What's New in Python 2.3

- PEP 218: A Standard Set Datatype
- PEP 255: Simple Generators
- PEP 263: Source Code Encodings
- PEP 273: Importing Modules from ZIP Archives
- PEP 277: Unicode file name support for Windows NT
- PEP 278: Universal Newline Support
- PEP 279: enumerate()
- PEP 282: The logging Package
- PEP 285: A Boolean Type
- PEP 293: Codec Error Handling Callbacks
- PEP 301: Package Index and Metadata for Distutils
- PEP 302: New Import Hooks
- PEP 305: Comma-separated Files
- PEP 307: Pickle Enhancements
- Extended Slices
- Other Language Changes
  - String Changes
  - Optimizations
- New, Improved, and Deprecated Modules
  - Date/Time Type
  - The optparse Module
- Pymalloc: A Specialized Object Allocator
- Build and C API Changes
  - Port-Specific Changes
- Other Changes and Fixes
- Porting to Python 2.3
- Acknowledgements

Author
A.M. Kuchling

This article explains the new features in Python 2.3. Python 2.3 was released on July 29, 2003.

The main themes for Python 2.3 are polishing some of the features added in 2.2, adding various small but useful enhancements to the core language, and expanding the standard library. The new object model introduced in the previous version has benefited from 18 months of bugfixes and from optimization efforts that have improved the performance of new-style classes. A few new built-in functions have been added such as `sum` and `enumerate`. The `in` operator can now be used for substring searches (e.g. `"ab" in "abc"` returns `True`).

Some of the many new library features include Boolean, set, heap, and date/time data types, the ability to import modules from ZIP-format archives, metadata support for the long-awaited Python catalog, an updated version of IDLE, and modules for logging messages, wrapping text, parsing CSV files, processing command-line options, using BerkeleyDB databases... the list of new and enhanced modules is lengthy.

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 2.3, such as the Python Library Reference and the Python Reference Manual. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

PEP 218: A Standard Set Datatype

The new `!sets` module contains an implementation of a set datatype. The `Set` class is for mutable sets, sets that can have members added and removed. The `!ImmutableSet` class is for sets that can't be modified, and instances of `!ImmutableSet` can therefore be used as dictionary keys. Sets are built on top of dictionaries, so the elements within a set must be hashable.

Here's a simple example:

```
>>> import sets
>>> S = sets.Set([1,2,3])
>>> S
Set([1, 2, 3])
>>> 1 in S
True
>>> 0 in S
False
>>> S.add(5)
```

```
>>> S.remove(3)
>>> S
Set([1, 2, 5])
>>>
```

The union and intersection of sets can be computed with the `~frozenset.union` and `~frozenset.intersection` methods; an alternative notation uses the bitwise operators `&` and `|`. Mutable sets also have in-place versions of these methods, `!union_update` and `~set.intersection_update`. :

```
>>> S1 = sets.Set([1,2,3])
>>> S2 = sets.Set([4,5,6])
>>> S1.union(S2)
Set([1, 2, 3, 4, 5, 6])
>>> S1 | S2                 # Alternative notation
Set([1, 2, 3, 4, 5, 6])
>>> S1.intersection(S2)
Set([])
>>> S1 & S2                 # Alternative notation
Set([])
>>> S1.union_update(S2)
>>> S1
Set([1, 2, 3, 4, 5, 6])
>>>
```

It's also possible to take the symmetric difference of two sets. This is the set of all elements in the union that aren't in the intersection. Another way of putting it is that the symmetric difference contains all elements that are in exactly one set. Again, there's an alternative notation (`^`), and an in-place version with the ungainly name `~set.symmetric_difference_update`. :

```
>>> S1 = sets.Set([1,2,3,4])
>>> S2 = sets.Set([3,4,5,6])
>>> S1.symmetric_difference(S2)
Set([1, 2, 5, 6])
>>> S1 ^ S2
Set([1, 2, 5, 6])
>>>
```

There are also `!issubset` and `!issuperset` methods for checking whether one set is a subset or superset of another:

```
>>> S1 = sets.Set([1,2,3])
>>> S2 = sets.Set([2,3])
>>> S2.issubset(S1)
True
>>> S1.issubset(S2)
False
>>> S1.issuperset(S2)
True
>>>
```

`218` - Adding a Built-In Set Object Type
PEP written by Greg V. Wilson. Implemented by Greg V. Wilson, Alex Martelli, and GvR.

PEP 255: Simple Generators

In Python 2.2, generators were added as an optional feature, to be enabled by a `from __future__ import generators` directive. In 2.3 generators no longer need to be specially enabled, and are now always present; this means that `yield` is now always a keyword. The rest of this section is a copy of the description of generators from the "What's New in Python 2.2" document; if you read it back when Python 2.2 came out, you can skip the rest of this section.

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.

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 `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 (https://www2.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 https://www2.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 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.

`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.

PEP 263: Source Code Encodings

Python source files can now be declared as being in different character set encodings. Encodings are declared by including a specially formatted comment in the first or second line of the source file. For example, a UTF-8 file can be declared with:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

Without such an encoding declaration, the default encoding used is 7-bit ASCII. Executing or importing modules that contain string literals with 8-bit characters and have no encoding declaration will result in a `DeprecationWarning` being signalled by Python 2.3; in 2.4 this will be a syntax error.

The encoding declaration only affects Unicode string literals, which will be converted to Unicode using the specified encoding. Note that Python identifiers are still restricted to ASCII characters, so you can't have variable names that use characters outside of the usual alphanumerics.

`263` - Defining Python Source Code Encodings
Written by Marc-André Lemburg and Martin von Löwis; implemented by Suzuki Hisao and Martin von Löwis.

PEP 273: Importing Modules from ZIP Archives

The new `zipimport` module adds support for importing modules from a ZIP-format archive. You don't need to import the module explicitly; it will be automatically imported if a ZIP archive's filename is added to `sys.path`. For example:

```
amk@nyman:~/src/python$ unzip -l /tmp/example.zip
Archive:  /tmp/example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
amk@nyman:~/src/python$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, '/tmp/example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'/tmp/example.zip/jwzthreading.py'
>>>
```

An entry in `sys.path` can now be the filename of a ZIP archive. The ZIP archive can contain any kind of files, but only files named `\*.py`, `\*.pyc`, or `\*.pyo` can be imported. If an archive only contains `\*.py` files, Python will not attempt to modify the archive by adding the corresponding `\*.pyc`

file, meaning that if a ZIP archive doesn't contain `\*.pyc` files, importing may be rather slow.

A path within the archive can also be specified to only import from a subdirectory; for example, the path `/tmp/example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

`273` - Import Modules from Zip Archives
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in `273`, but uses an implementation written by Just van Rossum that uses the import hooks described in `302`. See section `section-pep302` for a description of the new import hooks.

PEP 277: Unicode file name support for Windows NT

On Windows NT, 2000, and XP, the system stores file names as Unicode strings. Traditionally, Python has represented file names as byte strings, which is inadequate because it renders some file names inaccessible.

