, the latest version of nose is 0.9.3, but if you wanted 0.9.2, you could specify `easy_install nose==0.9.2` and it would do its best to find it.

This leads to the next setuptools feature of note, `pkg_resource.require`. `pkg_resources.require` lets you specify that certain packages must be installed. Let's try it out by requiring that CherryPy 3.0 or later is installed:

```
>> import pkg_resources
>> pkg_resources.require('CherryPy >= 3.0')
Traceback (most recent call last):
     ...
DistributionNotFound: CherryPy >= 3.0
```

OK, so that failed... but now let's install CherryPy:

```
% easy_install --install-dir=~/.packages CherryPy
```

Now the require will work:

```
>> pkg_resources.require('CherryPy >= 3.0')
>> import CherryPy
```

This version requirement capability is quite powerful, because it lets you specify exactly the versions of the software you need for your own code to work. And, if you need multiple versions of something installed, setuptools lets you do that, too – see the `--multi-version` flag for more information. While you still can't use *different* versions of the same package in the same program, at least you can have multiple versions of the same package installed!

Throughout this, we've been using another great feature of setuptools: user installs. By specifying the `--install-dir`, you can install most Python packages for yourself, which lets you take advantage of easy_install's capabilities without being the superuser on your development machine.

This brings us to the last feature of setuptools that I want to mention: eggs, and in particular binary eggs. We'll explore binary eggs later; for now let me just say that easy_install makes it possible for you to package up multiple binary versions of your software (*with* extension modules) so that people don't have to compile it themselves. This is an invaluable and somewhat underutilized feature of easy_install, but it can make life much easier for your users.

1.3 Testing Your Software

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian W. Kernighan.

Everyone tests their software to some extent, if only by running it and trying it out (technically known as "smoke testing"). Most programmers do a certain amount of exploratory testing, which involves running through various functional paths in your code and seeing if they work.

Systematic testing, however, is a different matter. Systematic testing simply cannot be done properly without a certain (large!) amount of automation, because every change to the software means that the software needs to be tested all over again.

Below, I will introduce you to some lower level automated testing concepts, and show you how to use built-in Python constructs to start writing tests.

1.3.1 An introduction to testing concepts

There are several types of tests that are particularly useful to research programmers. *Unit tests* are tests for fairly small and specific units of functionality. *Functional tests* test entire functional paths through your code. *Regression tests* make sure that (within the resolution of your records) your program's output has not changed.

All three types of tests are necessary in different ways.

Regression tests tell you when unexpected changes in behavior occur, and can reassure you that your basic data processing is still working. For scientists, this is particularly important if you are trying to link past research results to new research results: if you can no longer replicate your original results with your updated code, then you must regard your code with suspicion, *unless* the changes are intentional.

By contrast, both unit and functional tests tend to be *expectation* based. By this I mean that you use the tests to lay out what behavior you *expect* from your code, and write your tests so that they *assert* that those expectations are met.

The difference between unit and functional tests is blurry in most actual implementations; unit tests tend to be much shorter and require less setup and teardown, while functional tests can be quite long. I like Kumar McMillan's distinction: functional tests tell you *when* your code is broken, while unit tests tell you *where* your code is broken. That is, because of the finer granularity of unit tests, a broken unit test can identify a particular piece of code as the source of an error, while functional tests merely tell you that a feature is broken.

1.3.2 The doctest module

Let's start by looking at the doctest module. If you've been following along, you will be familiar with doctests, because I've been using them throughout this text! A doctest links code and behavior explicitly in a nice documentation format. Here's an example:

```
>>> print 'hello, world'
hello, world
```

When doctest sees this in a docstring or in a file, it knows that it should execute the code after the '>>>' and compare the actual output of the code to the strings immediately following the '>>>' line.

To execute doctests, you can use the doctest API that comes with Python: just type:

```
import doctest
doctest.testfile(textfile)
```

or

```
import doctest
doctest.testmod(modulefile)
```

The doctest docs contain complete documentation for the module, but in general there are only a few things you need to know.

First, for multi-line entries, use '...' instead of '>>>':

```
>>> def func():
...   print 'hello, world'
>>> func()
hello, world
```

Second, if you need to elide exception code, use '...':

```
>>> raise Exception("some error occurred")
Traceback (most recent call last):
   ...
Exception: some error occurred
```

More generally, you can use '...' to match random output, as long as you you specify a doctest directive:

```
>>> import random
>>> print 'random number:', random.randint(0, 10)
random number: ...
```

Third, doctests are terminated with a blank line, so if you explicitly expect a blank line, you need to use a special construct:

```
>>> print ''
```

To test out some doctests of your own, try modifying these files and running them with `doctest.testfile`.

Doctests are useful in a number of ways. They encourage a kind of conversation with the user, in which you (the author) demonstrate how to actually use the code. And, because they're executable, they ensure that your code works as you expect. However, they can also result in quite long docstrings, so I recommend putting long doctests in files separate from the code files. Short doctests can go anywhere – in module, class, or function docstrings.

1.3.3 Unit tests with unittest

If you've heard of automated testing, you've almost certainly heard of unit tests. The idea behind unit tests is that you can constrain the behavior of small units of code to be correct by testing the bejeezus out of them; and, if your smallest code units are broken, then how can code built on top of them be good?

The unittest module comes with Python. It provides a framework for writing and running unit tests that is at least convenient, if not as simple as it could be (see the 'nose' stuff, below, for something that is simpler).

Unit tests are almost always demonstrated with some sort of numerical process, and I will be no different. Here's a simple unit test, using the unittest module:

```
test_sort.py:

 #! /usr/bin/env python
 import unittest
 class Test(unittest.TestCase):
  def test_me(self):
     seq = [ 5, 4, 1, 3, 2 ]
     seq.sort()
     self.assertEqual(seq, [1, 2, 3, 4, 5])

 if __name__ == '__main__':
    unittest.main()
```

If you run this, you'll see the following output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Here, `unittest.main()` is running through all of the symbols in the global module namespace and finding out which classes inherit from `unittest.TestCase`. Then, for each such class, it finds all methods starting with `test`, and for each one it instantiates a new object and runs the function: so, in this case, just:

```
Test().test_me()
```

If any method fails, then the failure output is recorded and presented at the end, but the rest of the test methods are run irrespective.

`unittest` also includes support for test *fixtures*, which are functions run before and after each test; the idea is to use them to set up and tear down the test environment. In the code below, `setUp` creates and shuffles the `self.seq` sequence, while `tearDown` deletes it.

```
test_sort2.py:

   #! /usr/bin/env python
   import unittest
   import random

   class Test(unittest.TestCase):
       def setUp(self):
           self.seq = range(0, 10)
           random.shuffle(self.seq)

       def tearDown(self):
           del self.seq

       def test_basic_sort(self):
           self.seq.sort()
           self.assertEqual(self.seq, range(0, 10))

       def test_reverse(self):
           self.seq.sort()
           self.seq.reverse()
           self.assertEqual(self.seq, [9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

       def test_destruct(self):
           self.seq.sort()
           del self.seq[-1]
           self.assertEqual(self.seq, range(0, 9))

   unittest.main()
```

In both of these examples, it's important to realize that an *entirely new object* is created, and the fixtures run, for each test function. This lets you write tests that alter or destroy test data without having to worry about interactions between the code in different tests.

1.3.4 Testing with nose

nose is a unit test discovery system that makes writing and organizing unit tests very easy. I've actually written a whole separate article on them, so we should go check that out.

1.3.5 Code coverage analysis

figleaf is a code coverage recording and analysis system that I wrote and maintain. It's published in PyPI, so you can install it with easy_install.

Basic use of figleaf is very easy. If you have a script `program.py`, rather than typing

```
% python program.py
```

to run the script, run

```
% figleaf program.py
```

This will transparently and invisibly record coverage to the file '.figleaf' in the current directory. If you run the program several times, the coverage will be aggregated.

To get a coverage report, run 'figleaf2html'. This will produce a subdirectory `html/` that you can view with any Web browser; the index.html file will contain a summary of the code coverage, along with links to individual annotated files. In these annotated files, executed lines are colored green, while lines of code that are not executed are colored red. Lines that are not considered lines of code (e.g. docstrings, or comments) are colored black.

My main use for code coverage analysis is in testing (which is why I discuss it in this section!) I record the code coverage for my unit and functional tests, and then examine the output to figure out which files or libraries to focus on testing next. As I discuss below, it is relatively easy to achieve 70-80% code coverage by this method.

When is code coverage most useful? I think it's most useful in the early and middle stages of testing, when you need to track down code that is not touched by your tests. However, 100% code coverage by your tests doesn't guarantee bug free code: this is because figleaf only measures line coverage, not branch coverage. For example, consider this code:

```
if a.x or a.y:
   f()
```

If `a.x` is True in all your tests, then `a.y` will never be evaluated – even though `a` may not have an attribute `y`, which would cause an AttributeError (which would in turn be a bug, if not properly caught). Python does not record which subclauses of the `if` statement are executed, so without analyzing the structure of the program there's no simple way to figure it out.

Here's another buggy example with 100% code coverage:

```
def f(a):
   if a:
      a = a.upper()
   return a.strip()

s = f("some string")
```

Here, there's an implicit `else` after the if statement; the function f() could be rewritten to this:

```
def f(a):
   if a:
      a = a.upper()
   else:
      pass
   return a.strip()

s = f("some string")
```

and the pass statement would show up as "not executed".

So, bottom line: 100% test coverage is *necessary* for a well-tested program, because code that is not executed by any test at all is simply not being tested. However, 100% test coverage is not *sufficient* to guarantee that your program is free of bugs, as you can see from some of the examples above.

1.3.6 Adding tests to an existing project

This testing discussion should help to convince you that not only *should* you test, but that there are plenty of tools available to *help* you test in Python. It may even give you some ideas about how to start testing new projects. However, retrofitting an *existing* project with tests is a different, challenging problem – where do you start? People are often overwhelmed by the amount of code they've written in the past.

I suggest the following approach.

First, start by writing a test for each bug as they are discovered. The procedure is fairly simple: isolate the cause of the bug; write a test that demonstrates the bug; fix the bug; verify that the test passes. This has several benefits in the short term: you are fixing bugs, you're discovering weak points in your software, you're becoming more familiar with the testing approach, and you can start to think about commonalities in the fixtures necessary to *support* the tests.

Next, take out some time – half a day or so – and write fixtures and functional tests for some small chunk of code; if you can, pick a piece of code that you're planning to clean up or extend. Don't worry about being exhaustive, but just write tests that target the main point of the code that you're working on.

Repeat this a few times. You should start to discover the benefits of testing at this point, as you increasingly prevent bugs from occurring in the code that's covered by the tests. You should also start to get some idea of what fixtures are necessary for your code base.

Now use code coverage analysis to analyze what code your tests cover, and what code isn't covered. At this point you can take a targetted approach and spend some time writing tests aimed directly at uncovered areas of code. There should now be tests that cover 30-50% of your code, at least (it's very easy to attain this level of code coverage!).

Once you've reached this point, you can either decide to focus on increasing your code coverage, or (my recommendation) you can simply continue incrementally constraining your code by writing tests for bugs and new features. Assuming you have a fairly normal code churn, you should get to the point of 70-80% coverage within a few months to a few years (depending on the size of the project!)

This approach is effective because at each stage you get immediate feedback from your efforts, and it's easier to justify to managers than a whole-team effort to add testing. Plus, if you're unfamiliar with testing or with parts of the code base, it gives you time to adjust and adapt your approach to the needs of the particular project.

Two articles that discuss similar approaches in some detail are available online: Strangling Legacy Code, and Growing Your Test Harness. I can also recommend the book Working Effectively with Legacy Code, by Robert Martin.

