What's New in Python 2.2

Release 1.00

A.M. Kuchling

October 14, 2002

akuchlin@mems-exchange.org

Contents

1 Introduction

This article explains the new features in Python 2.2, released on December 21, 2001.

Python 2.2 can be thought of as the ”cleanup release”. There are some features such as generators and iterators that are completely new, but most of the changes, significant and far-reaching though they may be, are aimed at cleaning up irregularities and dark corners of the language design.

This article doesn't attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.2, such as the *Python Library Reference* and the *Python Reference Manual*. If you want to understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature.

See Also:

http://www.unixreview.com/documents/s=1356/urm0109h/0109h.htm
"What's So Special About Python 2.2?" is also about the new 2.2 features, and was written by Cameron Laird and Kathryn Soraiz.

2 PEPs 252 and 253: Type and Class Changes

The largest and most far-reaching changes in Python 2.2 are to Python's model of objects and classes. The changes should be backward compatible, so it's likely that your code will continue to run unchanged, but the changes provide some amazing new capabilities. Before beginning this, the longest and most complicated section of this article, I'll provide an overview of the changes and offer some comments.

A long time ago I wrote a Web page (http://www.amk.ca/python/writing/warts.html) listing flaws in Python's design. One of the most significant flaws was that it's impossible to subclass Python types implemented in C. In particular, it's not possible to subclass built-in types, so you can't just subclass, say, lists in order to add a single useful method to them. The `UserList` module provides a class that supports all of the methods of lists and that can be subclassed further, but there's lots of C code that expects a regular Python list and won't accept a `UserList` instance.

Python 2.2 fixes this, and in the process adds some exciting new capabilities. A brief summary:

- You can subclass built-in types such as lists and even integers, and your subclasses should work in every place that requires the original type.
- It's now possible to define static and class methods, in addition to the instance methods available in previous versions of Python.
- It's also possible to automatically call methods on accessing or setting an instance attribute by using a new mechanism called *properties*. Many uses of `__getattr__` can be rewritten to use properties instead, making the resulting code simpler and faster. As a small side benefit, attributes can now have docstrings, too.
- The list of legal attributes for an instance can be limited to a particular set using *slots*, making it possible to safeguard against typos and perhaps make more optimizations possible in future versions of Python.

Some users have voiced concern about all these changes. Sure, they say, the new features are neat and lend themselves to all sorts of tricks that weren't possible in previous versions of Python, but they also make the language more complicated. Some people have said that they've always recommended Python for its simplicity, and feel that its simplicity is being lost.

Personally, I think there's no need to worry. Many of the new features are quite esoteric, and you can write a lot of Python code without ever needed to be aware of them. Writing a simple class is no more difficult than it ever was, so you don't need to bother learning or teaching them unless they're actually needed. Some very complicated tasks that were previously only possible from C will now be possible in pure Python, and to my mind that's all for the better.

I'm not going to attempt to cover every single corner case and small change that were required to make the new features work. Instead this section will paint only the broad strokes. See section 2.5, "Related Links", for further sources of information about Python 2.2's new object model.

2.1 Old and New Classes

First, you should know that Python 2.2 really has two kinds of classes: classic or old-style classes, and new-style classes. The old-style class model is exactly the same as the class model in earlier versions of Python. All the new features described in this section apply only to new-style classes. This divergence isn't intended to last forever; eventually old-style classes will be dropped, possibly in Python 3.0.

So how do you define a new-style class? You do it by subclassing an existing new-style class. Most of Python's built-in types, such as integers, lists, dictionaries, and even files, are new-style classes now. A new-style class named `object`, the base class for all built-in types, has been also been added so if no built-in type is suitable, you can just subclass `object`:

```
class C(object):
    def __init__ (self):
        ...
    ...
```

This means that `class` statements that don't have any base classes are always classic classes in Python 2.2. (Actually you can also change this by setting a module-level variable named `__metaclass__` — see PEP 253 for the details — but it's easier to just subclass `object`.)

The type objects for the built-in types are available as built-ins, named using a clever trick. Python has always had built-in functions named `int()`, `float()`, and `str()`. In 2.2, they aren't functions any more, but type objects that behave as factories when called.

```
>>> int
<type 'int'>
>>> int('123')
123
```

To make the set of types complete, new type objects such as `dict` and `file` have been added. Here's a more interesting example, adding a `lock()` method to file objects:

```
class LockableFile(file):
    def lock (self, operation, length=0, start=0, whence=0):
        import fcntl
        return fcntl.lockf(self.fileno(), operation,
                           length, start, whence)
```

The now-obsolete `posixfile` module contained a class that emulated all of a file object's methods and also added a `lock()` method, but this class couldn't be passed to internal functions that expected a built-in file, something which is possible with our new `LockableFile`.

2.2 Descriptors

In previous versions of Python, there was no consistent way to discover what attributes and methods were supported by an object. There were some informal conventions, such as defining `__members__` and `__methods__` attributes that were lists of names, but often the author of an extension type or a class wouldn't bother to define them. You could fall back on inspecting the `__dict__` of an object, but when class inheritance or an arbitrary `__getattr__` hook were in use this could still be inaccurate.