Python now allows using arbitrary Unicode strings (within the limitations of the file system) for all functions that expect file names, most notably the `open` built-in function. If a Unicode string is passed to `os.listdir`, Python now returns a list of Unicode strings. A new function, `!os.getcwdu`, returns the current directory as a Unicode string.

Byte strings still work as file names, and on Windows Python will transparently convert them to Unicode using the `mbcs` encoding.

Other systems also allow Unicode strings as file names but convert them to byte strings before passing them to the system, which can cause a `UnicodeError` to be raised. Applications can test whether arbitrary Unicode strings are supported as file names by checking `os.path.supports_unicode_filenames`, a Boolean value.

Under MacOS, `os.listdir` may now return Unicode filenames.

`277` - Unicode file name support for Windows NT
Written by Neil Hodgson; implemented by Neil Hodgson, Martin von Löwis, and Mark Hammond.

single: universal newlines; What's new

PEP 278: Universal Newline Support

The three major operating systems used today are Microsoft Windows, Apple's Macintosh OS, and the various Unix derivatives. A minor irritation of cross-platform work is that these three platforms all use different characters to mark the ends of lines in text files. Unix uses the linefeed (ASCII character 10), MacOS uses the carriage return (ASCII character 13), and Windows uses a two-character sequence of a carriage return plus a newline.

Python's file objects can now support end of line conventions other than the one followed by the platform on which Python is running. Opening a file with the mode `'U'` or `'rU'` will open a file for reading in `universal newlines` mode. All three line ending conventions will be translated to a `'\n'` in the strings returned by the various file methods such as `!read` and `!readline`.

Universal newline support is also used when importing modules and when executing a file with the `!execfile` function. This means that Python modules can be shared between all three operating systems without needing to convert the line-endings.

This feature can be disabled when compiling Python by specifying the `!--without-universal-newlines` switch when running Python's `configure` script.

`278` - Universal Newline Support
Written and implemented by Jack Jansen.

PEP 279: enumerate()

A new built-in function, `enumerate`, will make certain loops a bit clearer. `enumerate(thing)`, where *thing* is either an iterator or a sequence, returns an iterator that will return `(0, thing[0])`, `(1, thing[1])`, `(2, thing[2])`, and so forth.

A common idiom to change every element of a list looks like this:

```
for i in range(len(L)):
    item = L[i]
    # ... compute some result based on item ...
    L[i] = result
```

This can be rewritten using `enumerate` as:

```
for i, item in enumerate(L):
    # ... compute some result based on item ...
    L[i] = result
```

`279` - The enumerate() built-in function
Written and implemented by Raymond D. Hettinger.

PEP 282: The logging Package

A standard package for writing logs, `logging`, has been added to Python 2.3. It provides a powerful and flexible mechanism for generating logging output which can then be filtered and processed in various ways. A configuration file written in a standard format can be used to control the logging behavior of a program. Python includes handlers that will write log records to standard error or to a file or socket, send them to the system log, or even e-mail them to a particular address; of course, it's also possible to write your own handler classes.

The `~logging.Logger` class is the primary class. Most application code will deal with one or more `~logging.Logger` objects, each one used by a particular subsystem of the application. Each `~logging.Logger` is identified by a name, and names are organized into a hierarchy using `.` as the component separator. For example, you might have `~logging.Logger` instances named `server`, `server.auth` and `server.network`. The latter two instances are below `server` in the hierarchy. This means that if you turn up the verbosity for `server` or direct `server` messages to a different handler, the changes will also apply to records logged to `server.auth` and `server.network`. There's also a root `~logging.Logger` that's the parent of all other loggers.

For simple uses, the `logging` package contains some convenience functions that always use the root log:

```
import logging


logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

In the default configuration, informational and debugging messages are suppressed and the output is sent to standard error. You can enable the display of informational and debugging messages by calling the `~logging.Logger.setLevel` method on the root logger.

Notice the `~logging.warning` call's use of string formatting operators; all of the functions for logging messages take the arguments `(msg, arg1, arg2, ...)` and log the string resulting from `msg % (arg1, arg2, ...)`.

There's also an `~logging.exception` function that records the most recent traceback. Any of the other functions will also record the traceback if you specify a true value for the keyword argument *exc_info*. :

```
def f():
    try:    1/0
    except: logging.exception('Problem recorded')


f()
```

This produces the following output:

```
ERROR:root:Problem recorded
Traceback (most recent call last):
  File "t.py", line 6, in f
    1/0
ZeroDivisionError: integer division or modulo by zero
```

Slightly more advanced programs will use a logger other than the root logger. The `getLogger(name)` function is used to get a particular log, creating it if it doesn't exist yet. `getLogger(None)` returns the root logger. :

```
log = logging.getLogger('server')
 ...
log.info('Listening on port %i', port)
 ...
log.critical('Disk full')
 ...
```

Log records are usually propagated up the hierarchy, so a message logged to `server.auth` is also seen by `server` and `root`, but a `~logging.Logger` can prevent this by setting its `~logging.Logger.propagate` attribute to `False`.

There are more classes provided by the `logging` package that can be customized. When a `~logging.Logger` instance is told to log a message, it creates a `~logging.LogRecord` instance that is

sent to any number of different `~logging.Handler` instances. Loggers and handlers can also have an attached list of filters, and each filter can cause the `~logging.LogRecord` to be ignored or can modify the record before passing it along. When they're finally output, `~logging.LogRecord` instances are converted to text by a `~logging.Formatter` class. All of these classes can be replaced by your own specially written classes.

With all of these features the `logging` package should provide enough flexibility for even the most complicated applications. This is only an incomplete overview of its features, so please see the package's reference documentation for all of the details. Reading `282` will also be helpful.

`282` - A Logging System
Written by Vinay Sajip and Trent Mick; implemented by Vinay Sajip.

PEP 285: A Boolean Type

A Boolean type was added to Python 2.3. Two new constants were added to the `!__builtin__` module, `True` and `False`. (`True` and `False` constants were added to the built-ins in Python 2.2.1, but the 2.2.1 versions are simply set to integer values of 1 and 0 and aren't a different type.)

The type object for this new type is named `bool`; the constructor for it takes any Python value and converts it to `True` or `False`. :

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool([])
False
>>> bool( (1,) )
True
```

Most of the standard library modules and built-in functions have been changed to return Booleans. :

```
>>> obj = []
>>> hasattr(obj, 'append')
True
>>> isinstance(obj, list)
True
>>> isinstance(obj, tuple)
False
```

Python's Booleans were added with the primary goal of making code clearer. For example, if you're reading a function and encounter the statement `return 1`, you might wonder whether the `1` represents a Boolean truth value, an index, or a coefficient that multiplies some other quantity. If the statement is `return True`, however, the meaning of the return value is quite clear.

Python's Booleans were *not* added for the sake of strict type-checking. A very strict language such as Pascal would also prevent you performing arithmetic with Booleans, and would require that the expression in an `if` statement always evaluate to a Boolean result. Python is not this strict and never will be, as `285` explicitly says. This means you can still use any expression in an `!if` statement, even ones that evaluate to a list or tuple or some random object. The Boolean type is a subclass of the `int` class so that arithmetic using a Boolean still works. :

```
>>> True + 1
2
>>> False + 1
1
>>> False * 75
0
>>> True * 75
75
```

To sum up `True` and `False` in a sentence: they're alternative ways to spell the integer values 1 and 0, with the single difference that `str` and `repr` return the strings `'True'` and `'False'` instead of `'1'` and `'0'`.