1.3.7 Concluding thoughts on automated testing

Starting to do automated testing of your code can lead to immense savings in maintenance and can also increase productivity dramatically. There are a number of reasons why automated testing can help so much, including quick discovery of regressions, increased design awareness due to more interaction with the code, and early detection of simple bugs as well as unwanted epistatic interactions between code modules. The single biggest improvement for me has been the ability to refactor code without worrying as much about breakage. In my personal experience, automated testing is a 5-10x productivity booster when working alone, and it can save multi-person teams from potentially disastrous errors in communication.

Automated testing is not, of course, a silver bullet. There are several common worries.

One worry is that by increasing the total amount of code in a project, you increase both the development time and the potential for bugs and maintenance problems. This is certainly possible, but test code is very different from regular project code: it can be removed much more easily (which can be done whenever the code being tested undergoes revision), and it should be *much* simpler even if it is in fact bulkier.

Another worry is that too much of a focus on testing will decrease the drive for new functionality, because people will focus more on writing tests than they will on the new code. While this is partly a managerial issues, it is worth pointing out that the process of writing new code will be dramatically faster if you don't have to worry about old code breaking in unexpected ways as you add functionality.

A third worry is that by focusing on automation, you will miss bugs in code that is difficult to automate. There are two considerations here. First, it is possible to automate quite a bit of testing; the decision not to automat a particular test is almost always made because of financial or time considerations rather than technical limitations. And, second, automated testing is simply not a replacement for certain types of manual testing – in particular, exploratory testing, in which the programmers or users interact with the program, will always turn up new bugs, and is worth doing independent of the automated tests.

How much to test, and what to test, are decisions that need to be made on an individual project basis; there are no hard and fast rules. However, I feel confident in saying that some automated testing will always improve the quality of your code and result in maintenance improvements.

1.4 An Extended Introduction to the nose Unit Testing Framework

Welcome! This is an introduction, with lots and lots of examples, to the nose unit test discovery & execution framework. If that's not what you want to read, I suggest you hit the Back button now.

The latest version of this document can be found at

> http://ivory.idyll.org/articles/nose-intro.html

(Last modified October 2006.)

1.4.1 What are unit tests?

A unit test is an automated code-level test for a small "unit" of functionality. Unit tests are often designed to test a broad range of the expected functionality, including any weird corner cases and some tests that *should not* work. They tend to interact minimally with external resources like the disk, the network, and databases; testing code that accesses these resources is usually put under functional tests, regression tests, or integration tests.

(There's lots of discussion on whether unit tests should do things like access external resources, and whether or not they are still "unit" tests if they do. The arguments are fun to read, and I encourage you to read them. I'm going to stick with a fairly pragmatic and broad definition: anything that exercises a small, fairly isolated piece of functionality is a unit test.)

Unit tests are almost always pretty simple, by intent; for example, if you wanted to test an (intentionally naive) regular expression for validating the form of e-mail addresses, your test might look something like this:

```
EMAIL_REGEXP = r'[\S.]+@[\S.]+'

def test_email_regexp():
   # a regular e-mail address should match
   assert re.match(EMAIL_REGEXP, 'test@nowhere.com')

   # no domain should fail
   assert not re.match(EMAIL_REGEXP, 'test@')
```

There are a couple of ways to integrate unit tests into your development style. These include Test Driven Development, where unit tests are written prior to the functionality they're testing; during refactoring, where existing code – sometimes code without any automated tests to start with – is retrofitted with unit tests as part of the refactoring process; bug fix testing, where bugs are first pinpointed by a targetted test and then fixed; and straight test enhanced development, where tests are written organically as the code evolves. In the end, I think it matters more that you're writing unit tests than it does exactly how you write them.

For me, the most important part of having unit tests is that they can be run *quickly*, *easily*, and *without any thought* by developers. They serve as executable, enforceable documentation for function and API, and they also serve as an invaluable reminder of bugs you've fixed in the past. As such, they improve my ability to more quickly deliver functional code – and that's really the bottom line.

1.4.2 Why use a framework? (and why nose?)

It's pretty common to write tests for a library module like so:

```
def test_me():
   # ... many tests, which raise an Exception if they fail ...

if __name__ -- '__main__':
   test_me()
```

The 'if' statement is a little hook that runs the tests when the module is executed as a script from the command line. This is great, and fulfills the goal of having automated tests that can be run easily. Unfortunately, they *cannot be run without thought*, which is an amazingly important and oft-overlooked requirement for automated tests! In practice, this means that they will only be run when that module is being worked on – a big problem.

People use unit test discovery and execution frameworks so that they can add tests to existing code, execute those tests, and get a simple report, without thinking. Below, you'll see some of the advantages that using such a framework gives you: in addition to finding and running your tests, frameworks can let you selectively execute certain tests, capture and collate error output, and add coverage and profiling information. (You can always write your own framework – but why not take advantage of someone else's, even if they're not as smart as you?)

"Why use nose in particular?" is a more difficult question. There are many unit test frameworks in Python, and more arise every day. I personally use nose, and it fits my needs fairly well. In particular, it's actively developed, by a guy (Jason Pellerin) who answers his e-mail pretty quickly; it's fairly stable (it's in beta at the time of this writing); it has a really fantastic plug-in architecture that lets me extend it in convenient ways; it integrates well with distutils; it can be adapted to mimic any *other* unit test discovery framework pretty easily; and it's being used by a number of big projects, which means it'll probably still be around in a few years.

I hope the best reason *for you* to use nose will be that I'm giving you this extended introduction ;).

1.4.3 A few simple examples

First, install nose. Using setuptools, this is easy:

```
easy_install nose
```

Now let's start with a few examples. Here's the simplest nose test you can write:

```
def test_b():
    assert 'b' -- 'b'
```

Put this in a file called `test_me.py`, and then run `nosetests`. You will see this output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.005s

OK
```

If you want to see exactly what test was run, you can use `nosetests -v`.

```
test_stuff.test_b ... ok

----------------------------------------------------------------------
Ran 1 test in 0.015s

OK
```

Here's a more complicated example.

```
class TestExampleTwo:
    def test_c(self):
        assert 'c' -- 'c'
```

Here, nose will first create an object of type `TestExampleTwo`, and only *then* run `test_c`:

```
test_stuff.TestExampleTwo.test_c ... ok
```

Most new test functions you write should look like either of these tests – a simple test function, or a class containing one or more test functions. But don't worry – if you have some old tests that you ran with `unittest`, you can still run them. For example, this test:

```
class ExampleTest(unittest.TestCase):
    def test_a(self):
        self.assert_(1 -- 1)
```

still works just fine:

```
test_a (test_stuff.ExampleTest) ... ok
```

Test fixtures

A fairly common pattern for unit tests is something like this:

```
def test():
   setup_test()
   try:
      do_test()
      make_test_assertions()
   finally:
      cleanup_after_test()
```

Here, `setup_test` is a function that creates necessary objects, opens database connections, finds files, etc. – anything that establishes necessary preconditions for the test. Then `do_test` and `make_test_assertions` acually run the test code and check to see that the test completed successfully. Finally – and independently of whether or not the test *succeeded* – the preconditions are cleaned up, or "torn down".

This is such a common pattern for unit tests that most unit test frameworks let you define setup and teardown "fixtures" for each test; these fixtures are run before and after the test, as in the code sample above. So, instead of the pattern above, you'd do:

```
def test():
   do_test()
   make_test_assertions()

test.setUp = setup_test
test.tearDown = cleanup_after_test
```

The unit test framework then examines each test function, class, and method for fixtures, and runs them appropriately.

Here's the canonical example of fixtures, used in classes rather than in functions:

```
class TestClass:
   def setUp(self):
      ...

   def tearDown(self):
      ...

   def test_case_1(self):
      ...

   def test_case_2(self):
      ...

   def test_case_3(self):
      ...
```

The code that's actually run by the unit test framework is then

```
for test_method in get_test_classes():
   obj = TestClass()
   obj.setUp()
   try:
      obj.test_method()
   finally:
      obj.tearDown()
```

That is, for *each* test case, a new object is created, set up, and torn down – thus approximating the Platonic ideal of running each test in a completely new, pristine environment.

(Fixture, incidentally, comes from the Latin "fixus", meaning "fixed". The origin of its use in unit testing is not clear to me, but you can think of fixtures as permanent appendages of a set of tests, "fixed" in place. The word "fixtures" make more sense when considered as part of a test suite than when used on a single test – one fixture for each *set* of tests.)

Examples are included!

All of the example code in this article is available in a .tar.gz file. Just download the package at

```
http://darcs.idyll.org/~t/projects/nose-demo.tar.gz
```

and unpack it somewhere; information on running the examples is in each section, below.

To run the simple examples above, go to the top directory in the example distribution and type

```
nosetests -w simple/ -v
```

1.4.4 A somewhat more complete guide to test discovery and execution

nose is a unit test **discovery** and execution package. Before it can execute any tests, it needs to discover them. nose has a set of rules for discovering tests, and then a fixed protocol for running them. While both can be modified by plugins, for the moment let's consider only the default rules.

nose only looks for tests under the working directory – normally the current directory, unless you specify one with the `-w` command line option.

Within the working directory, it looks for any directories, files, modules, or packages that match the test pattern. [...] In particular, note that packages are recursively scanned for test cases.

Once a test module or a package is found, it's loaded, the setup fixtures are run, and the modules are examined for test functions and classes – again, anything that matches the test pattern. Any test functions are run – along with associated fixtures – and test classes are also executed. For each test method in test classes, a new object of that type is instantiated, the setup fixture (if any) is run, the test method is run, and (if there was a setup fixture) the teardown fixture is run.

Running tests

Here's the basic logic of test running used by nose (in Python pseudocode)

```
if has_setup_fixture(test):
   run_setup(test)

try:

   run_test(test)

finally:
   if has_setup_fixture(test):
      run_teardown(test)
```

Unlike tests themselves, however, test fixtures on test modules and test packages are run only once. This extends the test logic above to this (again, pseudocode):

```
### run module setup fixture

if has_setup_fixture(test_module):
   run_setup(test_module)

### run all tests

try:
   for test in get_tests(test_module):

      try:                          ### allow individual tests to fail
         if has_setup_fixture(test):
            run_setup(test)

         try:

            run_test(test)

         finally:
            if has_setup_fixture(test):
               run_teardown(test)
      except:
         report_error()

finally:

   ### run module teardown fixture

   if has_setup_fixture(test_module):
      run_teardown(test_module)
```

A few additional notes:

- if the setup fixture fails, no tests are run and the teardown fixture isn't run, either.
- if there is no setup fixture, then the teardown fixture is not run.
- whether or not the tests succeed, the teardown fixture is run.
- all tests are executed even if some of them fail.

Debugging test discovery

nose can only execute tests that it *finds*. If you're creating a new test suite, it's relatively easy to make sure that nose finds all your tests – just stick a few `assert 0` statements in each new module, and if nose doesn't kick up an error it's not running those tests! It's more difficult when you're retrofitting an existing test suite to run inside of nose; in the extreme case, you may need to write a plugin or modify the top-level nose logic to find the existing tests.

The main problem I've run into is that nose will only find tests that are properly named *and* within directory or package hierarchies that it's actually traversing! So placing your test modules under the directory `my_favorite_code` won't work, because nose will not even enter that directory. However, if you make `my_favorite_code` a *package*, then nose *will* find your tests because it traverses over modules within packages.

In any case, using the `-vv` flag gives you verbose output from nose's test discovery algorithm. This will tell you whether or not nose is even looking in the right place(s) to find your tests.

1.4.5 The nose command line

Apart from the plugins, there are only a few options that I use regularly.

-w: Specifying the working directory

nose only looks for tests in one place. The -w flag lets you specify that location; e.g.