The one big idea underlying the new class model is that an API for describing the attributes of an object using *descriptors* has been formalized. Descriptors specify the value of an attribute, stating whether it's a method or a field. With the descriptor API, static methods and class methods become possible, as well as more exotic constructs.

Attribute descriptors are objects that live inside class objects, and have a few attributes of their own:

- `__name__` is the attribute's name.
- `__doc__` is the attribute's docstring.
- `__get__(`*object*`)` is a method that retrieves the attribute value from *object*.
- `__set__(`*object*`, `*value*`)` sets the attribute on *object* to *value*.
- `__delete__(`*object*`, `*value*`)` deletes the *value* attribute of *object*.

For example, when you write `obj.x`, the steps that Python actually performs are:

```
descriptor = obj.__class__.x
descriptor.__get__(obj)
```

For methods, `descriptor.__get__` returns a temporary object that's callable, and wraps up the instance and the method to be called on it. This is also why static methods and class methods are now possible; they have descriptors that wrap up just the method, or the method and the class. As a brief explanation of these new kinds of methods, static methods aren't passed the instance, and therefore resemble regular functions. Class methods are passed the class of the object, but not the object itself. Static and class methods are defined like this:

```
class C(object):
    def f(arg1, arg2):
        ...
    f = staticmethod(f)

    def g(cls, arg1, arg2):
        ...
    g = classmethod(g)
```

The `staticmethod()` function takes the function `f`, and returns it wrapped up in a descriptor so it can be stored in the class object. You might expect there to be special syntax for creating such methods (`def static f()`, `defstatic f()`, or something like that) but no such syntax has been defined yet; that's been left for future versions of Python.

More new features, such as slots and properties, are also implemented as new kinds of descriptors, and it's not difficult to write a descriptor class that does something novel. For example, it would be possible to write a descriptor class that made it possible to write Eiffel-style preconditions and postconditions for a method. A class that used this feature might be defined like this:

```
from eiffel import eiffelmethod

class C(object):
    def f(self, arg1, arg2):
        # The actual function
        ...
    def pre_f(self):
        # Check preconditions
        ...
    def post_f(self):
        # Check postconditions
        ...

    f = eiffelmethod(f, pre_f, post_f)
```

Note that a person using the new `eiffelmethod()` doesn't have to understand anything about descriptors. This is why I think the new features don't increase the basic complexity of the language. There will be a few wizards who need to know about it in order to write `eiffelmethod()` or the ZODB or whatever, but most users will just write code on top of the resulting libraries and ignore the implementation details.

2.3 Multiple Inheritance: The Diamond Rule

Multiple inheritance has also been made more useful through changing the rules under which names are resolved. Consider this set of classes (diagram taken from PEP 253 by Guido van Rossum):

```
          class A:
            ^ ^  def save(self): ...
           /   \
          /     \
         /       \
        /         \
    class B     class C:
        ^         ^  def save(self): ...
         \       /
          \     /
           \   /
            \ /
          class D
```

The lookup rule for classic classes is simple but not very smart; the base classes are searched depth-first, going from left to right. A reference to `D.save` will search the classes `D`, `B`, and then `A`, where `save()` would be found and returned. `C.save()` would never be found at all. This is bad, because if `C`'s `save()` method is saving some internal state specific to `C`, not calling it will result in that state never getting saved.

New-style classes follow a different algorithm that's a bit more complicated to explain, but does the right thing in this situation.

1. List all the base classes, following the classic lookup rule and include a class multiple times if it's visited repeatedly. In the above example, the list of visited classes is [`D`, `B`, `A`, `C`, `A`].
2. Scan the list for duplicated classes. If any are found, remove all but one occurrence, leaving the *last* one in the list. In the above example, the list becomes [`D`, `B`, `C`, `A`] after dropping duplicates.

Following this rule, referring to `D.save()` will return `C.save()`, which is the behaviour we're after. This lookup rule is the same as the one followed by Common Lisp. A new built-in function, `super()`, provides a way to get at a class's superclasses without having to reimplement Python's algorithm. The most commonly used form will be `super(`*class*`, `*obj*`)`, which returns a bound superclass object (not the actual class object). This form will be used in methods to call a method in the superclass; for example, `D`'s `save()` method would look like this:

```
class D:
    def save (self):
# Call superclass .save()
        super(D, self).save()
        # Save D's private information here
        ...
```

`super()` can also return unbound superclass objects when called as `super(`*class*`)` or `super(`*class1*`, `*class2*`)`, but this probably won't often be useful.

2.4 Attribute Access

A fair number of sophisticated Python classes define hooks for attribute access using `__getattr__`; most commonly this is done for convenience, to make code more readable by automatically mapping an attribute access such as `obj.parent` into a method call such as `obj.get_parent()`. Python 2.2 adds some new ways of controlling attribute access.

First, `__getattr__`(*attr_name*) is still supported by new-style classes, and nothing about it has changed. As before, it will be called when an attempt is made to access `obj.foo` and no attribute named '`foo`' is found in the instance's dictionary.