`285` - Adding a bool type
Written and implemented by GvR.

PEP 293: Codec Error Handling Callbacks

When encoding a Unicode string into a byte string, unencodable characters may be encountered. So far, Python has allowed specifying the error processing as either "strict" (raising `UnicodeError`), "ignore" (skipping the character), or "replace" (using a question mark in the output string), with "strict" being the default behavior. It may be desirable to specify alternative processing of such errors, such as inserting an XML character reference or HTML entity reference into the converted string.

Python now has a flexible framework to add different processing strategies. New error handlers can be added with `codecs.register_error`, and codecs then can access the error handler with `codecs.lookup_error`

. An equivalent C API has been added for codecs written in C. The error handler gets the necessary state information such as the string being converted, the position in the string where the error was detected, and the target encoding. The handler can then either raise an exception or return a replacement string.

Two additional error handlers have been implemented using this framework: "backslashreplace" uses Python backslash quoting to represent unencodable characters and "xmlcharrefreplace" emits XML character references.

`293` - Codec Error Handling Callbacks
Written and implemented by Walter Dörwald.

PEP 301: Package Index and Metadata for Distutils

Support for the long-requested Python catalog makes its first appearance in 2.3.

The heart of the catalog is the new Distutils `register` command. Running `python setup.py register` will collect the metadata describing a package, such as its name, version, maintainer, description, &c., and send it to a central catalog server. The resulting catalog is available from https://pypi.org.

To make the catalog a bit more useful, a new optional *classifiers* keyword argument has been added to the Distutils `!setup` function. A list of Trove-style strings can be supplied to help classify the software.

Here's an example `setup.py` with classifiers, written to be compatible with older versions of the Distutils:

```
from distutils import core
kw = {'name': "Quixote",
      'version': "0.5.1",
      'description': "A highly Pythonic Web application framework",
      # ...
      }

if (hasattr(core, 'setup_keywords') and
    'classifiers' in core.setup_keywords):
    kw['classifiers'] = \
        ['Topic :: Internet :: WWW/HTTP :: Dynamic Content',
```

```
          'Environment :: No Input/Output (Daemon)',
          'Intended Audience :: Developers'],

core.setup(**kw)
```

The full list of classifiers can be obtained by running `python setup.py register --list-classifiers`.

`301` - Package Index and Metadata for Distutils
Written and implemented by Richard Jones.

PEP 302: New Import Hooks

While it's been possible to write custom import hooks ever since the `!ihooks` module was introduced in Python 1.3, no one has ever been really happy with it because writing new import hooks is difficult and messy. There have been various proposed alternatives such as the `!imputil` and `!iu` modules, but none of them has ever gained much acceptance, and none of them were easily usable from C code.

`302` borrows ideas from its predecessors, especially from Gordon McMillan's `!iu` module. Three new items are added to the `sys` module:

- `sys.path_hooks` is a list of callable objects; most often they'll be classes. Each callable takes a string containing a path and either returns an importer object that will handle imports from this path or raises an `ImportError` exception if it can't handle this path.
- `sys.path_importer_cache` caches importer objects for each path, so `sys.path_hooks` will only need to be traversed once for each path.
- `sys.meta_path` is a list of importer objects that will be traversed before `sys.path` is checked. This list is initially empty, but user code can add objects to it. Additional built-in and frozen modules can be imported by an object added to this list.

Importer objects must have a single method, `find_module(fullname, path=None)`. *fullname* will be a module or package name, e.g. `string` or `distutils.core`. `!find_module` must return a loader object that has a single method, `load_module(fullname)`, that creates and returns the corresponding module object.

Pseudo-code for Python's new import logic, therefore, looks something like this (simplified a bit; see `302` for the full details):

```
for mp in sys.meta_path:
    loader = mp(fullname)
    if loader is not None:
        <module> = loader.load_module(fullname)
```

```
for path in sys.path:
    for hook in sys.path_hooks:
        try:
            importer = hook(path)
        except ImportError:
            # ImportError, so try the other path hooks
            pass
        else:
            loader = importer.find_module(fullname)
            <module> = loader.load_module(fullname)

# Not found!
raise ImportError
```

`302` - New Import Hooks
Written by Just van Rossum and Paul Moore. Implemented by Just van Rossum.

PEP 305: Comma-separated Files

Comma-separated files are a format frequently used for exporting data from databases and spreadsheets. Python 2.3 adds a parser for comma-separated files.

Comma-separated format is deceptively simple at first glance:

```
Costs,150,200,3.95
```

Read a line and call `line.split(',')`: what could be simpler? But toss in string data that can contain commas, and things get more complicated:

```
"Costs",150,200,3.95,"Includes taxes, shipping, and sundry items"
```

A big ugly regular expression can parse this, but using the new `csv` package is much simpler:

```
import csv

input = open('datafile', 'rb')
reader = csv.reader(input)
for line in reader:
    print line
```

The `~csv.reader` function takes a number of different options. The field separator isn't limited to the comma and can be changed to any character, and so can the quoting and line-ending characters.

Different dialects of comma-separated files can be defined and registered; currently there are two dialects, both used by Microsoft Excel. A separate `csv.writer` class will generate comma-separated files from a succession of tuples or lists, quoting strings that contain the delimiter.

`305` - CSV File API
Written and implemented by Kevin Altis, Dave Cole, Andrew McNamara, Skip Montanaro, Cliff Wells.

PEP 307: Pickle Enhancements

The `pickle` and `!cPickle` modules received some attention during the 2.3 development cycle. In 2.2, new-style classes could be pickled without difficulty, but they weren't pickled very compactly; `307` quotes a trivial example where a new-style class results in a pickled string three times longer than that for a classic class.

The solution was to invent a new pickle protocol. The `pickle.dumps` function has supported a text-or-binary flag for a long time. In 2.3, this flag is redefined from a Boolean to an integer: 0 is the old text-mode pickle format, 1 is the old binary format, and now 2 is a new 2.3-specific format. A new constant, `pickle.HIGHEST_PROTOCOL`, can be used to select the fanciest protocol available.

Unpickling is no longer considered a safe operation. 2.2's `pickle` provided hooks for trying to prevent unsafe classes from being unpickled (specifically, a `!__safe_for_unpickling__` attribute), but none of this code was ever audited and therefore it's all been ripped out in 2.3. You should not unpickle untrusted data in any version of Python.

To reduce the pickling overhead for new-style classes, a new interface for customizing pickling was added using three special methods: `~object.__getstate__`, `~object.__setstate__`, and `~object.__getnewargs__`. Consult `307` for the full semantics of these methods.

As a way to compress pickles yet further, it's now possible to use integer codes instead of long strings to identify pickled classes. The Python Software Foundation will maintain a list of standardized codes; there's also a range of codes for private use. Currently no codes have been specified.