```
nosetests -w simple/
```

will run only those tests in the directory `./simple/`.

As of the latest development version (October 2006) you can specify multiple working directories on the command line:

```
nosetests -w simple/ -w basic/
```

See *Running nose programmatically* for an example of how to specify multiple working directories using Python, in nose 0.9.

-s: Not capturing stdout

By default, nose captures all output and only presents stdout from tests that fail. By specifying '-s', you can turn this behavior off.

-v: Info and debugging output

nose is intentionally pretty terse. If you want to see what tests are being run, use '-v'.

Specifying a list of tests to run

nose lets you specify a set of tests on the command line; only tests that are *both* discovered *and* in this set of tests will be run. For example,

```
nosetests -w simple tests/test_stuff.py:test_b
```

only runs the function `test_b` found in `simple/tests/test_stuff.py`.

1.4.6 Running doctests in nose

Doctests are a nice way to test individual Python functions in a convenient documentation format. For example, the docstring for the function `multiply`, below, contains a doctest:

```
def multiply(a, b):
  """
  'multiply' multiplies two numbers and returns the result.

  >>> multiply(5, 10)                # doctest: +SKIP
  50
  >>> multiply(-1, 1)                # doctest: +SKIP
  -1
  >>> multiply(0.5, 1.5)             # doctest: +SKIP
  0.75
  """
  return a*b
```

(Ignore the SKIP pragmas; they're put in so that this file itself can be run through doctest without failing...)

The doctest module (part of the Python standard module) scans through all of the docstrings in a package or module, executes any line starting with a `>>>`, and compares the actual output with the expected output contained in the docstring.

Typically you run these directly on a module level, using the sort of `__main__` hack I showed above. The doctest plug-in for nose adds doctest discovery into nose – all non-test packages are scanned for doctests, and any doctests are executed along with the rest of the tests.

To use the doctest plug-in, go to the directory containing the modules and packages you want searched and do

```
nosetests --with-doctest
```

All of the doctests will be automatically found and executed. Some example doctests are included with the demo code, under `basic`; you can run them like so:

```
% nosetests -w basic/ --with-doctest -v
doctest of app_package.stuff.function_with_doctest ... ok
...
```

Note that by default nose only looks for doctests in *non-test* code. You can add `--doctest-tests` to the command line to search for doctests in your test code as well.

The doctest plugin gives you a nice way to combine your various low-level tests (e.g. both unit tests and doctests) within one single nose run; it also means that you're less likely to forget about running your doctests!

1.4.7 The 'attrib' plug-in – selectively running subsets of tests

The attrib extension module lets you flexibly select subsets of tests based on test *attributes* – literally, Python variables attached to individual tests.

Suppose you had the following code (in `attr/test_attr.py`):

```
def testme1():
    assert 1

testme1.will_fail = False

def testme2():
    assert 0

testme2.will_fail = True

def testme3():
    assert 1
```

Using the attrib extension, you can select a subset of these tests based on the attribute `will_fail`. For example, `nosetests -a will_fail` will run only `testme2`, while `nosetests -a \!will_fail` will run both `testme1` and `testme3`. You can also specify precise values, e.g. `nosetests -a will_fail=False` will run only `testme1`, because `testme3` doesn't have the attribute `will_fail`.

You can also tag tests with *lists* of attributes, as in `attr/test_attr2.py`:

```
def testme5():
    assert 1

testme5.tags = ['a', 'b']

def testme6():
    assert 1

testme6.tags = ['a', 'c']
```

Then `nosetests -a tags=a` will run both `testme5` and `testme6`, while `nosetests -a tags=b` will run only `testme5`.

Attribute tags also work on classes and methods as you might expect. In `attr/test_attr3.py`, the following code

```
class TestMe:
    x = True

    def test_case1(self):
        assert 1

    def test_case2(self):
        assert 1

    test_case2.x = False
```

lets you run both `test_case1` (with `-a x`) and `test_case2` (with `-a \!x`); here, methods inherit the attributes of their parent class, but can override the class attributes with method-specific attributes.

1.4.8 Running nose programmatically

nose has a friendly top-level API which makes it accessible to Python programs. You can run nose inside your own code by doing this:

```
import nose

### configure paths, etc here

nose.run()

### do other stuff here
```

By default nose will pick up on `sys.argv`; if you want to pass in your own arguments, use `nose.run(argv=args)`. You can also override the default test collector, test runner, test loader, and environment settings at this level. This makes it convenient to add in certain types of new behavior; see `multihome/multihome-nose` for a script that lets you specify multiple "test home directories" by overriding the test collector.

There are a few caveats to mention about using the top-level nose commands. First, be sure to use `nose.run`, not `nose.main` – `nose.main` will exit after running the tests (although you can wrap it in a 'try/finally' if you insist). Second, in the current version of nose (0.9b1), `nose.run` swipes `sys.stdout`, so `print` will not yield any output after `nose.run` completes. (This should be fixed soon.)

1.4.9 Writing plug-ins – a simple guide

As nice as nose already is, the plugin system is probably the best thing about it. nose uses the setuptools API to load all registered nose plugins, allowing you to install 3rd party plugins quickly and easily; plugins can modify or override output handling, test discovery, and test execution.

nose comes with a couple of plugins that demonstrate the power of the plugin API; I've discussed two (the attrib and doctest plugins) above. I've also written a few, as part of the pinocchio nose extensions package.

Here are a few tips and tricks for writing plugins.

- read through the `nose.plugins.IPluginInterface` code a few times.
- for the `want*` functions (`wantClass`, `wantMethod`, etc.) you need to know:
  - a return value of True indicates that your plugin wants this item.
  - a return value of False indicates that your plugin doesn't want this item.
  - a return value of None indicates that your plugin doesn't care about this item.

  Also note that plugins aren't guaranteed to be run in any particular order, so you have to order them yourself if you need this. See the `pinocchio.decorator` module (part of pinocchio) for an example.
- abuse stderr. As much as I like the logging package, it can confuse matters by capturing output in ways I don't fully understand (or at least don't want to have to configure for debugging purposes). While you're working on your plugin, put `import sys; err = sys.stderr` at the top of your plugin module, and then use `err.write` to produce debugging output.
- notwithstanding the stderr advice, `-vv` is your friend – it will tell you that your test file isn't even being examined for tests, and it will also tell you what order things are being run in.
- write your initial plugin code by simply copying `nose.plugins.attrib` and deleting everything that's not generic. This greatly simplifies getting your plugin loaded & functioning.
- to register your plugin, you need this code in e.g. a file called 'setup.py'

```
    from setuptools import setup

    setup(
        name='my_nose_plugin',
        packages = ['my_nose_plugin'],
        entry_points = {
            'nose.plugins': [
                'pluginA = my_nose_plugin:pluginA',
                ]
            },
    )
```

You can then install (and register) the plugin with `easy_install .`, run in the directory containing 'setup.py'.

1.4.10 nose caveats – let the buyer beware, occasionally

I've been using nose fairly seriously for a while now, on multiple projects. The two most frustrating problems I've had are with the output capture (mentioned above, in *Running nose programmatically*) and a situation involving the `logging` module. The output capture problem is easily taken care of, once you're aware of it – just be sure to save sys.stdout before running any nose code. The logging module problem cropped up when converting an existing unit test suite over to nose: the code tested an application that used the `logging` module, and reconfigured logging so that nose's output didn't show up. This frustrated my attempts to trace test discovery to no end – as far as I could tell, nose was simply stopping test discovery at a certain point! I doubt there's a general solution to this, but I thought I'd mention it.

1.4.11 Credits

Jason Pellerin, besides for being the author of nose, has been very helpful in answering questions! Terry Peppers and Chad Whitacre kindly sent me errata.

CHAPTER 2

Day 2

Contents:

2.1 Idiomatic Python revisited

2.1.1 sets

Sets recently (2.4?) migrated from a stdlib component into a default type. They're exactly what you think: unordered collections of values.

```
>>> s = set((1, 2, 3, 4, 5))
>>> t = set((4, 5, 6))
>>> print s
set([1, 2, 3, 4, 5])
```

You can union and intersect them:

```
>>> print s.union(t)
set([1, 2, 3, 4, 5, 6])
>>> print s.intersection(t)
set([4, 5])
```

And you can also check for supersets and subsets:

```
>>> u = set((4, 5, 6, 7))
>>> print t.issubset(u)
True
>>> print u.issubset(t)
False
```

One more note: you can convert between sets and lists pretty easily:

```
>>> sl = list(s)
>>> ss = set(sl)
```

2.1.2 `any` and `all`

`all` and `any` are two new functions in Python that work with iterables (e.g. lists, generators, etc.). `any` returns True if *any* element of the iterable is True (and False otherwise); `all` returns True if *all* elements of the iterable are True (and False otherwise).

Consider:

```
>>> x = [ True, False ]
>>> print any(x)
True
>>> print all(x)
False
```

```
>>> y = [ True, True ]
>>> print any(x)
True
>>> print all(x)
False
```

```
>>> z = [ False, False ]
>>> print any(z)
False
>>> print all(z)
False
```

2.1.3 Exceptions and exception hierarchies

You're all familiar with exception handling using try/except:

```
>>> x = [1, 2, 3, 4, 5]
>>> x[10]
Traceback (most recent call last):
   ...
IndexError: list index out of range
```

You can catch all exceptions quite easily:

```
>>> try:
...    y = x[10]
... except:
...    y = None
```

but this is considered bad form, because of the potential for over-broad exception handling:

```
>>> try:
...    y = x["10"]
... except:
...    y = None
```

In general, try to catch the exception most specific to your code:

```
>>> try:
...    y = x[10]
... except IndexError:
...    y = None
```

...because then you will see the errors you didn't plan for:

```
>>> try:
...    y = x["10"]
... except IndexError:
...    y = None
Traceback (most recent call last):
   ...
TypeError: list indices must be integers
```

Incidentally, you can re-raise exceptions, potentially after doing something else:

```
>>> try:
...    y = x[10]
... except IndexError:
...    # do something else here #
...    raise
Traceback (most recent call last):
   ...
IndexError: list index out of range
```

There are some special exceptions to be aware of. Two that I run into a lot are SystemExit and KeyboardInterrupt. KeyboardInterrupt is what is raised when a CTRL-C interrupts Python; you can handle it and exit gracefully if you like, e.g.

```
>>> try:
...    # do_some_long_running_task()
...    pass
... except KeyboardInterrupt:
...    sys.exit(0)
```

which is sometimes nice for things like Web servers (more on that tomorrow).

SystemExit is also pretty useful. It's actually an exception raised by `sys.exit`, i.e.

```
>>> import sys
>>> try:
...    sys.exit(0)
... except SystemExit:
...    pass
```

means that sys.exit has no effect! You can also raise SystemExit instead of calling sys.exit, e.g.

```
>>> raise SystemExit(0)
Traceback (most recent call last):
   ...
SystemExit: 0
```

is equivalent to `sys.exit(0)`:

```
>>> sys.exit(0)
Traceback (most recent call last):
   ...
SystemExit: 0
```

Another nice feature of exceptions is exception hierarchies. Exceptions are just classes that derive from `Exception`, and you can catch exceptions based on their base classes. So, for example, you can catch most standard errors by catching the StandardError exception, from which e.g. IndexError inherits:

```
>>> print issubclass(IndexError, StandardError)
True
```

```
>>> try:
...    y = x[10]
... except StandardError:
...    y = None
```

You can also catch some exceptions more specifically than others. For example, KeyboardInterrupt inherits from Exception, and some times you want to catch KeyboardInterrupts while ignoring all other exceptions:

```
>>> try:
...    # ...
...    pass
... except KeyboardInterrupt:
...    raise
... except Exception:
...    pass
```

Note that if you want to print out the error, you can do coerce a string out of the exception to present to the user:

```
>>> try:
...    y = x[10]
... except Exception, e:
...    print 'CAUGHT EXCEPTION!', str(e)
CAUGHT EXCEPTION! list index out of range
```

Last but not least, you can define your own exceptions and exception hierarchies:

```
>>> class MyFavoriteException(Exception):
...    pass
>>> raise MyFavoriteException
Traceback (most recent call last):
   ...
MyFavoriteException
```

I haven't used this much myself, but it is invaluable when you are writing packages that have a lot of different detailed exceptions that you might want to let users handle.

(By default, I usually raise a simple Exception in my own code.)

Oh, one more note: AssertionError. Remember assert?