New-style classes also support a new method, `__getattribute__`(*attr_name*). The difference between the two methods is that `__getattribute__` is *always* called whenever any attribute is accessed, while the old `__getattr__` is only called if '`foo`' isn't found in the instance's dictionary.

However, Python 2.2's support for *properties* will often be a simpler way to trap attribute references. Writing a `__getattr__` method is complicated because to avoid recursion you can't use regular attribute accesses inside them, and instead have to mess around with the contents of `__dict__`. `__getattr__` methods also end up being called by Python when it checks for other methods such as `__repr__` or `__coerce__`, and so have to be written with this in mind. Finally, calling a function on every attribute access results in a sizable performance loss.

`property` is a new built-in type that packages up three functions that get, set, or delete an attribute, and a docstring. For example, if you want to define a `size` attribute that's computed, but also settable, you could write:

```
class C(object):
    def get_size (self):
        result = ... computation ...
        return result
    def set_size (self, size):
        ... compute something based on the size
        and set internal state appropriately ...

    # Define a property.  The 'delete this attribute'
    # method is defined as None, so the attribute
    # can't be deleted.
    size = property(get_size, set_size,
                    None,
                    "Storage size of this instance")
```

That is certainly clearer and easier to write than a pair of `__getattr__`/`__setattr__` methods that check for the `size` attribute and handle it specially while retrieving all other attributes from the instance's `__dict__`. Accesses to `size` are also the only ones which have to perform the work of calling a function, so references to other attributes run at their usual speed.

Finally, it's possible to constrain the list of attributes that can be referenced on an object using the new `__slots__` class attribute. Python objects are usually very dynamic; at any time it's possible to define a new attribute on an instance by just doing `obj.new_attr=1`. This is flexible and convenient, but this flexibility can also lead to bugs, as when you meant to write `obj.template = 'a'` but made a typo and wrote `obj.templtae` by accident.

A new-style class can define a class attribute named `__slots__` to constrain the list of legal attribute names. An example will make this clear:

```
>>> class C(object):
...     __slots__ = ('template', 'name')
...
>>> obj = C()
>>> print obj.template
None
>>> obj.template = 'Test'
>>> print obj.template
Test
>>> obj.templtae = None
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'C' object has no attribute 'templtae'
```

Note how you get an `AttributeError` on the attempt to assign to an attribute not listed in `__slots__`.

2.5 Related Links

This section has just been a quick overview of the new features, giving enough of an explanation to start you programming, but many details have been simplified or ignored. Where should you go to get a more complete picture?

http://www.python.org/2.2/descrintro.html is a lengthy tutorial introduction to the descriptor features, written by Guido van Rossum. If my description has whetted your appetite, go read this tutorial next, because it goes into much more detail about the new features while still remaining quite easy to read.

Next, there are two relevant PEPs, PEP 252 and PEP 253. PEP 252 is titled "Making Types Look More Like Classes", and covers the descriptor API. PEP 253 is titled "Subtyping Built-in Types", and describes the changes to type objects that make it possible to subtype built-in objects. PEP 253 is the more complicated PEP of the two, and at a few points the necessary explanations of types and meta-types may cause your head to explode. Both PEPs were written and implemented by Guido van Rossum, with substantial assistance from the rest of the Zope Corp. team.

Finally, there's the ultimate authority: the source code. Most of the machinery for the type handling is in 'Objects/typeobject.c', but you should only resort to it after all other avenues have been exhausted, including posting a question to python-list or python-dev.

3 PEP 234: Iterators

Another significant addition to 2.2 is an iteration interface at both the C and Python levels. Objects can define how they can be looped over by callers.

In Python versions up to 2.1, the usual way to make `for item in obj` work is to define a `__getitem__()` method that looks something like this:

```
def __getitem__(self, index):
    return <next item>
```

`__getitem__()` is more properly used to define an indexing operation on an object so that you can write `obj[5]` to retrieve the sixth element. It's a bit misleading when you're using this only to support `for` loops. Consider some file-like object that wants to be looped over; the *index* parameter is essentially meaningless, as the class probably assumes that a series of `__getitem__()` calls will be made with *index* incrementing by one each time. In other words, the presence of the `__getitem__()` method doesn't mean that using `file[5]` to randomly access the sixth element will work, though it really should.

In Python 2.2, iteration can be implemented separately, and `__getitem__()` methods can be limited to classes that really do support random access. The basic idea of iterators is simple. A new built-in function, `iter(obj)` or `iter(`*C*`,` *sentinel*`)`, is used to get an iterator. `iter(obj)` returns an iterator for the object *obj*, while `iter(`*C*`,` *sentinel*`)` returns an iterator that will invoke the callable object *C* until it returns *sentinel* to signal that the iterator is done.

Python classes can define an `__iter__()` method, which should create and return a new iterator for the object; if the object is its own iterator, this method can just return `self`. In particular, iterators will usually be their own iterators. Extension types implemented in C can implement a `tp_iter` function in order to return an iterator, and extension types that want to behave as iterators can define a `tp_iternext` function.