`307` - Extensions to the pickle protocol
Written and implemented by Guido van Rossum and Tim Peters.

Extended Slices

Ever since Python 1.4, the slicing syntax has supported an optional third "step" or "stride" argument. For example, these are all legal Python syntax: `L[1:10:2]`, `L[:-1:1]`, `L[::-1]`. This was added to Python at the request of the developers of Numerical Python, which uses the third argument extensively. However, Python's built-in list, tuple, and string sequence types have never supported this feature, raising a `TypeError` if you tried it. Michael Hudson contributed a patch to fix this shortcoming.

For example, you can now easily extract the elements of a list that have even indexes:

```
>>> L = range(10)
>>> L[::2]
[0, 2, 4, 6, 8]
```

Negative values also work to make a copy of the same list in reverse order:

```
>>> L[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

This also works for tuples, arrays, and strings:

```
>>> s='abcd'
>>> s[::2]
'ac'
>>> s[::-1]
'dcba'
```

If you have a mutable sequence such as a list or an array you can assign to or delete an extended slice, but there are some differences between assignment to extended and regular slices. Assignment to a regular slice can be used to change the length of the sequence:

```
>>> a = range(3)
>>> a
[0, 1, 2]
>>> a[1:3] = [4, 5, 6]
>>> a
[0, 4, 5, 6]
```

Extended slices aren't this flexible. When assigning to an extended slice, the list on the right hand side of the statement must contain the same number of items as the slice it is replacing:

```
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> a[::2] = [0, -1]
>>> a
[0, 1, -1, 3]
>>> a[::2] = [0,1,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
```

Deletion is more straightforward:

```
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> del a[::2]
>>> a
[1, 3]
```

One can also now pass slice objects to the `~object.__getitem__` methods of the built-in sequences:

```
>>> range(10).__getitem__(slice(0, 5, 2))
[0, 2, 4]
```

Or use slice objects directly in subscripts:

```
>>> range(10)[slice(0, 5, 2)]
[0, 2, 4]
```

To simplify implementing sequences that support extended slicing, slice objects now have a method `indices(length)` which, given the length of a sequence, returns a `(start, stop, step)` tuple that can be passed directly to `range`. `!indices` handles omitted and out-of-bounds indices in a manner consistent with regular slices (and this innocuous phrase hides a welter of confusing details!). The method is intended to be used like this:

```
class FakeSeq:
    ...
    def calc_item(self, i):
        ...
    def __getitem__(self, item):
        if isinstance(item, slice):
            indices = item.indices(len(self))
            return FakeSeq([self.calc_item(i) for i in range(*indices)])
        else:
            return self.calc_item(i)
```

From this example you can also see that the built-in `slice` object is now the type object for the slice type, and is no longer a function. This is consistent with Python 2.2, where `int`, `str`, etc., underwent the same change.

Other Language Changes

Here are all of the changes that Python 2.3 makes to the core Python language.

- The `yield` statement is now always a keyword, as described in section `section-generators` of this document.
- A new built-in function `enumerate` was added, as described in section `section-enumerate` of this document.
- Two new constants, `True` and `False` were added along with the built-in `bool` type, as described in section `section-bool` of this document.
- The `int` type constructor will now return a long integer instead of raising an `OverflowError` when a string or floating-point number is too large to fit into an integer. This can lead to the paradoxical result that `isinstance(int(expression), int)` is false, but that seems unlikely to cause problems in practice.
- Built-in types now support the extended slicing syntax, as described in section `section-slices` of this document.
- A new built-in function, `sum(iterable, start=0)`, adds up the numeric items in the iterable object and returns their sum. `sum` only accepts numbers, meaning that you can't use it to concatenate a bunch of strings. (Contributed by Alex Martelli.)
- `list.insert(pos, value)` used to insert *value* at the front of the list when *pos* was negative. The behaviour has now been changed to be consistent with slice indexing, so when *pos* is -1 the value will be inserted before the last element, and so forth.
- `list.index(value)`, which searches for *value* within the list and returns its index, now takes optional *start* and *stop* arguments to limit the search to only part of the list.
- Dictionaries have a new method, `pop(key[, *default*])`, that returns the value corresponding to *key* and removes that key/value pair from the dictionary. If the

requested key isn't present in the dictionary, *default* is returned if it's specified and `KeyError` raised if it isn't. :

```
>>> d = {1:2}
>>> d
{1: 2}
>>> d.pop(4)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 4
>>> d.pop(1)
2
>>> d.pop(1)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 'pop(): dictionary is empty'
>>> d
{}
>>>
```

There's also a new class method, `dict.fromkeys(iterable, value)`, that creates a dictionary with keys taken from the supplied iterator *iterable* and all values set to *value*, defaulting to `None`.
(Patches contributed by Raymond Hettinger.)
Also, the `dict` constructor now accepts keyword arguments to simplify creating small dictionaries:

```
>>> dict(red=1, blue=2, green=3, black=4)
{'blue': 2, 'black': 4, 'green': 3, 'red': 1}
```

(Contributed by Just van Rossum.)

- The `assert` statement no longer checks the `__debug__` flag, so you can no longer disable assertions by assigning to `__debug__`. Running Python with the `-O` switch will still generate code that doesn't execute any assertions.
- Most type objects are now callable, so you can use them to create new objects such as functions, classes, and modules. (This means that the `!new` module can be deprecated in a future Python version, because you can now use the type objects available in the `types` module.) For example, you can create a new module object with the following code:

```
>>> import types
>>> m = types.ModuleType('abc','docstring')
>>> m
<module 'abc' (built-in)>
```

```
>>> m.__doc__
'docstring'
```

- A new warning, `PendingDeprecationWarning` was added to indicate features which are in the process of being deprecated. The warning will *not* be printed by default. To check for use of features that will be deprecated in the future, supply `-Walways::PendingDeprecationWarning:: <-W>` on the command line or use `warnings.filterwarnings`.
- The process of deprecating string-based exceptions, as in `raise "Error occurred"`, has begun. Raising a string will now trigger `PendingDeprecationWarning`.
- Using `None` as a variable name will now result in a `SyntaxWarning` warning. In a future version of Python, `None` may finally become a keyword.
- The `!xreadlines` method of file objects, introduced in Python 2.1, is no longer necessary because files now behave as their own iterator. `!xreadlines` was originally introduced as a faster way to loop over all the lines in a file, but now you can simply write `for line in file_obj`. File objects also have a new read-only `!encoding` attribute that gives the encoding used by the file; Unicode strings written to the file will be automatically converted to bytes using the given encoding.
- The method resolution order used by new-style classes has changed, though you'll only notice the difference if you have a really complicated inheritance hierarchy. Classic classes are unaffected by this change. Python 2.2 originally used a topological sort of a class's ancestors, but 2.3 now uses the C3 algorithm as described in the paper "A Monotonic Superclass Linearization for Dylan". To understand the motivation for this change, read Michele Simionato's article `python_2.3_mro`, or read the thread on python-dev starting with the message at https://mail.python.org/pipermail/python-dev/2002-October/029035.html. Samuele Pedroni first pointed out the problem and also implemented the fix by coding the C3 algorithm.
- Python runs multithreaded programs by switching between threads after executing N bytecodes. The default value for N has been increased from 10 to 100 bytecodes, speeding up single-threaded applications by reducing the switching overhead. Some multithreaded applications may suffer slower response time, but that's easily fixed by setting the limit back to a lower number using `sys.setcheckinterval(N)`. The limit can be retrieved with the new `!sys.getcheckinterval` function.
- One minor but far-reaching change is that the names of extension types defined by the modules included with Python now contain the module and a `'.'` in front of the type name. For example, in Python 2.2, if you created a socket and printed its `!__class__`, you'd get this output:

```
>>> s = socket.socket()
>>> s.__class__
<type 'socket'>
```

In 2.3, you get this:

```
>>> s.__class__
<type '_socket.socket'>
```

- One of the noted incompatibilities between old- and new-style classes has been removed: you can now assign to the `~type.__name__` and `~type.__bases__` attributes of new-style classes. There are some restrictions on what can be assigned to `!__bases__` along the lines of those relating to assigning to an instance's `~object.__class__` attribute.

String Changes

- The `in` operator now works differently for strings. Previously, when evaluating `X in Y` where *X* and *Y* are strings, *X* could only be a single character. That's now changed; *X* can be a string of any length, and `X in Y` will return `True` if *X* is a substring of *Y*. If *X* is the empty string, the result is always `True`. :

  ```
  >>> 'ab' in 'abcd'
  True
  >>> 'ad' in 'abcd'
  False
  >>> '' in 'abcd'
  True
  ```

  Note that this doesn't tell you where the substring starts; if you need that information, use the `~str.find` string method.
- The `~str.strip`, `~str.lstrip`, and `~str.rstrip` string methods now have an optional argument for specifying the characters to strip. The default is still to remove all whitespace characters:

  ```
  >>> '   abc '.strip()
  'abc'
  >>> '><><abc<><><>'.strip('<>')
  'abc'
  >>> '><><abc<><><>\n'.strip('<>')
  'abc<><><>\n'
  >>> u'\u4000\u4001abc\u4000'.strip(u'\u4000')
  u'\u4001abc'
  >>>
  ```

  (Suggested by Simon Brunning and implemented by Walter Dörwald.)
- The `~str.startswith` and `~str.endswith` string methods now accept negative numbers for the *start* and *end* parameters.
- Another new string method is `~str.zfill`, originally a function in the `string` module. `~str.zfill` pads a numeric string with zeros on the left until it's the specified width. Note

that the `%` operator is still more flexible and powerful than `~str.zfill`. :

```
>>> '45'.zfill(4)
'0045'
>>> '12345'.zfill(4)
'12345'
>>> 'goofy'.zfill(6)
'0goofy'
```

(Contributed by Walter Dörwald.)

- A new type object, `!basestring`, has been added. Both 8-bit strings and Unicode strings inherit from this type, so `isinstance(obj, basestring)` will return `True` for either kind of string. It's a completely abstract type, so you can't create `!basestring` instances.
- Interned strings are no longer immortal and will now be garbage-collected in the usual way when the only reference to them is from the internal dictionary of interned strings. (Implemented by Oren Tirosh.)

Optimizations

- The creation of new-style class instances has been made much faster; they're now faster than classic classes!
- The `~list.sort` method of list objects has been extensively rewritten by Tim Peters, and the implementation is significantly faster.
- Multiplication of large long integers is now much faster thanks to an implementation of Karatsuba multiplication, an algorithm that scales better than the $O(n^2)$ required for the grade-school multiplication algorithm. (Original patch by Christopher A. Craig, and significantly reworked by Tim Peters.)
- The `SET_LINENO` opcode is now gone. This may provide a small speed increase, depending on your compiler's idiosyncrasies. See section `23section-other` for a longer explanation. (Removed by Michael Hudson.)
- `!xrange` objects now have their own iterator, making `for i in xrange(n)` slightly faster than `for i in range(n)`. (Patch by Raymond Hettinger.)
- A number of small rearrangements have been made in various hotspots to improve performance, such as inlining a function or removing some code. (Implemented mostly by GvR, but lots of people have contributed single changes.)

The net result of the 2.3 optimizations is that Python 2.3 runs the pystone benchmark around 25% faster than Python 2.2.

New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the `Misc/NEWS` file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The `array` module now supports arrays of Unicode characters using the `'u'` format character. Arrays also now support using the `+=` assignment operator to add another array's contents, and the `*=` assignment operator to repeat an array. (Contributed by Jason Orendorff.)
- The `!bsddb` module has been replaced by version 4.1.6 of the PyBSDDB package, providing a more complete interface to the transactional features of the BerkeleyDB library.
  The old version of the module has been renamed to `!bsddb185` and is no longer built automatically; you'll have to edit `Modules/Setup` to enable it. Note that the new `!bsddb` package is intended to be compatible with the old module, so be sure to file bugs if you discover any incompatibilities. When upgrading to Python 2.3, if the new interpreter is compiled with a new version of the underlying BerkeleyDB library, you will almost certainly have to convert your database files to the new version. You can do this fairly easily with the new scripts `db2pickle.py` and `pickle2db.py` which you will find in the distribution's `Tools/scripts` directory. If you've already been using the PyBSDDB package and importing it as `!bsddb3`, you will have to change your `import` statements to import it as `!bsddb`.
- The new `bz2` module is an interface to the bz2 data compression library. bz2-compressed data is usually smaller than corresponding `zlib`-compressed data. (Contributed by Gustavo Niemeyer.)
- A set of standard date/time types has been added in the new `datetime` module. See the following section for more details.
- The Distutils `!Extension` class now supports an extra constructor argument named *depends* for listing additional source files that an extension depends on. This lets Distutils recompile the module if any of the dependency files are modified. For example, if `sampmodule.c` includes the header file `sample.h`, you would create the `!Extension` object like this:

  ```
  ext = Extension("samp",
                  sources=["sampmodule.c"],
                  depends=["sample.h"])
  ```

  Modifying `sample.h` would then cause the module to be recompiled. (Contributed by Jeremy Hylton.)

- Other minor changes to Distutils: it now checks for the `CC`, `CFLAGS`, `!CPP`, `LDFLAGS`, and `CPPFLAGS` environment variables, using them to override the settings in Python's configuration (contributed by Robert Weber).
- Previously the `doctest` module would only search the docstrings of public methods and functions for test cases, but it now also examines private ones as well. The `~doctest.DocTestSuite` function creates a `unittest.TestSuite` object from a set of `doctest` tests.
- The new `gc.get_referents(object)` function returns a list of all the objects referenced by *object*.
- The `getopt` module gained a new function, `~getopt.gnu_getopt`, that supports the same arguments as the existing `~getopt.getopt` function but uses GNU-style scanning mode. The existing `~getopt.getopt` stops processing options as soon as a non-option argument is encountered, but in GNU-style mode processing continues, meaning that options and arguments can be mixed. For example:

  ```
  >>> getopt.getopt(['-f', 'filename', 'output', '-v'], 'f:v')
  ([('-f', 'filename')], ['output', '-v'])
  >>> getopt.gnu_getopt(['-f', 'filename', 'output', '-v'], 'f:v')
  ([('-f', 'filename'), ('-v', '')], ['output'])
  ```

  (Contributed by Peter Åstrand.)
- The `grp`, `pwd`, and `resource` modules now return enhanced tuples:

  ```
  >>> import grp
  >>> g = grp.getgrnam('amk')
  >>> g.gr_name, g.gr_gid
  ('amk', 500)
  ```

- The `gzip` module can now handle files exceeding 2 GiB.
- The new `heapq` module contains an implementation of a heap queue algorithm. A heap is an array-like data structure that keeps items in a partially sorted order such that, for every index *k*, `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]`. This makes it quick to remove the smallest item, and inserting a new item while maintaining the heap property is *O*(log *n*). (See https://xlinux.nist.gov/dads//HTML/priorityque.html for more information about the priority queue data structure.)
  The `heapq` module provides `~heapq.heappush` and `~heapq.heappop` functions for adding and removing items while maintaining the heap property on top of some other mutable Python sequence type. Here's an example that uses a Python list:

  ```
  >>> import heapq
  >>> heap = []
  >>> for item in [3, 7, 5, 11, 1]:
  ...     heapq.heappush(heap, item)
  ...
  ```

```
>>> heap
[1, 3, 5, 11, 7]
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
3
>>> heap
[5, 7, 11]
```

(Contributed by Kevin O'Connor.)

- The IDLE integrated development environment has been updated using the code from the IDLEfork project (https://idlefork.sourceforge.net). The most notable feature is that the code being developed is now executed in a subprocess, meaning that there's no longer any need for manual `reload()` operations. IDLE's core code has been incorporated into the standard library as the `idlelib` package.
- The `imaplib` module now supports IMAP over SSL. (Contributed by Piers Lauder and Tino Lange.)
- The `itertools` contains a number of useful functions for use with iterators, inspired by various functions provided by the ML and Haskell languages. For example, `itertools.ifilter(predicate, iterator)` returns all elements in the iterator for which the function `!predicate` returns `True`, and `itertools.repeat(obj, N)` returns `obj` *N* times. There are a number of other functions in the module; see the package's reference documentation for details. (Contributed by Raymond Hettinger.)
- Two new functions in the `math` module, `degrees(rads)` and `radians(degs)`, convert between radians and degrees. Other functions in the `math` module such as `math.sin` and `math.cos` have always required input values measured in radians. Also, an optional *base* argument was added to `math.log` to make it easier to compute logarithms for bases other than `e` and `10`. (Contributed by Raymond Hettinger.)
- Several new POSIX functions (`!getpgid`, `!killpg`, `!lchown`, `!loadavg`, `!major`, `!makedev`, `!minor`, and `!mknod`) were added to the `posix` module that underlies the `os` module. (Contributed by Gustavo Niemeyer, Geert Jansen, and Denis S. Otkidach.)
- In the `os` module, the `!\*stat` family of functions can now report fractions of a second in a timestamp. Such time stamps are represented as floats, similar to the value returned by `time.time`.
  During testing, it was found that some applications will break if time stamps are floats. For compatibility, when using the tuple interface of the `~os.stat_result` time stamps will be represented as integers. When using named fields (a feature first introduced in Python 2.2), time stamps are still represented as integers, unless `!os.stat_float_times` is invoked to enable float return values:

```
>>> os.stat("/tmp").st_mtime
1034791200
>>> os.stat_float_times(True)
```

```
>>> os.stat("/tmp").st_mtime
1034791200.6335014
```

In Python 2.4, the default will change to always returning floats.
Application developers should enable this feature only if all their libraries work properly when confronted with floating-point time stamps, or if they use the tuple API. If used, the feature should be activated on an application level instead of trying to enable it on a per-use basis.

- The `optparse` module contains a new parser for command-line arguments that can convert option values to a particular Python type and will automatically generate a usage message. See the following section for more details.
- The old and never-documented `!linuxaudiodev` module has been deprecated, and a new version named `!ossaudiodev` has been added. The module was renamed because the OSS sound drivers can be used on platforms other than Linux, and the interface has also been tidied and brought up to date in various ways. (Contributed by Greg Ward and Nicholas FitzRoy-Dale.)
- The new `platform` module contains a number of functions that try to determine various properties of the platform you're running on. There are functions for getting the architecture, CPU type, the Windows OS version, and even the Linux distribution version. (Contributed by Marc-André Lemburg.)
- The parser objects provided by the `pyexpat <xml.parsers.expat>` module can now optionally buffer character data, resulting in fewer calls to your character data handler and therefore faster performance. Setting the parser object's `~xml.parsers.expat.xmlparser.buffer_text` attribute to `True` will enable buffering.
- The `sample(population, k)` function was added to the `random` module. *population* is a sequence or `!xrange` object containing the elements of a population, and `~random.sample` chooses *k* elements from the population without replacing chosen elements. *k* can be any value up to `len(population)`. For example:

```
>>> days = ['Mo', 'Tu', 'We', 'Th', 'Fr', 'St', 'Sn']
>>> random.sample(days, 3)      # Choose 3 elements
['St', 'Sn', 'Th']
>>> random.sample(days, 7)      # Choose 7 elements
['Tu', 'Th', 'Mo', 'We', 'St', 'Fr', 'Sn']
>>> random.sample(days, 7)      # Choose 7 again
['We', 'Mo', 'Sn', 'Fr', 'Tu', 'St', 'Th']
>>> random.sample(days, 8)      # Can't choose eight
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "random.py", line 414, in sample
      raise ValueError, "sample larger than population"
ValueError: sample larger than population
>>> random.sample(xrange(1,10000,2), 10)   # Choose ten odd nos. under 10000
```

```
[3407, 3805, 1505, 7023, 2401, 2267, 9733, 3151, 8083, 9195]
```

The `random` module now uses a new algorithm, the Mersenne Twister, implemented in C. It's faster and more extensively studied than the previous algorithm.
(All changes contributed by Raymond Hettinger.)

- The `readline` module also gained a number of new functions: `~readline.get_history_item`, `~readline.get_current_history_length`, and `~readline.redisplay`.
- The `!rexec` and `!Bastion` modules have been declared dead, and attempts to import them will fail with a `RuntimeError`. New-style classes provide new ways to break out of the restricted execution environment provided by `!rexec`, and no one has interest in fixing them or time to do so. If you have applications using `!rexec`, rewrite them to use something else.
  (Sticking with Python 2.2 or 2.1 will not make your applications any safer because there are known bugs in the `!rexec` module in those versions. To repeat: if you're using `!rexec`, stop using it immediately.)
- The `!rotor` module has been deprecated because the algorithm it uses for encryption is not believed to be secure. If you need encryption, use one of the several AES Python modules that are available separately.
- The `shutil` module gained a `move(src, dest)` function that recursively moves a file or directory to a new location.
- Support for more advanced POSIX signal handling was added to the `signal` but then removed again as it proved impossible to make it work reliably across platforms.
- The `socket` module now supports timeouts. You can call the `settimeout(t)` method on a socket object to set a timeout of *t* seconds. Subsequent socket operations that take longer than *t* seconds to complete will abort and raise a `socket.timeout` exception.
  The original timeout implementation was by Tim O'Malley. Michael Gilfix integrated it into the Python `socket` module and shepherded it through a lengthy review. After the code was checked in, Guido van Rossum rewrote parts of it. (This is a good example of a collaborative development process in action.)
- On Windows, the `socket` module now ships with Secure Sockets Layer (SSL) support.
- The value of the C `!PYTHON_API_VERSION` macro is now exposed at the Python level as `sys.api_version`. The current exception can be cleared by calling the new `!sys.exc_clear` function.
- The new `tarfile` module allows reading from and writing to `tar`-format archive files. (Contributed by Lars Gustäbel.)
- The new `textwrap` module contains functions for wrapping strings containing paragraphs of text. The `wrap(text, width)` function takes a string and returns a list containing the text split into lines of no more than the chosen width. The `fill(text, width)` function returns a single string, reformatted to fit into lines no longer than the chosen width. (As you can guess, `~textwrap.fill` is built on top of `~textwrap.wrap`. For example:

```
>>> import textwrap
>>> paragraph = "Not a whit, we defy augury: ... more text ..."
>>> textwrap.wrap(paragraph, 60)
```

```
["Not a whit, we defy augury: there's a special providence in",
 "the fall of a sparrow. If it be now, 'tis not to come; if it",
 ...]
>>> print textwrap.fill(paragraph, 35)
Not a whit, we defy augury: there's
a special providence in the fall of
a sparrow. If it be now, 'tis not
to come; if it be not to come, it
will be now; if it be not now, yet
it will come: the readiness is all.
>>>
```

The module also contains a `~textwrap.TextWrapper` class that actually implements the text wrapping strategy. Both the `~textwrap.TextWrapper` class and the `~textwrap.wrap` and `~textwrap.fill` functions support a number of additional keyword arguments for fine-tuning the formatting; consult the module's documentation for details. (Contributed by Greg Ward.)

- The `!thread` and `threading` modules now have companion modules, `!dummy_thread` and `!dummy_threading`, that provide a do-nothing implementation of the `!thread` module's interface for platforms where threads are not supported. The intention is to simplify thread-aware modules (ones that *don't* rely on threads to run) by putting the following code at the top:

```
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

In this example, `!_threading` is used as the module name to make it clear that the module being used is not necessarily the actual `threading` module. Code can call functions and use classes in `!_threading` whether or not threads are supported, avoiding an `if` statement and making the code slightly clearer. This module will not magically make multithreaded code run without threads; code that waits for another thread to return or to do something will simply hang forever.

- The `time` module's `~time.strptime` function has long been an annoyance because it uses the platform C library's `~time.strptime` implementation, and different platforms sometimes have odd bugs. Brett Cannon contributed a portable implementation that's written in pure Python and should behave identically on all platforms.
- The new `timeit` module helps measure how long snippets of Python code take to execute. The `timeit.py` file can be run directly from the command line, or the module's `~timeit.Timer` class can be imported and used directly. Here's a short example that figures out whether it's faster to convert an 8-bit string to Unicode by appending an empty Unicode string to it or by using the `!unicode` function:

```
import timeit

timer1 = timeit.Timer('unicode("abc")')
timer2 = timeit.Timer('"abc" + u""')

# Run three trials
print timer1.repeat(repeat=3, number=100000)
print timer2.repeat(repeat=3, number=100000)

# On my laptop this outputs:
# [0.36831796169281006, 0.37441694736480713, 0.35304892063140869]
# [0.17574405670166016, 0.18193507194519043, 0.17565798759460449]
```

- The `!Tix` module has received various bug fixes and updates for the current version of the Tix package.
- The `!Tkinter` module now works with a thread-enabled version of Tcl. Tcl's threading model requires that widgets only be accessed from the thread in which they're created; accesses from another thread can cause Tcl to panic. For certain Tcl interfaces, `!Tkinter` will now automatically avoid this when a widget is accessed from a different thread by marshalling a command, passing it to the correct thread, and waiting for the results. Other interfaces can't be handled automatically but `!Tkinter` will now raise an exception on such an access so that you can at least find out about the problem. See
  https://mail.python.org/pipermail/python-dev/2002-December/031107.html for a more detailed explanation of this change. (Implemented by Martin von Löwis.)
- Calling Tcl methods through `!_tkinter` no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a `!_tkinter.Tcl_Obj` object if no Python equivalent exists. This behavior can be controlled through the `!wantobjects` method of `!tkapp` objects.
  When using `!_tkinter` through the `!Tkinter` module (as most Tkinter applications will), this feature is always activated. It should not cause compatibility problems, since Tkinter would always convert string results to Python types where possible.
  If any incompatibilities are found, the old behavior can be restored by setting the `!wantobjects` variable in the `!Tkinter` module to false before creating the first `!tkapp` object. :

  ```
  import Tkinter
  Tkinter.wantobjects = 0
  ```

  Any breakage caused by this change should be reported as a bug.
- The `!UserDict` module has a new `!DictMixin` class which defines all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries, such as the classes in the `shelve` module.

Adding the mix-in as a superclass provides the full dictionary interface whenever the class defines `~object.__getitem__`, `~object.__setitem__`, `~object.__delitem__`, and `!keys`. For example:

```
>>> import UserDict
>>> class SeqDict(UserDict.DictMixin):
...     """Dictionary lookalike implemented with lists."""
...     def __init__(self):
...         self.keylist = []
...         self.valuelist = []
...     def __getitem__(self, key):
...         try:
...             i = self.keylist.index(key)
...         except ValueError:
...             raise KeyError
...         return self.valuelist[i]
...     def __setitem__(self, key, value):
...         try:
...             i = self.keylist.index(key)
...             self.valuelist[i] = value
...         except ValueError:
...             self.keylist.append(key)
...             self.valuelist.append(value)
...     def __delitem__(self, key):
...         try:
...             i = self.keylist.index(key)
...         except ValueError:
...             raise KeyError
...         self.keylist.pop(i)
...         self.valuelist.pop(i)
...     def keys(self):
...         return list(self.keylist)
...
>>> s = SeqDict()
>>> dir(s)      # See that other dictionary methods are implemented
['__cmp__', '__contains__', '__delitem__', '__doc__', '__getitem__',
 '__init__', '__iter__', '__len__', '__module__', '__repr__',
 '__setitem__', 'clear', 'get', 'has_key', 'items', 'iteritems',
 'iterkeys', 'itervalues', 'keylist', 'keys', 'pop', 'popitem',
 'setdefault', 'update', 'valuelist', 'values']
```

(Contributed by Raymond Hettinger.)