```
>>> assert 0
Traceback (most recent call last):
   ...
AssertionError
```

Yep, it raises an AssertionError that you can catch, if you REALLY want to...

2.1.4 Function Decorators

Function decorators are a strange beast that I tend to use only in my testing code and not in my actual application code. Briefly, function decorators are functions that take functions as arguments, and return other functions. Confused? Let's see a simple example that makes sure that no keyword argument named 'something' ever gets passed into a function:

```
>>> def my_decorator(fn):
...
...    def new_fn(*args, **kwargs):
...       if 'something' in kwargs:
...          print 'REMOVING', kwargs['something']
...          del kwargs['something']
...       return fn(*args, **kwargs)
...
...    return new_fn
```

To apply this decorator, use this funny @ syntax:

```
>>> @my_decorator
... def some_function(a=5, b=6, something=None, c=7):
...   print a, b, something, c
```

OK, now `some_function` has been invisibly replaced with the result of `my_decorator`, which is going to be `new_fn`. Let's see the result:

```
>>> some_function(something='MADE IT')
REMOVING MADE IT
5 6 None 7
```

Mind you, without the decorator, the function does exactly what you expect:

```
>>> def some_function(a=5, b=6, something=None, c=7):
...   print a, b, something, c
>>> some_function(something='MADE IT')
5 6 MADE IT 7
```

OK, so this is a bit weird. What possible uses are there for this??

Here are three example uses:

First, synchronized functions like in Java. Suppose you had a bunch of functions (f1, f2, f3...) that could not be called concurrently, so you wanted to play locks around them. You could do this with decorators:

```
>>> import threading
>>> def synchronized(fn):
...   lock = threading.Lock()
...
...   def new_fn(*args, **kwargs):
...      lock.acquire()
...      print 'lock acquired'
...      result = fn(*args, **kwargs)
...      lock.release()
...      print 'lock released'
...      return result
...
...   return new_fn
```

and then when you define your functions, they will be locked:

```
>>> @synchronized
... def f1():
...   print 'in f1'
>>> f1()
lock acquired
in f1
lock released
```

Second, adding attributes to functions. (This is why I use them in my testing code sometimes.)

```
>>> def attrs(**kwds):
...    def decorate(f):
...        for k in kwds:
...            setattr(f, k, kwds[k])
...        return f
...    return decorate
```

```
>>> @attrs(versionadded="2.2",
...       author="Guido van Rossum")
... def mymethod(f):
...    pass
```

```
>>> print mymethod.versionadded
2.2
>>> print mymethod.author
Guido van Rossum
```

Third, memoize/caching of results. Here's a really simple example; you can find much more general ones online, in particular on the Python Cookbook site.

Imagine that you have a CPU-expensive one-parameter function:

```
>>> def expensive(n):
...   print 'IN EXPENSIVE', n
...   # do something expensive here, like calculate n'th prime
```

You could write a caching decorator to wrap this function and record results transparently:

```
>>> def simple_cache(fn):
...   cache = {}
...
...   def new_fn(n):
...      if n in cache:
...         print 'FOUND IN CACHE; RETURNING'
...         return cache[n]
...
...      # otherwise, call function & record value
...      val = fn(n)
...      cache[n] = val
...      return val
...
...   return new_fn
```

Then use this as a decorator to wrap the expensive function:

```
>>> @simple_cache
... def expensive(n):
...   print 'IN THE EXPENSIVE FN:', n
...   return n**2
```

Now, when you call this function twice with the same argument, if will only do the calculation once; the second time, the function call will be intercepted and the cached value will be returned.

```
>>> expensive(55)
IN THE EXPENSIVE FN: 55
3025
>>> expensive(55)
FOUND IN CACHE; RETURNING
3025
```

Check out Michele Simionato's writeup of decorators here for lots more information on decorators.

2.1.5 try/finally

Finally, we come to try/finally!

The syntax of try/finally is just like try/except:

```
try:
   do_something()
finally:
   do_something_else()
```

The purpose of try/finally is to ensure that something is done, whether or not an exception is raised:

```
>>> x = [0, 1, 2]
>>> try:
...   y = x[5]
... finally:
...   x.append('something')
Traceback (most recent call last):
   ...
IndexError: list index out of range
```

```
>>> print x
[0, 1, 2, 'something']
```

(It's actually semantically equivalent to:

```
>>> try:
...   y = x[5]
... except IndexError:
...   x.append('something')
...   raise
Traceback (most recent call last):
   ...
IndexError: list index out of range
```

but it's a bit cleaner, because the exception doesn't have to be re-raised and you don't have to catch a specific exception type.)

Well, why do you need this? Let's think about locking. First, get a lock:

```
>>> import threading
>>> lock = threading.Lock()
```

Now, if you're locking something, you want to be darn sure to *release* that lock. But what if an exception is raised right in the middle?

```
>>> def fn():
...     print 'acquiring lock'
...     lock.acquire()
...     y = x[5]
...     print 'releasing lock'
...     lock.release()
>>> try:
...     fn()
... except IndexError:
...   pass
acquiring lock
```

Note that 'releasing lock' is never printed: 'lock' is now left in a locked state, and next time you run 'fn' you will hang the program forever. Oops.

You can fix this with try/finally:

```
>>> lock = threading.Lock()             # gotta trash the previous lock, or hang!
>>> def fn():
```

```
...     print 'acquiring lock'
...     lock.acquire()
...     try:
...        y = x[5]
...     finally:
...        print 'releasing lock'
...        lock.release()
>>> try:
...    fn()
... except IndexError:
...    pass
acquiring lock
releasing lock
```

2.1.6 Function arguments, and wrapping functions

You may have noticed above (in the section on decorators) that we wrapped functions using this notation:

```
def wrapper_fn(*args, **kwargs):
    return fn(*args, **kwargs)
```

(This takes the place of the old 'apply'.) What does this do?

Here, *args assigns all of the positional arguments to a tuple 'args', and '**kwargs' assigns all of the keyword arguments to a dictionary 'kwargs':

```
>>> def print_me(*args, **kwargs):
...    print 'args is:', args
...    print 'kwargs is:', kwargs
```

```
>>> print_me(5, 6, 7, test='me', arg2=None)
args is: (5, 6, 7)
kwargs is: {'test': 'me', 'arg2': None}
```

When a function is called with this notation, the args and kwargs are unpacked appropriately and passed into the function. For example, the function `test_call`

```
>>> def test_call(a, b, c, x=1, y=2, z=3):
...    print a, b, c, x, y, z
```

can be called with a tuple of three args (matching 'a', 'b', 'c'):

```
>>> tuple_in = (5, 6, 7)
>>> test_call(*tuple_in)
5 6 7 1 2 3
```

with some optional keyword args:

```
>>> d = { 'x' : 'hello', 'y' : 'world' }
>>> test_call(*tuple_in, **d)
5 6 7 hello world 3
```

Incidentally, this lets you implement the 'dict' constructor in one line!

```
>>> def dict_replacement(**kwargs):
...     return kwargs
```

2.2 Measuring and Increasing Performance

"Premature optimization is the root of all evil (or at least most of it) in programming." Donald Knuth.

In other words, know thy code! The only way to find performance bottlenecks is to profile your code. Unfortunately, the situation is a bit more complex in Python than you would like it to be: see http://docs.python.org/lib/profile.html. Briefly, there are three (!?) standard profiling systems that come with Python: profile, cProfile (only since python 2.5!), and hotshot (thought note that profile and cProfile are Python and C implementations of the same API). There is also a separately maintained one called statprof, that I nominally maintain.

The ones included with Python are deterministic profilers, while statprof is a statistical profiler. What's the difference? To steal from the Python docs:

> Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

Let's go to the examples. Suppose we have two functions 'count1' and 'count2', and we want to run both and see where time is spent.

Here's some example hotshot code:

```
import hotshot, hotshot.stats
prof = hotshot.Profile('hotshot.prof')
prof.runcall(count1)
prof.runcall(count2)
prof.close()
stats = hotshot.stats.load('hotshot.prof')
stats.sort_stats('time', 'calls')
stats.print_stats(20)
```

and the resulting output:

```
      2 function calls in 5.769 CPU seconds

Ordered by: internal time, call count

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    4.335    4.335    4.335    4.335 count.py:8(count2)
     1    1.434    1.434    1.434    1.434 count.py:1(count1)
     0    0.000             0.000          profile:0(profiler)
```

Here's some example cProfile code:

```
def runboth():
    count1()
    count2()

import cProfile, pstats
cProfile.run('runboth()', 'cprof.out')

p = pstats.Stats('cprof.out')
p.sort_stats('time').print_stats(10)
```

and the resulting output:

```
Wed Jun 13 00:11:55 2007    cprof.out

         7 function calls in 5.643 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.817    3.817    4.194    4.194 count.py:8(count2)
        1    1.282    1.282    1.450    1.450 count.py:1(count1)
        2    0.545    0.272    0.545    0.272 {range}
        1    0.000    0.000    5.643    5.643 run-cprofile:8(runboth)
        1    0.000    0.000    5.643    5.643 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

And here's an example of statprof, the statistical profiler:

```
import statprof
statprof.start()
count1()
count2()
statprof.stop()
statprof.display()
```

And the output:

```
  %   cumulative      self
 time    seconds   seconds  name
 74.66      4.10      4.10  count.py:8:count2
 25.34      1.39      1.39  count.py:1:count1
  0.00      5.49      0.00  run-statprof:2:<module>
---
Sample count: 296
Total time: 5.490000 seconds
```

2.2.1 Which profiler should you use?

statprof used to report more accurate numbers than hotshot or cProfile, because hotshot and cProfile had to instrument the code (insert tracing statements, basically). However, the numbers shown above are pretty similar to each other and I'm not sure there's much of a reason to choose between them any more. So, I recommend starting with cProfile, because it's the officially supported one.

One note – none of these profilers really work all that well with threads, for a variety of reasons. You're best off doing performance measurements on non-threaded code.

2.2.2 Measuring code snippets with timeit

There's also a simple timing tool called timeit:

```
from timeit import Timer
from count import *

t1 = Timer("count1()", "from count import count1")
print 'count1:', t1.timeit(number=1)
```

```
t2 = Timer("count2()", "from count import count2")
print 'count2:', t2.timeit(number=1)
```

2.3 Speeding Up Python

There are a couple of options for speeding up Python.

2.3.1 psyco

(Taken almost verbatim from the psyco introduction!)

psyco is a specializing compiler that lets you run your existing Python code much faster, with *absolutely no change* in your source code. It acts like a just-in-time compiler by rewriting several versions of your code blocks and then optimizing them by specializing the variables they use.

The main benefit is that you get a 2-100x speed-up with an unmodified Python interpreter and unmodified source code. (You just need to import psyco.)

The main drawbacks are that it only runs on i386-compatible processors (so, not PPC Macs) and it's a bit of a memory hog.

For example, if you use the prime number generator generator code (see Idiomatic Python) to generate all primes under 100000, it takes about 10.4 seconds on my development server. With psyco, it takes about 1.6 seconds (that's about a 6x speedup). Even when doing less numerical stuff, I see at least a 2x speedup.

Installing psyco

(Note: psyco is an extension module and does not come in pre-compiled form. Therefore, you will need to have a Python-compatible C compiler installed in order to install psyco.)