So, after all this, what do iterators actually do? They have one required method, `next()`, which takes no arguments and returns the next value. When there are no more values to be returned, calling `next()` should raise the `StopIteration` exception.

```
>>> L = [1,2,3]
>>> i = iter(L)
>>> print i
<iterator object at 0x8116870>
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

In 2.2, Python's `for` statement no longer expects a sequence; it expects something for which `iter()` will return an iterator. For backward compatibility and convenience, an iterator is automatically constructed for sequences that don't implement `__iter__()` or a `tp_iter` slot, so `for i in [1,2,3]` will still work. Wherever the Python interpreter loops over a sequence, it's been changed to use the iterator protocol. This means you can do things like this:

```
>>> L = [1,2,3]
>>> i = iter(L)
>>> a,b,c = i
>>> a,b,c
(1, 2, 3)
```

Iterator support has been added to some of Python's basic types. Calling `iter()` on a dictionary will return an iterator which loops over its keys:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m: print key, m[key]
...
Mar 3
Feb 2
Aug 8
Sep 9
May 5
Jun 6
Jul 7
Jan 1
Apr 4
Nov 11
Dec 12
Oct 10
```

That's just the default behaviour. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate iterator. In a minor related change, the `in` operator now works on dictionaries, so *key* `in dict` is now equivalent to `dict.has_key(`*key*`)`.

Files also provide an iterator, which calls the `readline()` method until there are no more lines in the file. This means you can now read each line of a file using code like this:

```
for line in file:
    # do something for each line
```

```
...
```

Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. An iterator object could provide such additional capabilities, but the iterator protocol only requires a `next()` method.

See Also:

PEP 234, "*Iterators*"
Written by Ka-Ping Yee and GvR; implemented by the Python Labs crew, mostly by GvR and Tim Peters.

4 PEP 255: Simple Generators

Generators are another new feature, one that interacts with the introduction of iterators.

You're doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren't thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here's the simplest example of a generator function:

```
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a `yield` statement is a generator function; this is detected by Python's bytecode compiler which compiles the function specially as a result. Because a new keyword was introduced, generators must be explicitly enabled in a module by including a `from __future__ import generators` statement near the top of the module's source code. In Python 2.3 this statement will become unnecessary.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `next()` method, the function will resume executing immediately after the `yield` statement. (For complicated reasons, the `yield` statement isn't allowed inside the `try` block of a `try`...`finally` statement; read PEP 255 for a full explanation of the interaction between `yield` and exceptions.)

Here's a sample usage of the `generate_ints` generator:

```
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in generate_ints
```

```
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a,b,c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values; afterwards the generator cannot return any further values. `return` with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. 'Lib/test/test_generators.py' contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in 'Lib/test/test_generators.py' produce solutions for the N-Queens problem (placing N queens on an NxN chess board so that no queen threatens another) and the Knight's Tour (a route that takes a knight to every square of an NxN chessboard without visiting any square twice).

The idea of generators comes from other programming languages, especially Icon (http://www.cs.arizona.edu/icon/), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from "An Overview of the Icon Programming Language" at http://www.cs.arizona.edu/icon/docs/ipd266.htm gives an idea of what this looks like:

```
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```

In Icon the `find()` function returns the indexes at which the substring "or" is found: 3, 23, 33. In the `if` statement, `i` is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn't go nearly as far as Icon in adopting generators as a central concept. Generators are considered a new part of the core Python language, but learning or using them isn't compulsory; if they don't solve any problems that you have, feel free to ignore them. One novel feature of Python's interface as compared to Icon's is that a generator's state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

See Also:

PEP 255, "*Simple Generators*"
Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.

5 PEP 237: Unifying Long Integers and Integers

In recent versions, the distinction between regular integers, which are 32-bit values on most machines, and long integers, which can be of arbitrary size, was becoming an annoyance. For example, on platforms that support files larger than `2**32` bytes, the `tell()` method of file objects has to return a long integer. However, there were various bits of Python that expected plain integers and would raise an error if a long integer was provided instead. For example, in Python 1.5, only regular integers could be used as a slice index, and `'abc'[1L:]` would raise a `TypeError` exception with the message 'slice index must be int'.

Python 2.2 will shift values from short to long integers as required. The 'L' suffix is no longer needed to indicate a long integer literal, as now the compiler will choose the appropriate type. (Using the 'L' suffix will be discouraged in future 2.x versions of Python, triggering a warning in Python 2.4, and probably dropped in Python 3.0.) Many operations that used to raise an `OverflowError` will now return a long integer as their result. For example:

```
>>> 1234567890123
1234567890123L
>>> 2 ** 64
18446744073709551616L
```

In most cases, integers and long integers will now be treated identically. You can still distinguish them with the `type()` built-in function, but that's rarely needed.

See Also:

PEP 237, "*Unifying Long Integers and Integers*"
Written by Moshe Zadka and Guido van Rossum. Implemented mostly by Guido van Rossum.

6 PEP 238: Changing the Division Operator

The most controversial change in Python 2.2 heralds the start of an effort to fix an old design flaw that's been in Python from the beginning. Currently Python's division operator, `/`, behaves like C's division operator when presented with two integer arguments: it returns an integer result that's truncated down when there would be a fractional part. For example, `3/2` is 1, not 1.5, and `(-1)/2` is -1, not -0.5. This means that the results of divison can vary unexpectedly depending on the type of the two operands and because Python is dynamically typed, it can be difficult to determine the possible types of the operands.

(The controversy is over whether this is *really* a design flaw, and whether it's worth breaking existing code to fix this. It's caused endless discussions on python-dev, and in July 2001 erupted into an storm of acidly sarcastic postings on comp.lang.python. I won't argue for either side here and will stick to describing what's implemented in 2.2. Read PEP 238 for a summary of arguments and counter-arguments.)

Because this change might break code, it's being introduced very gradually. Python 2.2 begins the transition, but the switch won't be complete until Python 3.0.

First, I'll borrow some terminology from PEP 238. "True division" is the division that most non-programmers are familiar with: 3/2 is 1.5, 1/4 is 0.25, and so forth. "Floor division" is what Python's `/` operator currently does when given integer operands; the result is the floor of the value returned by true division. "Classic division" is the current mixed behaviour of `/`; it returns the result of floor division when the operands are integers, and returns the result of true division when one of the operands is a floating-point number.

Here are the changes 2.2 introduces:

- A new operator, `//`, is the floor division operator. (Yes, we know it looks like C++'s comment symbol.) `//` *always* performs floor division no matter what the types of its operands are, so `1 // 2` is 0 and `1.0 // 2.0` is also 0.0.

  `//` is always available in Python 2.2; you don't need to enable it using a `__future__` statement.

- By including a `from __future__ import division` in a module, the `/` operator will be changed to return the result of true division, so `1/2` is 0.5. Without the `__future__` statement, `/` still means classic division. The default meaning of `/` will not change until Python 3.0.

- Classes can define methods called `__truediv__` and `__floordiv__` to overload the two division operators. At the C level, there are also slots in the `PyNumberMethods` structure so extension types can define the two operators.
- Python 2.2 supports some command-line arguments for testing whether code will works with the changed division semantics. Running python with **-Q warn** will cause a warning to be issued whenever division is applied to two integers. You can use this to find code that's affected by the change and fix it. By default, Python 2.2 will simply perform classic division without a warning; the warning will be turned on by default in Python 2.3.

See Also:

PEP 238, "*Changing the Division Operator*"
Written by Moshe Zadka and Guido van Rossum. Implemented by Guido van Rossum..

7 Unicode Changes

Python's Unicode support has been enhanced a bit in 2.2. Unicode strings are usually stored as UCS-2, as 16-bit unsigned integers. Python 2.2 can also be compiled to use UCS-4, 32-bit unsigned integers, as its internal encoding by supplying **--enable-unicode=ucs4** to the configure script. (It's also possible to specify **--disable-unicode** to completely disable Unicode support.)

When built to use UCS-4 (a "wide Python"), the interpreter can natively handle Unicode characters from U+000000 to U+110000, so the range of legal values for the `unichr()` function is expanded accordingly. Using an interpreter compiled to use UCS-2 (a "narrow Python"), values greater than 65535 will still cause `unichr()` to raise a `ValueError` exception. This is all described in PEP 261, "Support for 'wide' Unicode characters"; consult it for further details.

Another change is simpler to explain. Since their introduction, Unicode strings have supported an `encode()` method to convert the string to a selected encoding such as UTF-8 or Latin-1. A symmetric `decode(`[*encoding*]`)` method has been added to 8-bit strings (though not to Unicode strings) in 2.2. `decode()` assumes that the string is in the specified encoding and decodes it, returning whatever is returned by the codec.

Using this new feature, codecs have been added for tasks not directly related to Unicode. For example, codecs have been added for uu-encoding, MIME's base64 encoding, and compression with the `zlib` module:

```
>>> s = """Here is a lengthy piece of redundant, overly verbose,
... and repetitive text.
... """
>>> data = s.encode('zlib')
>>> data
'x\x9c\r\xc9\xc1\r\x80 \x10\x04\xc0?Ul...'
>>> data.decode('zlib')
'Here is a lengthy piece of redundant, overly verbose,\nand repetitive text.\n'
>>> print s.encode('uu')
begin 666 <data>
M2&5R92!I<R!A(&QE;F=T:'D@<&EE8V4@;V8@<F5D=6YD86YT+"!O=F5R;'D@
>=F5R8F]S92P*86YD('")E<&5T:71I=F4@=&5X="X*

end
>>> "sheesh".encode('rot-13')
'furrfu'
```

To convert a class instance to Unicode, a `__unicode__` method can be defined by a class, analogous to `__str__`.

`encode()`, `decode()`, and `__unicode__` were implemented by Marc-André Lemburg. The changes to support using UCS-4 internally were implemented by Fredrik Lundh and Martin von Löwis.

See Also:

PEP 261, "*Support for 'wide' Unicode characters*"
Written by Paul Prescod.

8 PEP 227: Nested Scopes

In Python 2.1, statically nested scopes were added as an optional feature, to be enabled by a `from __future__ import nested_scopes` directive. In 2.2 nested scopes no longer need to be specially enabled, and are now always present. The rest of this section is a copy of the description of nested scopes from my "What's New in Python 2.1" document; if you read it when 2.1 came out, you can skip the rest of this section.

The largest change introduced in Python 2.1, and made complete in 2.2, is to Python's scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn't match their intuitive expectations. For example, a nested recursive function definition doesn't work:

```
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function `g()` will always raise a `NameError` exception, because the binding of the name '`g`' isn't in either its local namespace or in the module-level namespace. This isn't much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the `lambda` statement clumsier, and this was a problem in practice. In code which uses `lambda` you can often find local variables being copied by passing them as the default values of arguments.

```
def find(self, name):
    "Return list of any entries equal to 'name'"
    L = filter(lambda x, name=name: x == name,
               self.list_attribute)
    return L
```

The readability of Python code written in a strongly functional style suffers greatly as a result.

The most significant change to Python 2.2 is that static scoping has been added to the language to fix this problem. As a first effect, the `name=name` default argument is now unnecessary in the above example. Put simply, when a given variable name is not assigned a value within a function (by an assignment, or the `def`, `class`, or `import` statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable name is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the `from` *module* `import *` and `exec` statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that `from` *module* `import *` is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. `from` *module* `import *` and `exec` make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or `lambda` expressions with free variables, the compiler will flag this by raising a `SyntaxError` exception.

To make the preceding explanation a bit clearer, here's an example:

```
x = 1
def f():
    # The next line is a syntax error
    exec 'x=2'
    def g():
        return x
```

Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named '`x`' whose value should be accessed by `g()`.

This shouldn't be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it's often a sign of a poor design anyway).

See Also:

PEP 227, "*Statically Nested Scopes*"
Written and implemented by Jeremy Hylton.

9 New and Improved Modules

- The `xmlrpclib` module was contributed to the standard library by Fredrik Lundh, provding support for writing XML-RPC clients. XML-RPC is a simple remote procedure call protocol built on top of HTTP and XML. For example, the following snippet retrieves a list of RSS channels from the O'Reilly Network, and then lists the recent headlines for one channel:

  ```
  import xmlrpclib
  s = xmlrpclib.Server(
        'http://www.oreillynet.com/meerkat/xml-rpc/server.php')
  channels = s.meerkat.getChannels()
  # channels is a list of dictionaries, like this:
  # [{'id': 4, 'title': 'Freshmeat Daily News'}
  #  {'id': 190, 'title': '32Bits Online'},
  #  {'id': 4549, 'title': '3DGamers'}, ... ]

  # Get the items for one channel
  items = s.meerkat.getItems( {'channel': 4} )

  # 'items' is another list of dictionaries, like this:
  # [{'link': 'http://freshmeat.net/releases/52719/',
  #   'description': 'A utility which converts HTML to XSL FO.',
  #   'title': 'html2fo 0.3 (Default)'}, ... ]
  ```

  The `SimpleXMLRPCServer` module makes it easy to create straightforward XML-RPC servers. See http://www.xmlrpc.com/ for more information about XML-RPC.

- The new `hmac` module implements the HMAC algorithm described by RFC 2104. (Contributed by Gerhard Häring.)

- Several functions that originally returned lengthy tuples now return pseudo-sequences that still behave like tuples but also have mnemonic attributes such as memberst_mtime or `tm_year`. The enhanced functions include `stat()`, `fstat()`, `statvfs()`, and `fstatvfs()` in the `os` module, and `localtime()`, `gmtime()`, and `strptime()` in the `time` module.

  For example, to obtain a file's size using the old tuples, you'd end up writing something like `file_size = os.stat(filename)[stat.ST_SIZE]`, but now this can be written more clearly as `file_size = os.stat(filename).st_size`.

The original patch for this feature was contributed by Nick Mathewson.