Grab the latest psyco snapshot from here:

```
http://psyco.sourceforge.net/psycoguide/sources.html
```

unpack it, and run 'python setup.py install'.

Using psyco

Put the following code at the top of your __main__ Python script:

```
try:
   import psyco
   psyco.full()
except ImportError:
   pass
```

...and you're done. (Yes, it's magic!)

The only place where psyco won't help you much is when you have already recoded the CPU-intensive component of your code into an extension module.

2.3.2 pyrex

pyrex is a Python-like language used to create C modules for Python. You can use it for two purposes: to increase performance by (re)writing your code in C (but with a friendly extension language), and to make C libraries available to Python.

In the context of speeding things up, here's an example program:

```
def primes(int maxprime):
  cdef int n, k, i
  cdef int p[100000]
  result = []
  k = 0
  n = 2
  while n < maxprime:
    i = 0

    # test against previous primes
    while i < k and n % p[i] <> 0:
      i = i + 1

    # prime? if so, save.
    if i == k:
      p[k] = n
      k = k + 1
      result.append(n)
    n = n + 1

  return result
```

To compile this, you would execute:

```
pyrexc primes.pyx
gcc -c -fPIC -I /usr/local/include/python2.5 primes.c
gcc -shared primes.o -o primes.so
```

Or, more nicely, you can write a setup.py using some of the Pyrex helper functions:

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext                # <--

setup(
  name = "primes",
  ext_modules=[
    Extension("primes", ["primes.pyx"], libraries = [])
    ],
  cmdclass = {'build_ext': build_ext}
)
```

A few notes:

- 'cdef' is a C definition statement
- this is a "python-alike" language but not Python, per se ;)
- pyrex does handle a lot of the nasty C extension stuff for you.

There's an excellent guide to Pyrex available online here: http://ldots.org/pyrex-guide/.

I haven't used Pyrex much myself, but I have a friend who swears by it. My concerns are that it's a "C/Python-alike" language but not C or Python, and I have already memorized too many weird rules about too many languages!

We’ll encounter Pyrex a bit further down the road in the context of linking existing C/C++ code into your own code.

2.4 Tools to Help You Work

2.4.1 IPython

IPython is an interactive interpreter that aims to be a very convenient shell for working with Python.

Features of note:

- Tab completion
- ? and ?? help
- history
- CTRL-P search (in addition to standard CTRL-R/emacs)
- use an editor to write stuff, and export stuff into an edtor
- colored exception tracebacks
- automatic function/parameter call stuff
- auto-quoting with ‘,’
- ‘run’ (similar to execfile) but with -n, -i

See Quick tips for even more of a laundry list!

2.4.2 screen and VNC

screen is a non-graphical tool for running multiple text windows in a single login session.

Features:

- multiple windows w/hotkey switching
- copy/paste between windows
- detach/resume

VNC is a (free) graphical tool for persistent X Windows sessions (and Windows control, too).

To start:

```
% vncserver
```

WARNING: Running VNC on an open network is a big security risk!!

2.4.3 Trac

Trac is a really nice-looking and friendly project management Web site. It integrates a Wiki with a version control repository browser, a ticket management system, and some simple roadmap controls.

In particular, you can:

- browse the source code repository
- create tickets

- link checkin comments to specific tickets, revisions, etc.
- customize components, permissions, roadmaps, etc.
- view project status

It integrates well with subversion, which is "a better CVS".

2.5 pyparsing

2.5.1 Basic pyparsing

Matching text

```
>>> import pyparsing
>>> from pyparsing import Word, printables, Literal, StringEnd, Optional
>>> grammar = Literal("Hello,") + Word(printables)
>>> print grammar.parseString("Hello, nurse!")
['Hello,', 'nurse!']
```

So that's easy enough. But here, we *know* that 'Hello' is going to be there – we're really only interested in the word *after* 'Hello'.

```
>>> grammar = Literal("Hello,").suppress() + Word(printables)
>>> print grammar.parseString("Hello, nurse!")
['nurse!']
```

Let's break things down a bit:

```
>>> article = Word(printables)
>>> grammar = Literal("Hello,").suppress() + article
>>> print grammar.parseString("Hello, nurse!")
['nurse!']
```

Wouldn't it be nice to give the article ("nurse!") a name?

```
>>> article = Word(printables).setResultsName("the_article")
>>> grammar = Literal("Hello,").suppress() + article
>>> results = grammar.parseString("Hello, nurse!")
```

Now, given this, you can do two things: you can either refer to the result as an element of a list,

```
>>> print results[0]
nurse!
```

or by name:

```
>>> print results.the_article
nurse!
```

This kind of naming is incredibly handy and it's one of the main reasons I chose pyparsing. For example, let's try out more complicated example:

```
>>> article = Word(printables).setResultsName("the_article")
>>> salutation = (Literal("Hello,") | Literal("Goodbye,")).suppress()
>>> adjective = Word(printables).setResultsName('adjective')
>>> grammar = ((salutation + adjective + article) | \
...            (salutation + article)) + StringEnd()
```

This can match “Hello, nurse!”:

```
>>> results_1 = grammar.parseString("Hello, nurse!")
```

as well as “Goodbye, cruel world!”:

```
>>> results_2 = grammar.parseString("Goodbye, cruel world!")
```

but in *both* cases you can extract `the_article` by name:

```
>>> print results_1.the_article
nurse!
>>> print results_2.the_article
world!
```

And, of course, the `adjective` result is only set in the case where it was matched:

```
>>> print results_1.adjective

>>> print results_2.adjective
cruel
```

Note that this was not a particularly good example; rather than writing the grammer like so:

```
>>> grammar = ((salutation + adjective + article) | \
...             (salutation + article)) + StringEnd()
```

I could have written it like this:

```
>>> grammar = salutation + Optional(adjective) + article + StringEnd()
```

Interlude: whitespace drives tokenizing

I’ve studiously avoided talking about removing that final ‘!’ from “Hello, nurse!” until now, and that’s because it’s not this simple:

```
article = Word(printables) + "!"
print article.parseString("nurse!")
```

You see, the ‘+’ operator (a.k.a. `pyparsing.And`) only joins *tokens*, and pyparsing implicitly tokenizes on *whitespace*. So *this* would work,

```
>>> article = Word(printables) + "!"
>>> print article.parseString("nurse !")
['nurse', '!']
```

because now you’re parsing ‘Word AND !’.

(The only way I know of to remove that ‘!’ is to use a parse action:

```
>>> def remove_exclamation(x):
...     return x[0].rstrip('!')
>>> article = Word(printables)
>>> article = article.setParseAction(remove_exclamation)
>>> print article.parseString("nurse!")
['nurse']
```

More about parse actions later.)

Bottom line: tokenizing on whitespace makes a lot of things easier, and some things harder; I guess it’s a good thing...

SkipTo

Suppose you have an annoying section of text that you want to just jump past and not parse:

```
>>> annoying = """
... SOMETHING
... SOMETHING ELSE
... END
... MORE STUFF THAT MATTERS
... """
```

This is easily handled with SkipTo:

```
>>> from pyparsing import SkipTo
>>> end_marker = SkipTo("END", include=True).suppress()
>>> (end_marker + "MORE STUFF THAT MATTERS").parseString(annoying)
(['MORE STUFF THAT MATTERS'], {})
```

Regex matches

You can do regular expression matches too:

```
>>> from pyparsing import Regex
>>> hex_num = Regex("[0-9a-fA-F]+")
>>> hex_num.parseString("1f")
(['1f'], {})
```

Lists and more

Suppose we want to allow matches to multiple hex numbers. We can do this:

```
>>> from pyparsing import OneOrMore
>>> multiple = OneOrMore(hex_num)
>>> multiple.parseString('1f')
(['1f'], {})
>>> multiple.parseString('1f 2f 3f')
(['1f', '2f', '3f'], {})
```

Parse actions

Parse actions are functions that are run on parsed tokens; generally, the result of the parse action replaces the parsed token. For example,

```
>>> def convert_hex(x):
...   return eval('0x' + x[0])
>>> hex_num = hex_num.setParseAction(convert_hex).setResultsName('num')
>>> result = hex_num.parseString('1f')
>>> print result.num
31
```

As you can see, this sort of parse function allows you to convert parse results into objects automagically (after all, there's no reason that `convert_hex` needs to return an integer; it could return an object of any type).

Defining convenient and re-usable parse objects

This brings us to a "putting it all together" moment: suppose that you have some kind of string that turns up throughout the string you're parsing. For the parser I was working on, one common string was an expectation value (i.e. a floating point number). This was no *ordinary* floating point number, though: it *could* start with 'e', which meant that you needed to prepend a '1'. For example,

```
1.0  ==>  1.0
1e-5 ==>  1e-5
e-5  ==>  1e-5
```

Well, the first obvious way to handle this is like so:

```
>>> from pyparsing import Word, nums
>>> e_val = Word(nums + "e-")
>>> def convert_eval(x):
...    e = x[0]
...    if e.startswith('e'): e = '1' + e
...    return float(e)
>>> e_val = e_val.setParseAction(convert_eval)
>>> e_val.parseString('e-5')
([1.0000000000000001e-05], {})
```

OK, that's acceptable, but ugly if you have lots of e_val's floating around.

You could refine things by naming your expectation values when they occur:

```
>>> e_val_1 = e_val.setResultsName('e_val_1')
>>> e_val_2 = e_val.setResultsName('e_val_2')
>>> results = (e_val_1 + e_val_2).parseString('1e-3 5.0')
>>> results.e_val_1
0.001
>>> results.e_val_2
5.0
```

...but that's still a lot of code. Here's the suggestion that Paul made to me when he first saw my hacked-together parser:

```
>>> e_val = Word(nums + "e-")
>>> def named_e_val(name):
...    x = Word(nums + "e-").setParseAction(convert_eval).copy()
...    x = x.setResultsName(name)
...    return x
```

Now I can just say

```
>>> grammar = named_e_val('e_val_1') + named_e_val('e_val_2')
>>> results = grammar.parseString('1e-3 5.0')
>>> results.e_val_1
0.001
>>> results.e_val_2
5.0
```

2.5.2 Building a BLAST output parser

Now on to my real problem: building an output parser for NCBI BLAST.

Briefly, NCBI BLAST is a very widely used sequence search algorithm, and it has a notoriously annoying set of output formats. The most annoying thing about the output is that only the human-intended output contains a full set of information, so you need to parse something that has been formatted for humans.

Another annoying thing about the output format is that it changes regularly in subtle ways. This means that most BLAST parsers break at least once a year.

Why did I choose pyparsing?

I chose pyparsing for this project partly because I am using it for twill, and it worked quite well there. However, the main decision point was that I needed to make a *readable* and *maintainable* parser, so that down the road I wouldn't have to relearn all sorts of nasty syntax in order to update the parser when NCBI changed their output formats. pyparsing seemed to fit that bill.

What kind of output do I need to deal with?

So, how bad is the output? Well, here's an example:

```
>ref|NP_598432.1| U2 small nuclear ribonucleoprotein auxiliary factor (U2AF) 2 [Mus
           musculus]
          Length = 306

 Score =  394 bits (1013), Expect = e-110
 Identities = 202/279 (72%), Positives = 222/279 (79%)
 Frame = -1

Query: 888 FLNNQMKLAGLAQAPGNPVLAVQITWDKNFSSLEFRSVDETTQALAFDGIIFQGQSLKLR 709
            F N QM+L GL QAPGNPVLAVQI  DKNF+ LEFRSVDETTQA+AFDGIIFQGQSLK+R
Sbjct: 3   FFNAQMRLGGLTQAPGNPVLAVQINQDKNFAFLEFRSVDETTQAMAFDGIIFQGQSLKIR 62

Query: 708 RPHDYQPLPGMSESPALHVPVGVVSTVVQDTPHKLFIGGLPSYLTDDQVKELLTSFGPLK 529
            RPHDYQPLPGMSE+P+++VP GVVSTVV D+ HKLFIGGLP+YL DDQVKELLTSFGPLK
Sbjct: 63  RPHDYQPLPGMSENPSVYVP-GVVSTVVPDSAHKLFIGGLPNYLNDDQVKELLTSFGPLK 121

Query: 528 AFNLVKDSATCFSKGYAFCEYADVNVTDQAIAGLNGMQLGDKKLIVQRASVGAKNANXXX 349
            AFNLVKDSAT  SKGYAFCEY D+NVTDQAIAGLNGMQLGDKKL+VQRASVGAKNA
Sbjct: 122 AFNLVKDSATGLSKGYAFCEYVDINVTDQAIAGLNGMQLGDKKLLVQRASVGAKNATLST 181

Query: 348 XXXXXXXXXXPGLASSQVQHSGLPTEVLCLMNMVTPXXXXXXXXXXXXXXXXXXECGKYG 169
                      PGL SSQVQ  G PTEVLCLMNMV P                  EC KYG
Sbjct: 182 INQTPVTLQVPGLMSSQVQMGGHPTEVLCLMNMVLPEELLDDEEYEEIVEDVRDECSKYG 241

Query: 168 SVRSVEIPRPVNGLDIPGCGKIFVEFASLLDCQRAQQAL 52
             V+S+EIPRPV+G+++PGCGKIFVEF S+ DCQ+A Q L
Sbjct: 242 LVKSIEIPRPVDGVEVPGCGKIFVEFTSVFDCQKAMQGL 280
```

Lots of finicky things to parse in there, eh? Let's focus on the score:

```
Score =  394 bits (1013), Expect = e-110
Identities = 202/279 (72%), Positives = 222/279 (79%)
Frame = -1
```

My first iteration

Here's my first set of code:

```
self.score = Literal("Score =").suppress() +
 Word(nums + ".").setParseAction(make_float).setResultsName('bits') +
        Literal("bits (").suppress() +
 Word(nums).setParseAction(make_int).setResultsName('bits_max') +
        Literal("),").suppress() +
        Word("Expect()" + nums).suppress() + Literal("=") +
 Word(e_val).setParseAction(make_float).setResultsName('expect') +
        Literal("Identities =").suppress() + identities +
        Optional(Literal("Positives =").suppress() + positives) +
        Optional(Literal("Gaps =").suppress() + gaps) +
        Optional((Literal("Frame =").suppress() +
        Word(frame).setParseAction(make_int).setResultsName('frame1') +
        Optional(Literal("/").suppress() +
 Word(frame).setParseAction(make_int).setResultsName('frame2'))) |
        (Literal("Strand =") + restOfLine))
```

What can I say? It worked...

What it looked like after Paul's suggestions

I sent the above on to Paul, and after some gagging, he sent me back a bunch of suggestions. I ended up with this:

```
self.score = Literal("Score =") + named_float('bits') +
        "bits (" + named_int('bits_max') + ")," +
        Word("Expect()" + nums) + "=" + named_float('expect') +
        "Identities =" + identities +
        Optional("Positives =" + positives) +
        Optional("Gaps =" + gaps) +
        Optional(("Frame =" + named_frame('frame1') +
          Optional("/" + named_frame('frame2'))) |
                 ("Strand =" + restOfLine))
```

This is clearly much friendler to read!

I would also like to note that this kind of refactoring is tricky to do without being able to test each subset of the parse grammar. pyparsing let me break the parse grammar down into subsets in a very nice and convenient way, which really helped with testing.

A sort of conclusion

My parsing code is basically a generator wrapped around pyparsing results; here's what the `blastparser` API looks like:

```
for record in parse_file('blast_output.txt'):
   print '-', record.query_name, record.database.name
   for hit in record.hits:
      print '--', hit.subject_name, hit.subject_length
      print '  ', hit.total_score, hit.total_expect
      for submatch in hit:
         print submatch.expect, submatch.bits

         print submatch.query_sequence
         print submatch.alignment
         print submatch.subject_sequence
```

Because I use parse actions to turn each block into an object, it's really a very thin layer.

Future thoughts

Speed. Speed speed speed speed. How can I speed things up?

Without profiling, I'm not sure where the bottlenecks are, and that should probably be my first step. Nonetheless, I'm planning to try out lazy evaluation, which would work something like this.

First, define the block structure:

```
>>> from pyparsing import SkipTo
>>> complex_block = """
... SOMETHING
... SOMETHING ELSE
... END
... MORE STUFF THAT MATTERS
... """
>>> end_marker = SkipTo("END", include=True).suppress()
```

Build a grammar for the internal structure:

```
>>> internal_grammar = "..."
```

Define a lazy evaluation class:

```
>>> class LazyParse:
...     def __init__(self, text):
...         self.text = text
...         self.parsed = None
...     def parse(self):
...         if self.parsed:
...             return self.parsed
...         self.parsed = internal_grammar.parseString(self.text)
...         return self.parsed
```

Then, set a parse action:

```
>>> def parse_complex_block(x):
...    return LazyParse(x[0])
>>> end_marker = end_marker.setParseAction(parse_complex_block)
```

and Bob's your uncle... but I haven't gotten all the mechanics worked out.

2.6 Online Resources for Python

The obvious one: http://www.python.org/ (including, of course, http://docs.python.org/).

The next most obvious one: comp.lang.python.announce / python-announce. This is a low traffic list that is really quite handy; note especially a brief summary of postings called "the Weekly Python-URL", which as far as I can tell is only available on this list.

The Python Cookbook is chock full of useful recipes; some of them have been extracted and prettified in the O'Reilly Python Cookbook book, but they're all available through the Cookbook site.

The Daily Python-URL is distinct from the Weekly Python-URL; read it at http://www.pythonware.com/daily/. Postings vary from daily to weekly.

http://planet.python.org and http://www.planetpython.org/ are Web sites that aggregate Python blogs (mine included, hint hint). Very much worth skimming over a coffee break.

And, err, Google is a fantastic way to figure stuff out!

CHAPTER 3

Day 3

Contents:

3.1 Wrapping C/C++ for Python

There are a number of options if you want to wrap existing C or C++ functionality in Python.

3.1.1 Manual wrapping

If you have a relatively small amount of C/C++ code to wrap, you can do it by hand. The Extending and Embedding section of the docs is a pretty good reference.

When I write wrappers for C and C++ code, I usually provide a procedural interface to the code and then use Python to construct an object-oriented interface. I do things this way for two reasons: first, exposing C++ objects to Python is a pain; and second, I prefer writing higher-level structures in Python to writing them in C++.

Let's take a look at a basic wrapper: we have a function 'hello' in a file 'hello.c'. 'hello' is defined like so:

```
char * hello(char * what)
```

To wrap this manually, we need to do the following.

First, write a Python-callable function that takes in a string and returns a string.

```
static PyObject * hello_wrapper(PyObject * self, PyObject * args)
{
  char * input;
  char * result;
  PyObject * ret;

  // parse arguments
  if (!PyArg_ParseTuple(args, "s", &input)) {
    return NULL;
  }

  // run the actual function
  result = hello(input);

  // build the resulting string into a Python object.
  ret = PyString_FromString(result);
  free(result);
  return ret;
}
```

Second, register this function within a module's symbol table (all Python functions live in a module, even if they're actually C functions!)

```
static PyMethodDef HelloMethods[] = {
 { "hello", hello_wrapper, METH_VARARGS, "Say hello" },
 { NULL, NULL, 0, NULL }
};
```

Third, write an init function for the module (all extension modules require an init function).

```
DL_EXPORT(void) inithello(void)
{
  Py_InitModule("hello", HelloMethods);
}
```

Fourth, write a setup.py script:

```
from distutils.core import setup, Extension

# the c++ extension module
extension_mod = Extension("hello", ["hellomodule.c", "hello.c"])

setup(name = "hello", ext_modules=[extension_mod])
```

There are two aspects of this code that are worth discussing, even at this simple level.

First, error handling: note the PyArg_ParseTuple call. That call is what tells Python that the 'hello' wrapper function takes precisely one argument, a string ("s" means "string"; "ss" would mean "two strings"; "si" would mean "string and integer"). The convention in the C API to Python is that a NULL return from a function that returns PyObject* indicates an error has occurred; in this case, the error information is set within PyArg_ParseTuple and we're just passing the error on up the stack by returning NULL.

Second, references. Python works on a system of reference counting: each time a function "takes ownership" of an object (by, for example, assigning it to a list, or a dictionary) it increments that object's reference count by one using Py_INCREF. When the object is removed from use in that particular place (e.g. removed from the list or dictionary), the reference count is decremented with Py_DECREF. When the reference count reaches 0, Python knows that this object is not being used by anything and can be freed (it may not be freed immediately, however).

Why does this matter? Well, we're creating a PyObject in this code, with PyString_FromString. Do we need to INCREF it? To find out, go take a look at the documentation for PyString_FromString:

> http://docs.python.org/api/stringObjects.html#l2h-461

See where it says "New reference"? That means it's handing back an object with a reference count of 1, and that's what we want. If it had said "Borrowed reference", then we would need to INCREF the object before returning it, to indicate that we wanted the allocated memory to survive past the end of the function.

Here's a way to think about references:

- if you receive a Python object from the Python API, you can use it within your own C code without INCREFing it.
- if you want to guarantee that the Python object survives past the end of your own C code, you must INCREF it.
- if you received an object from Python code and it was a new reference, but you don't want it to survive past the end of your own C code, you should DECREF it.

If you wanted to return None, by the way, you can use Py_None. Remember to INCREF it!

Another note: during the class, I talked about using PyCObjects to pass opaque C/C++ data types around. This is useful if you are using Python to organize your code, but you have complex structures that you don't need to be Python-accessible. You can wrap pointers in PyCObjects (with an associated destructor, if so desired) at which point they become opaque Python objects whose memory is managed by the Python interpreter. You can see an example in the example code, under `code/hello/hellmodule.c`, functions `cobj_in`, `cobj_out`, and `free_my_struct`, which pass an allocated C structure back to Python using a PyCObject wrapper.

So that's a brief introduction to how you wrap things by hand.

As you might guess, however, there are a number of projects devoted to automatically wrapping code. Here's a brief introduction to some of them.

3.1.2 Wrapping Python code with SWIG

SWIG stands for "Simple Wrapper Interface Generator", and it is capable of wrapping C in a large variety of languages. To quote, "SWIG is used with different types of languages including common scripting languages such as Perl, PHP, Python, Tcl, Ruby and PHP. The list of supported languages also includes non-scripting languages such as C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), Java, Modula-3 and OCAML. Also several interpreted and compiled Scheme implementations (Guile, MzScheme, Chicken) are supported."

Whew.

But we only care about Python for now!

SWIG is essentially a macro language that groks C code and can spit out wrapper code for your language of choice.

You'll need three things for a SWIG wrapping of our 'hello' program. First, a Makefile:

```
all:
	swig -python -c++ -o _swigdemo_module.cc swigdemo.i
	python setup.py build_ext --inplace
```

This shows the steps we need to run: first, run SWIG to generate the C code extension; then run `setup.py build` to actually build it.

Second, we need a SWIG wrapper file, 'swigdemo.i'. In this case, it can be pretty simple:

```
%module swigdemo

%{
#include <stdlib.h>
#include "hello.h"
%}

%include "hello.h"
```

A few things to note: the %module specifies the name of the module to be generated from this wrapper file. The code between the %{ %} is placed, verbatim, in the C output file; in this case it just includes two header files. And, finally, the last line, %include, just says "build your interface against the declarations in this header file".

OK, and third, we will need a setup.py. This is virtually identical to the setup.py we wrote for the manual wrapping:

```python
from distutils.core import setup, Extension

extension_mod = Extension("_swigdemo", ["_swigdemo_module.cc", "hello.c"])

setup(name = "swigdemo", ext_modules=[extension_mod])
```

Now, when we run 'make', swig will generate the _swigdemo_module.cc file, as well as a 'swigdemo.py' file; then, setup.py will compile the two C files together into a single shared library, '_swigdemo', which is imported by swigdemo.py; then the user can just 'import swigdemo' and have direct access to everything in the wrapped module.

Note that swig can wrap most simple types "out of the box". It's only when you get into your own types that you will have to worry about providing what are called "typemaps"; I can show you some examples.

I've also heard (from someone in the class) that SWIG is essentially not supported any more, so buyer beware. (I will also say that SWIG is pretty crufty. When it works and does exactly what you want, your life is good. Fixing bugs in it is messy, though, as is adding new features, because it's a template language, and hence many of the constructs are ad hoc.)

3.1.3 Wrapping C code with pyrex

pyrex, as I discussed yesterday, is a weird hybrid of C and Python that's meant for generating fast Python-esque code. I'm not sure I'd call this "wrapping", but ... here goes.

First, write a .pyx file; in this case, I'm calling it 'hellomodule.pyx', instead of 'hello.pyx', so that I don't get confused with 'hello.c'.