- The Python profiler has been extensively reworked and various errors in its output have been corrected. (Contributed by Fred Fred L. Drake, Jr. and Tim Peters.)
- The `socket` module can be compiled to support IPv6; specify the **--enable-ipv6** option to Python's configure script. (Contributed by Jun-ichiro "itojun" Hagino.)
- Two new format characters were added to the `struct` module for 64-bit integers on platforms that support the C `long long` type. '`q`' is for a signed 64-bit integer, and '`Q`' is for an unsigned one. The value is returned in Python's long integer type. (Contributed by Tim Peters.)
- In the interpreter's interactive mode, there's a new built-in function `help()` that uses the `pydoc` module introduced in Python 2.1 to provide interactive help. `help(`*object*`)` displays any available help text about *object*. `help()` with no argument puts you in an online help utility, where you can enter the names of functions, classes, or modules to read their help text. (Contributed by Guido van Rossum, using Ka-Ping Yee's `pydoc` module.)
- Various bugfixes and performance improvements have been made to the SRE engine underlying the `re` module. For example, the `re.sub()` and `re.split()` functions have been rewritten in C. Another contributed patch speeds up certain Unicode character ranges by a factor of two, and a new `finditer()` method that returns an iterator over all the non-overlapping matches in a given string. (SRE is maintained by Fredrik Lundh. The BIGCHARSET patch was contributed by Martin von Löwis.)
- The `smtplib` module now supports RFC 2487, "Secure SMTP over TLS", so it's now possible to encrypt the SMTP traffic between a Python program and the mail transport agent being handed a message. `smtplib` also supports SMTP authentication. (Contributed by Gerhard Häring.)
- The `imaplib` module, maintained by Piers Lauder, has support for several new extensions: the NAMESPACE extension defined in RFC 2342, SORT, GETACL and SETACL. (Contributed by Anthony Baxter and Michel Pelletier.)
- The `rfc822` module's parsing of email addresses is now compliant with RFC 2822, an update to RFC 822. (The module's name is *not* going to be changed to '`rfc2822`'.) A new package, `email`, has also been added for parsing and generating e-mail messages. (Contributed by Barry Warsaw, and arising out of his work on Mailman.)
- The `difflib` module now contains a new `Differ` class for producing human-readable lists of changes (a "delta") between two sequences of lines of text. There are also two generator functions, `ndiff()` and `restore()`, which respectively return a delta from two sequences, or one of the original sequences from a delta. (Grunt work contributed by David Goodger, from ndiff.py code by Tim Peters who then did the generatorization.)
- New constants `ascii_letters`, `ascii_lowercase`, and `ascii_uppercase` were added to the `string` module. There were several modules in the standard library that used `string.letters` to mean the ranges A-Za-z, but that assumption is incorrect when locales are in use, because `string.letters` varies depending on the set of legal characters defined by the current locale. The buggy modules have all been fixed to use `ascii_letters` instead. (Reported by an unknown person; fixed by Fred L. Drake, Jr.)
- The `mimetypes` module now makes it easier to use alternative MIME-type databases by the addition of a `MimeTypes` class, which takes a list of filenames to be parsed. (Contributed by Fred L. Drake, Jr.)
- A `Timer` class was added to the `threading` module that allows scheduling an activity to happen at some future time. (Contributed by Itamar Shtull-Trauring.)

10 Interpreter Changes and Fixes

Some of the changes only affect people who deal with the Python interpreter at the C level because they're writing Python extension modules, embedding the interpreter, or just hacking on the interpreter itself. If you only write Python code, none of the changes described here will affect you very much.

- Profiling and tracing functions can now be implemented in C, which can operate at much higher speeds than Python-based functions and should reduce the overhead of profiling and tracing. This will be of interest to authors of development environments for Python. Two new C functions were added to Python's API, `PyEval_SetProfile()` and `PyEval_SetTrace()`. The existing `sys.setprofile()` and `sys.settrace()` functions still exist, and have simply been changed to use the new C-level interface. (Contributed by Fred L. Drake, Jr.)
- Another low-level API, primarily of interest to implementors of Python debuggers and development tools, was added. `PyInterpreterState_Head()` and `PyInterpreterState_Next()` let a caller walk through all the existing interpreter objects; `PyInterpreterState_ThreadHead()` and `PyThreadState_Next()` allow looping over all the thread states for a given interpreter. (Contributed by David Beazley.)
- A new '`et`' format sequence was added to `PyArg_ParseTuple`; '`et`' takes both a parameter and an encoding name, and converts the parameter to the given encoding if the parameter turns out to be a Unicode string, or leaves it alone if it's an 8-bit string, assuming it to already be in the desired encoding. This differs from the '`es`' format character, which assumes that 8-bit strings are in Python's default ASCII encoding and converts them to the specified new encoding. (Contributed by M.-A. Lemburg, and used for the MBCS support on Windows described in the following section.)
- A different argument parsing function, `PyArg_UnpackTuple()`, has been added that's simpler and presumably faster. Instead of specifying a format string, the caller simply gives the minimum and maximum number of arguments expected, and a set of pointers to `PyObject*` variables that will be filled in with argument values.
- Two new flags `METH_NOARGS` and `METH_O` are available in method definition tables to simplify implementation of methods with no arguments or a single untyped argument. Calling such methods is more efficient than calling a corresponding method that uses `METH_VARARGS`. Also, the old `METH_OLDARGS` style of writing C methods is now officially deprecated.
- Two new wrapper functions, `PyOS_snprintf()` and `PyOS_vsnprintf()` were added to provide cross-platform implementations for the relatively new `snprintf()` and `vsnprintf()` C lib APIs. In contrast to the standard `sprintf()` and `vsprintf()` functions, the Python versions check the bounds of the buffer used to protect against buffer overruns. (Contributed by M.-A. Lemburg.)
- The `_PyTuple_Resize()` function has lost an unused parameter, so now it takes 2 parameters instead of 3. The third argument was never used, and can simply be discarded when porting code from earlier versions to Python 2.2.

11 Other Changes and Fixes

As usual there were a bunch of other improvements and bugfixes scattered throughout the source tree. A search through the CVS change logs finds there were 527 patches applied, and 683 bugs fixed; both figures are likely to be underestimates. Some of the more notable changes are:

- The code for the MacOS port for Python, maintained by Jack Jansen, is now kept in the main Python CVS tree, and many changes have been made to support MacOS X.