```
cdef extern from "hello.h":
    char * hello(char *s)

def hello_fn(s):
    return hello(s)
```

What the 'cdef' says is, "grab the symbol 'hello' from the file 'hello.h"'. Then you just go ahead and define your 'hello_fn' as you would if it were Python.

and... that's it. You've still got to write a setup.py, of course:

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

setup(
  name = "hello",
  ext_modules=[ Extension("hellomodule", ["hellomodule.pyx", "hello.c"]) ],
  cmdclass = {'build_ext': build_ext}
)
```

but then you can just run 'setup.py build_ext –inplace' and you'll be able to 'import hellomodule; hellomodule.hello_fn'.

3.1.4 ctypes

In Python 2.5, the ctypes module is included. This module lets you talk directly to shared libraries on both Windows and UNIX, which is pretty darned handy. But can it be used to call our C code directly?

The answer is yes, with a caveat or two.

First, you need to compile 'hello.c' into a shared library.

```
gcc -o hello.so -shared -fPIC hello.c
```

Then, you need to tell the system where to find the shared library.

```
export LD_LIBRARY_PATH=.
```

Now you can load the library with ctypes:

```python
from ctypes import cdll

hello_lib = cdll.LoadLibrary("hello.so")
hello = hello_lib.hello
```

So far, so good – now what happens if you run it?

```
>> print hello("world")
136040696
```

Whoops! You still need to tell Python/ctypes what kind of return value to expect! In this case, we're expecting a char pointer:

```python
from ctypes import c_char_p
hello.restype = c_char_p
```

And now it will work:

>> print hello("world") hello, world

Voila!

I should say that ctypes is not intended for this kind of wrapping, because of the whole LD_LIBRARY_PATH setting requirement. That is, it's really intended for accessing *system* libraries. But you can still use it for other stuff like this.

3.1.5 SIP

SIP is the tool used to generate Python bindings for Qt (PyQt), a graphics library. However, it can be used to wrap any C or C++ API.

As with SWIG, you have to start with a definition file. In this case, it's pretty easy: just put this in 'hello.sip':

```
%CModule hellomodule 0

char * hello(char *);
```

Now you need to write a 'configure' script:

```python
import os
import sipconfig

# The name of the SIP build file generated by SIP and used by the build
# system.
build_file = "hello.sbf"

# Get the SIP configuration information.
config = sipconfig.Configuration()

# Run SIP to generate the code.
os.system(" ".join([config.sip_bin, "-c", ".", "-b", build_file, "hello.sip"]))

# Create the Makefile.
makefile = sipconfig.SIPModuleMakefile(config, build_file)

# Add the library we are wrapping.  The name doesn't include any platform
# specific prefixes or extensions (e.g. the "lib" prefix on UNIX, or the
# ".dll" extension on Windows).
makefile.extra_libs = ["hello"]
makefile.extra_lib_dirs = ["."]

# Generate the Makefile itself.
makefile.generate()
```

Now, run 'configure.py', and then run 'make' on the generated Makefile, and your extension will be compiled.

(At this point I should say that I haven't really used SIP before, and I feel like it's much more powerful than this example would show you!)

3.1.6 Boost.Python

If you are an expert C++ programmer and want to wrap a lot of C++ code, I would recommend taking a look at the Boost.Python library, which lets you run C++ code from Python, and Python code from C++, seamlessly. I haven't used it at all, and it's too complicated to cover in a short period!

http://www.boost-consulting.com/writing/bpl.html

3.1.7 Recommendations

Based on my little survey above, I would suggest using SWIG to write wrappers for relatively small libraries, while SIP probably provides a more manageable infrastructure for wrapping large libraries (which I know I did not demonstrate!)

Pyrex is astonishingly easy to use, and it may be a good option if you have a small library to wrap. My guess is that you would spend a lot of time converting types back and forth from C/C++ to Python, but I could be wrong.

ctypes is excellent if you have a bunch of functions to run and you don't care about extracting complex data types from them: you just want to pass around the encapsulated data types between the functions in order to accomplish a goal.

3.1.8 One or two more notes on wrapping

As I said at the beginning, I tend to write procedural interfaces to my C++ code and then use Python to wrap them in an object-oriented interface. This lets me adjust the OO structure of my code more flexibly; on the flip side, I only use the code from Python, so I really don't care what the C++ code looks like as long as it runs fast ;). So, you might find it worthwhile to invest in figuring out how to wrap things in a more object-oriented manner.

Secondly, one of the biggest benefits I find from wrapping my C code in Python is that all of a sudden I can test it pretty easily. Testing is something you *do not* want to do in C, because you have to declare all the variables and stuff that you use, and that just gets in the way of writing simple tests. I find that once I've wrapped something in Python, it becomes much more testable.

3.2 Packages for Multiprocessing

3.2.1 threading

Python has basic support for threading built in: for example, here's a program that runs two threads, each of which prints out messages after sleeping a particular amount of time:

```
from threading import Thread, local
import time

class MessageThread(Thread):
    def __init__(self, message, sleep):
        self.message = message
        self.sleep = sleep
        Thread.__init__(self)                   # remember to run Thread init!

    def run(self):                              # automatically run by 'start'
        i = 0
        while i < 50:
            i += 1
            print i, self.message

            time.sleep(self.sleep)

t1 = MessageThread("thread - 1", 1)
t2 = MessageThread("thread - 2", 2)

t1.start()
t2.start()
```

However, due to the existence of the Global Interpreter Lock (GIL) (http://docs.python.org/api/threads.html), CPU-intensive code will not run faster on dual-core CPUs than it will on single-core CPUs.

Briefly, the idea is that the Python interpreter holds a global lock, and no Python code can be executed without holding that lock. (Code execution will still be interleaved, but no two Python instructions can execute at the same time.) Therefore, any Python code that you write (or GIL-naive C/C++ extension code) will not take advantage of multiple CPUs.

This is intentional:

> http://mail.python.org/pipermail/python-3000/2007-May/007414.html

There is a long history of wrangling about the GIL, and there are a couple of good arguments for it. Briefly,

- it dramatically simplifies writing C extension code, because by default, C extension code does not need to know anything about threads.
- putting in locks appropriately to handle places where contention might occur is not only error-prone but makes the code quite slow; locks really affect performance.
- threaded code is difficult to debug, and most people don't need it, despite having been brainwashed to think that they do ;).

But we don't care about that: *we* do want our code to run on multiple CPUs. So first, let's dip back into C code: what do we have to do to make our C code release the GIL so that it can do a long computation?

Basically, just wrap I/O blocking code or CPU-intensive code in the following macros:

```
Py_BEGIN_ALLOW_THREADS

...Do some time-consuming operation...

Py_END_ALLOW_THREADS
```

This is actually pretty easy to do to your C code, and it does result in that code being run in parallel on multi-core CPUs. (note: example?)

The big problem with the GIL, however, is that it really means that you simply can't write parallel code in Python without jumping through some kind of hoop. Below, we discuss a couple of these hoops ;).

3.2.2 Writing (and indicating) threadsafe C extensions

Suppose you had some CPU-expensive C code:

```
void waste_time() {
     int i, n;
     for (i = 0; i < 1024*1024*1024; i++) {
         if ((i % 2) == 0) n++;
     }
}
```

and you wrapped this in a Python function:

```
PyObject * waste_time_fn(PyObject * self, PyObject * args) {
     waste_time();
}
```

Now, left like this, any call to `waste_time_fn` will cause all Python threads and processes to block, waiting for `waste_time` to finish. That's silly, though – `waste_time` is clearly threadsafe, because it uses only local variables!

To tell Python that you are engaged in some expensive operations that are threadsafe, just enclose the waste_time code like so:

```
PyObject * waste_time_fn(PyObject * self, PyObject * args) {
     Py_BEGIN_ALLOW_THREADS

     waste_time();

     Py_END_ALLOW_THREADS
}
```

This code will now be run in parallel when threading is used. One caveat: you can't do *any* call to the Python C API in the code between the Py_BEGIN_ALLOW_THREADS and Py_END_ALLOW_THREADS, because the Python C API is not threadsafe.

3.2.3 parallelpython

parallelpython is a system for controlling multiple Python processes on multiple machines. Here's an example program:

```
#!/usr/bin/python
def isprime(n):
    """Returns True if n is prime and False otherwise"""
    import math

    if n < 2:
        return False
    if n == 2:
        return True
    max = int(math.ceil(math.sqrt(n)))
    i = 2
    while i <= max:
        if n % i == 0:
            return False
        i += 1
    return True

def sum_primes(n):
    """Calculates sum of all primes below given integer n"""
    return sum([x for x in xrange(2, n) if isprime(x)])

####

import sys, time

import pp
# Creates jobserver with specified number of workers
job_server = pp.Server(ncpus=int(sys.argv[1]))

print "Starting pp with", job_server.get_ncpus(), "workers"

start_time = time.time()

# Submit a job of calulating sum_primes(100) for execution.
#
#    * sum_primes - the function
#    * (input,) - tuple with arguments for sum_primes
#    * (isprime,) - tuple with functions on which sum_primes depends
#
# Execution starts as soon as one of the workers will become available

inputs = (100000, 100100, 100200, 100300, 100400, 100500, 100600, 100700)

jobs = []
for input in inputs:
    job = job_server.submit(sum_primes, (input,), (isprime,))
    jobs.append(job)

for job, input in zip(jobs, inputs):
    print "Sum of primes below", input, "is", job()

print "Time elapsed: ", time.time() - start_time, "s"
job_server.print_stats()
```

If you add “ppservers=(‘host1’)” to to the line

```
pp.Server(...)
```

pp will check for parallelpython servers running on those other hosts and send jobs to them as well.

The way parallelpython works is it literally sends the Python code across the network & evaluates it there! It seems to work well.

3.2.4 Rpyc

Rpyc is a remote procedure call system built in (and tailored to) Python. It is basically a way to transparently control remove Python processes. For example, here’s some code that will connect to an Rpyc server and ask the server to calculate the first 500 prime numbers:

> from Rpyc import SocketConnection
>
> # connect to the “remote” server c = SocketConnection(“localhost”)
>
> # make sure it has the right code in its path c.modules.sys.path.append(‘/u/t/dev/misc/rpyc’)
>
> # tell it to execute ‘primestuff.get_n_primes’ primes = c.modules.primestuff.get_n_primes(500) print primes[-20:]

Note that this is a synchronous connection, so the client waits for the result; you could also have it do the computation asynchronously, leaving the client free to request results from other servers.

In terms of parallel computing, the server has to be controlled directly, which makes it less than ideal. I think parallelpython is a better choice for straightforward number crunching.

3.2.5 pyMPI

pyMPI is a nice Python implementation to the MPI (message-passing interface) library. MPI enables different processors to communicate with each other. I can't demo pyMPI, because I couldn't get it to work on my other machine, but here's some example code that computs pi to a precision of 1e-6 on however many machines you have running MPI.

```
import random
import mpi

def computePi(nsamples):
    rank, size = mpi.rank, mpi.size
    oldpi, pi, mypi = 0.0,0.0,0.0

    done = False
    while(not done):
        inside = 0
        for i in xrange(nsamples):
            x = random.random()
            y = random.random()
            if ((x*x)+(y*y)<1):
                inside+=1

        oldpi = pi
        mypi = (inside * 1.0)/nsamples
        pi =  (4.0 / mpi.size) * mpi.allreduce(mypi, mpi.SUM)

        delta = abs(pi - oldpi)
        if(mpi.rank==0):
            print "pi:",pi," - delta:",delta
        if(delta < 0.00001):
            done = True
    return pi

if __name__=="__main__":
    pi = computePi(10000)
    if(mpi.rank==0):
        print "Computed value of pi on",mpi.size,"processors is",pi
```

One big problem with MPI is that documentation is essentially absent, but I can still make a few points ;).

First, the "magic" happens in the 'allreduce' function up above, where it sums the results from all of the machines and then divides by the number of machines.

Second, pyMPI takes the unusual approach of actually building an MPI-aware Python interpreter, so instead of running your scripts in normal Python, you run them using 'pyMPI'.

3.2.6 multitask

multitask is not a multi-machine mechanism; it's a library that implements cooperative multitasking around I/O operations. Briefly, whenever you're going to do an I/O operation (like wait for more data from the network) you can tell multitask to yield to another thread of control. Here is a simple example where control is voluntarily yielded after a 'print':

```
import multitask

 def printer(message):
     while True:
         print message
         yield

 multitask.add(printer('hello'))
 multitask.add(printer('goodbye'))
 multitask.run()
```

Here's another example from the home page:

```
import multitask

def listener(sock):
    while True:
        conn, address = (yield multitask.accept(sock))     # WAIT
        multitask.add(client_handler(conn))

def client_handler(sock):
    while True:
        request = (yield multitask.recv(sock, 1024))       # WAIT
        if not request:
            break
        response = handle_request(request)
        yield multitask.send(sock, response)               # WAIT

multitask.add(listener(sock))
multitask.run()
```

3.3 Useful Packages

3.3.1 subprocess

'subprocess' is a new addition (Python 2.4), and it provides a convenient and powerful way to run system commands. (...and you should use it instead of os.system, commands.getstatusoutput, or any of the Popen modules).

Unfortunately subprocess is a bi hard to use at the moment; I'm hoping to help fix that for Python 2.6, but in the meantime here are some basic commands.

Let's just try running a system command and retrieving the output:

```
>>> import subprocess
>>> p = subprocess.Popen(['/bin/echo', 'hello, world'], stdout=subprocess.PIPE)
>>> (stdout, stderr) = p.communicate()
>>> print stdout,
hello, world
```

What's going on is that we're starting a subprocess (running '/bin/echo hello, world') and then asking for all of the output aggregated together.

We could, for short strings, read directly from p.stdout (which is a file handle):

```
>>> p = subprocess.Popen(['/bin/echo', 'hello, world'], stdout=subprocess.PIPE)
>>> print p.stdout.read(),
hello, world
```

but you could run into trouble here if the command returns a lot of data; you should use communicate to get the output instead.

Let's do something a bit more complicated, just to show you that it's possible: we're going to write to 'cat' (which is basically an echo chamber):

```
>>> from subprocess import PIPE
>>> p = subprocess.Popen(["/bin/cat"], stdin=PIPE, stdout=PIPE)
>>> (stdout, stderr) = p.communicate('hello, world')
>>> print stdout,
hello, world
```

There are a number of more complicated things you can do with subprocess – like interact with the stdin and stdout of other processes – but they are fraught with peril.

3.3.2 rpy

rpy is an extension for R that lets R and Python talk naturally. For those of you that have never used R, it's a very nice package that's mainly used for statistics, and it has *tons* of libraries.

To use rpy, just

```
from rpy import *
```

The most important symbol that will be imported is 'r', which lets you run arbitrary R comments:

```
r("command")
```

For example, if you wanted to run a principle component analysis, you could do it like so:

```
from rpy import *

def plot_pca(filename):
    r("""data <- read.delim('%s', header=FALSE, sep=" ", nrows=5000)""" \
      % (filename,))

    r("""pca <- prcomp(data, scale=FALSE, center=FALSE)""")
    r("""pairs(pca$x[,1:3], pch=20)""")

plot_pca('vectors.txt')
```

Now, the problem with this code is that I'm really just using Python to drive R, which seems inefficient. You *can* go access the data directly if you want; I'm just using R's loading features directly because they're faster. For example,

x = r.pca['x']

is equivalent to 'x <- pca$x'.

3.3.3 matplotlib

matplotlib is a plotting package that aims to make "simple things easy, and hard things possible". It's got a fair amount of matlab compatibility if you're into that.

Simple example:

```
x = [ i**2 for i in range(0, 500) ]
hist(x, 100)
```

3.4 Idiomatic Python Take 3: new-style classes

Someone (Lila) asked me a question about pickling and memory usage that led me on a chase through google, and along the way I was reminded that new-style classes do have one or two interesting points.

You may remember from the first day that there was a brief discussion of new-style classes. Basically, they're classes that inherit from 'object' explicitly:

```
>>> class N(object):
...   pass
```

and they have a bunch of neat features (covered here in detail). I'm going to talk about two of them: __slots__ and descriptors.

__slots__ are a memory optimization. As you know, you can assign any attribute you want to an object:

```
>>> n = N()
>>> n.test = 'some string'
>>> print n.test
some string
```

Now, the way this is implemented behind the scenes is that there's a dictionary hiding in 'n' (called 'n.__dict__') that holds all of the attributes. However, dictionaries consume a fair bit of memory above and beyond their contents, so it might be good to get rid of the dictionary in some circumstances and specify precisely what attributes a class has.

You can do that by creating a __slots__ entry:

```
>>> class M(object):
...   __slots__ = ['x', 'y', 'z']
```

Now objects of type 'M' will contain only enough space to hold those three attributes, and nothing else.

A side consequence of this is that you can no longer assign to arbitrary attributes, however!

```
>>> m = M()
>>> m.x = 5
>>> m.a = 10
Traceback (most recent call last):
  ...
AttributeError: 'M' object has no attribute 'a'
```

This will look strangely like some kind of type declaration to people familiar with B&D languages, but I assure you that it is not! You are supposed to use __slots__ only for memory optimization...

Speaking of memory optimization (which is what got me onto this in the first place) apparently using new-style classes and __slots__ both dramatically decrease memory consumption:

> http://mail.python.org/pipermail/python-list/2004-November/291840.html
>
> http://mail.python.org/pipermail/python-list/2004-November/291986.html

3.4.1 Managed attributes

Another nifty pair of features in new-style classes are managed attributes and descriptors.

You may know that in the olden days, you could intercept attribute access by overwriting __getattr__:

```
>>> class A:
...     def __getattr__(self, name):
...         if name == 'special':
...            return 5
...         return self.__dict__[name]
>>> a = A()
>>> a.special
5
```

This turns out to be kind of inefficient, because *every* attribute access now goes through __getattr__. Plus, it's a bit ugly and it can lead to buggy code.

Python 2.2 introduced "managed attributes". With managed attributes, you can *define* functions that handle the get, set, and del operations for an attribute:

```
>>> class B(object):
...    def _get_special(self):
...        return 5
...    special = property(_get_special)
>>> b = B()
>>> b.special
5
```

If you wanted to provide a '_set_special' function, you could do some really bizarre things:

```
>>> class B(object):
...    def _get_special(self):
...       return 5
...    def _set_special(self, value):
...       print 'ignoring', value
...    special = property(_get_special, _set_special)
>>> b = B()
```

Now, retrieving the value of the 'special' attribute will give you '5', no matter what you set it to:

```
>>> b.special
5
>>> b.special = 10
ignoring 10
>>> b.special
5
```

Ignoring the array of perverse uses you could apply, this is actually useful – for one example, you can now do internal consistency checking on attributes, intercepting inconsistent values before they actually get set.

3.4.2 Descriptors

Descriptors are a related feature that let you implement attribute access functions in a different way. First, let's define a read-only descriptor:

```
>>> class D(object):
...    def __get__(self, obj, type=None):
...       print 'in get:', obj
...       return 6
```

Now attach it to a class:

```
>>> class A(object):
...   val = D()
```

```
>>> a = A()
>>> a.val
in get: <A object at ...>
6
```

What happens is that 'a.val' is checked for the presence of a __get__ function, and if such exists, it is called instead of returning 'val'. You can also do this with __set__ and __delete__:

```
>>> class D(object):
...   def __get__(self, obj, type=None):
...      print 'in get'
...      return 6
...
...   def __set__(self, obj, value):
...      print 'in set:', value
...
...   def __delete__(self, obj):
...      print 'in del'
```

```
>>> class A(object):
...   val = D()
>>> a = A()
>>> a.val = 15
in set: 15
>>> del a.val
in del
>>> print a.val
in get
6
```

This can actually give you control over things like the *types* of objects that are assigned to particular classes: no mean thing.

3.5 GUI Gossip

Tkinter

- fairly primitive;
- but: comes with every Python install!
- still a bit immature (feb 2007) for Mac OS X native ("Aqua"); X11 version works fine on OS X.

PyQT (http://www.riverbankcomputing.com/software/pyqt/intro)

- mature;
- cross platform;
- freely available for Open Source Software use;
- has a testing framework!

KWWidgets (http://www.kwwidgets.org/)

- immature; based on Tk, so Mac OS X native is still a bit weak;
- lightweight;

- attractive;
- has a testing framework!

pyFLTK (http://sf.net/projects/pyfltk/)

- cross platform;
- FLTK is mature, although primitive;
- not very pretty;
- very lightweight;

wxWindows (http://www.wxwindows.org/)

- cross platform;
- mature?; looks good.
- no personal or "friend" experience;
- try reading http://www.ibm.com/developerworks/library/l-wxwin.html

pyGTK (http://www.pygtk.org/)

- cross platform;
- mature; looks good.
- no personal or "friend" experience;
- UI designer;

Mild recommendation: start with Qt, which is apparently very mature and very powerful.

3.6 Python 3.0

What's coming in Python 3000 (a.k.a. Python 3.0)?

First of all, Python 3000 will be out sometime in 2008; large parts of it have already been implemented. It is simply an increment on the current code base.

The biggest point is that Python 3000 will break backwards compatibility, abruptly. This is very unusual for Python, but is necessary to get rid of some old cruft. In general, Python has been very good about updating slowly and incrementally without breaking backwards compatibility very much; this approach is being abandoned for the jump from 2.x to 3.x.

However, the actual impact of this is likely to be small. There will be a few expressions that no longer work – for example, 'dict.has_key' is being removed, because you can just do 'if key in dict' – but a lot of the changes are behind the scenes, e.g. functions that currently return lists will return iterators (dict.iterkeys -> dict.keys).

The biggest impact on this audience (scientists & numerical people) is probably that (in Python 3.0) 6 / 5 will no longer be 0, and <> is being removed.

Where lots of existing code must be made Python 3k compatible, you will be able to use an automated conversion tool. This should "just work" except for cases where there is ambiguity in intent.

The most depressing aspect of Py3k (for me) is that the stdlib is not being reorganized, but this does mean that most existing modules will still be in the same place!

See David Mertz's blog for his summary of the changes.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search