  The most significant change is the ability to build Python as a framework, enabled by supplying the **--enable-framework** option to the configure script when compiling Python. According to

Jack Jansen, “This installs a self-contained Python installation plus the OS X framework ”glue” into ‘/Library/Frameworks/Python.framework’ (or another location of choice). For now there is little immediate added benefit to this (actually, there is the disadvantage that you have to change your PATH to be able to find Python), but it is the basis for creating a full-blown Python application, porting the MacPython IDE, possibly using Python as a standard OSA scripting language and much more.”

Most of the MacPython toolbox modules, which interface to MacOS APIs such as windowing, QuickTime, scripting, etc. have been ported to OS X, but they’ve been left commented out in ‘setup.py’. People who want to experiment with these modules can uncomment them manually.

- Keyword arguments passed to builtin functions that don’t take them now cause a `TypeError` exception to be raised, with the message ”*function* takes no keyword arguments”.
- Weak references, added in Python 2.1 as an extension module, are now part of the core because they’re used in the implementation of new-style classes. The `ReferenceError` exception has therefore moved from the `weakref` module to become a built-in exception.
- A new script, ‘Tools/scripts/cleanfuture.py’ by Tim Peters, automatically removes obsolete `__future__` statements from Python source code.
- An additional *flags* argument has been added to the built-in function `compile()`, so the behaviour of `__future__` statements can now be correctly observed in simulated shells, such as those presented by IDLE and other development environments. This is described in PEP 264. (Contributed by Michael Hudson.)
- The new license introduced with Python 1.6 wasn’t GPL-compatible. This is fixed by some minor textual changes to the 2.2 license, so it’s now legal to embed Python inside a GPLed program again. Note that Python itself is not GPLed, but instead is under a license that’s essentially equivalent to the BSD license, same as it always was. The license changes were also applied to the Python 2.0.1 and 2.1.1 releases.
- When presented with a Unicode filename on Windows, Python will now convert it to an MBCS encoded string, as used by the Microsoft file APIs. As MBCS is explicitly used by the file APIs, Python’s choice of ASCII as the default encoding turns out to be an annoyance. On UNIX, the locale’s character set is used if `locale.nl_langinfo(CODESET)` is available. (Windows support was contributed by Mark Hammond with assistance from Marc-André Lemburg. UNIX support was added by Martin von Löwis.)
- Large file support is now enabled on Windows. (Contributed by Tim Peters.)
- The ‘Tools/scripts/ftpmirror.py’ script now parses a ‘.netrc’ file, if you have one. (Contributed by Mike Romberg.)
- Some features of the object returned by the `xrange()` function are now deprecated, and trigger warnings when they’re accessed; they’ll disappear in Python 2.3. `xrange` objects tried to pretend they were full sequence types by supporting slicing, sequence multiplication, and the `in` operator, but these features were rarely used and therefore buggy. The `tolist()` method and the `start`, `stop`, and `step` attributes are also being deprecated. At the C level, the fourth argument to the `PyRange_New()` function, ‘`repeat`’, has also been deprecated.
- There were a bunch of patches to the dictionary implementation, mostly to fix potential core dumps if a dictionary contains objects that sneakily changed their hash value, or mutated the dictionary they were contained in. For a while python-dev fell into a gentle rhythm of Michael Hudson finding a case that dumped core, Tim Peters fixing the bug, Michael finding another case, and round and round it went.
- On Windows, Python can now be compiled with Borland C thanks to a number of patches contributed by Stephen Hansen, though the result isn’t fully functional yet. (But this *is* progress...)
- Another Windows enhancement: Wise Solutions generously offered PythonLabs use of their InstallerMaster 8.1 system. Earlier PythonLabs Windows installers used Wise 5.0a, which was beginning to show its age. (Packaged up by Tim Peters.)

- Files ending in '.pyw' can now be imported on Windows. '.pyw' is a Windows-only thing, used to indicate that a script needs to be run using PYTHONW.EXE instead of PYTHON.EXE in order to prevent a DOS console from popping up to display the output. This patch makes it possible to import such scripts, in case they're also usable as modules. (Implemented by David Bolen.)

- On platforms where Python uses the C `dlopen()` function to load extension modules, it's now possible to set the flags used by `dlopen()` using the `sys.getdlopenflags()` and `sys.setdlopenflags()` functions. (Contributed by Bram Stolk.)

- The `pow()` built-in function no longer supports 3 arguments when floating-point numbers are supplied. `pow(`*x*`, `*y*`, `*z*`)` returns `(x**y) %z`, but this is never useful for floating point numbers, and the final result varies unpredictably depending on the platform. A call such as `pow(2.0, 8.0, 7.0)` will now raise a `TypeError` exception.

12 Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Fred Bremmer, Keith Briggs, Andrew Dalke, Fred L. Drake, Jr., Carel Fellinger, David Goodger, Mark Hammond, Stephen Hansen, Michael Hudson, Jack Jansen, Marc-André Lemburg, Martin von Löwis, Fredrik Lundh, Michael McLay, Nick Mathewson, Paul Moore, Gustavo Niemeyer, Don O'Donnell, Tim Peters, Jens Quade, Tom Reinhardt, Neil Schemenauer, Guido van Rossum, Greg Ward.