- The DOM implementation in `xml.dom.minidom` can now generate XML output in a particular encoding by providing an optional encoding argument to the `~xml.dom.minidom.Node.toxml` and `~xml.dom.minidom.Node.toprettyxml` methods of DOM nodes.
- The `!xmlrpclib` module now supports an XML-RPC extension for handling nil data values such as Python's `None`. Nil values are always supported on unmarshalling an XML-RPC response. To generate requests containing `None`, you must supply a true value for the *allow_none* parameter when creating a `!Marshaller` instance.
- The new `!DocXMLRPCServer` module allows writing self-documenting XML-RPC servers. Run it in demo mode (as a program) to see it in action. Pointing the web browser to the RPC server produces pydoc-style documentation; pointing xmlrpclib to the server allows invoking the actual methods. (Contributed by Brian Quinlan.)
- Support for internationalized domain names (RFCs 3454, 3490, 3491, and 3492) has been added. The "idna" encoding can be used to convert between a Unicode domain name and the ASCII-compatible encoding (ACE) of that name. :

  ```
  >{}>{}> u"www.Alliancefrançaise.nu".encode("idna")
  'www.xn--alliancefranaise-npb.nu'
  ```

  The `socket` module has also been extended to transparently convert Unicode hostnames to the ACE version before passing them to the C library. Modules that deal with hostnames such as `!httplib` and `ftplib`) also support Unicode host names; `!httplib` also sends HTTP `Host` headers using the ACE version of the domain name. `urllib` supports Unicode URLs with non-ASCII host names as long as the `path` part of the URL is ASCII only.
  To implement this change, the `stringprep` module, the `mkstringprep` tool and the `punycode` encoding have been added.

Date/Time Type

Date and time types suitable for expressing timestamps were added as the `datetime` module. The types don't support different calendars or many fancy features, and just stick to the basics of representing time.

The three primary types are: `~datetime.date`, representing a day, month, and year; `~datetime.time`, consisting of hour, minute, and second; and `~datetime.datetime`, which contains all the attributes of both `~datetime.date` and `~datetime.time`. There's also a `~datetime.timedelta` class representing differences between two points in time, and time zone logic is implemented by classes inheriting from the abstract `~datetime.tzinfo` class.

You can create instances of `~datetime.date` and `~datetime.time` by either supplying keyword arguments to the appropriate constructor, e.g. `datetime.date(year=1972, month=10, day=15)`, or by using one of a number of class methods. For example, the `~datetime.date.today` class method returns the current local date.

Once created, instances of the date/time classes are all immutable. There are a number of methods for producing formatted strings from objects:

```
>>> import datetime as dt
>>> now = dt.datetime.now()
>>> now.isoformat()
'2002-12-30T21:27:03.994956'
>>> now.ctime()  # Only available on date, datetime
'Mon Dec 30 21:27:03 2002'
>>> now.strftime('%Y %d %b')
'2002 30 Dec'
```

The `~datetime.datetime.replace` method allows modifying one or more fields of a `~datetime.date` or `~datetime.datetime` instance, returning a new instance:

```
>>> d = dt.datetime.now()
>>> d
datetime.datetime(2002, 12, 30, 22, 15, 38, 827738)
>>> d.replace(year=2001, hour=12)
datetime.datetime(2001, 12, 30, 12, 15, 38, 827738)
>>>
```

Instances can be compared, hashed, and converted to strings (the result is the same as that of `~datetime.datetime.isoformat`). `~datetime.date` and `~datetime.datetime` instances can be subtracted from each other, and added to `~datetime.timedelta` instances. The largest missing feature is that there's no standard library support for parsing strings and getting back a `~datetime.date` or `~datetime.datetime`.

For more information, refer to the module's reference documentation. (Contributed by Tim Peters.)

The optparse Module

The `getopt` module provides simple parsing of command-line arguments. The new `optparse` module (originally named Optik) provides more elaborate command-line parsing that follows the Unix conventions, automatically creates the output for `!--help`, and can perform different actions for different options.

You start by creating an instance of `~optparse.OptionParser` and telling it what your program's options are. :

```
import sys
from optparse import OptionParser
```

```
op = OptionParser()
op.add_option('-i', '--input',
              action='store', type='string', dest='input',
              help='set input filename')
op.add_option('-l', '--length',
              action='store', type='int', dest='length',
              help='set maximum length of output')
```

Parsing a command line is then done by calling the `~optparse.OptionParser.parse_args` method. :

```
options, args = op.parse_args(sys.argv[1:])
print options
print args
```

This returns an object containing all of the option values, and a list of strings containing the remaining arguments.

Invoking the script with the various arguments now works as you'd expect it to. Note that the length argument is automatically converted to an integer.

```
$ ./python opt.py -i data arg1
<Values at 0x400cad4c: {'input': 'data', 'length': None}>
['arg1']
$ ./python opt.py --input=data --length=4
<Values at 0x400cad2c: {'input': 'data', 'length': 4}>
[]
$
```

The help message is automatically generated for you:

```
$ ./python opt.py --help
usage: opt.py [options]

options:
  -h, --help            show this help message and exit
  -iINPUT, --input=INPUT
                        set input filename
  -lLENGTH, --length=LENGTH
                        set maximum length of output
$
```

See the module's documentation for more details.

Optik was written by Greg Ward, with suggestions from the readers of the Getopt SIG.

Pymalloc: A Specialized Object Allocator

Pymalloc, a specialized object allocator written by Vladimir Marangozov, was a feature added to Python 2.1. Pymalloc is intended to be faster than the system `malloc` and to have less memory overhead for allocation patterns typical of Python programs. The allocator uses C's `malloc` function to get large pools of memory and then fulfills smaller memory requests from these pools.

In 2.1 and 2.2, pymalloc was an experimental feature and wasn't enabled by default; you had to explicitly enable it when compiling Python by providing the `!--with-pymalloc` option to the `configure` script. In 2.3, pymalloc has had further enhancements and is now enabled by default; you'll have to supply `!--without-pymalloc` to disable it.

This change is transparent to code written in Python; however, pymalloc may expose bugs in C extensions. Authors of C extension modules should test their code with pymalloc enabled, because some incorrect code may cause core dumps at runtime.

There's one particularly common error that causes problems. There are a number of memory allocation functions in Python's C API that have previously just been aliases for the C library's `malloc` and `free`, meaning that if you accidentally called mismatched functions the error wouldn't be noticeable. When the object allocator is enabled, these functions aren't aliases of `malloc` and `free` any more, and calling the wrong function to free memory may get you a core dump. For example, if memory was allocated using `PyObject_Malloc`, it has to be freed using `PyObject_Free`, not `free`. A few modules included with Python fell afoul of this and had to be fixed; doubtless there are more third-party modules that will have the same problem.

As part of this change, the confusing multiple interfaces for allocating memory have been consolidated down into two API families. Memory allocated with one family must not be manipulated with functions from the other family. There is one family for allocating chunks of memory and another family of functions specifically for allocating Python objects.

- To allocate and free an undistinguished chunk of memory use the "raw memory" family: `PyMem_Malloc`, `PyMem_Realloc`, and `PyMem_Free`.
- The "object memory" family is the interface to the pymalloc facility described above and is biased towards a large number of "small" allocations: `PyObject_Malloc`, `PyObject_Realloc`, and `PyObject_Free`.
- To allocate and free Python objects, use the "object" family `PyObject_New`, `PyObject_NewVar`, and `PyObject_Del`.

Thanks to lots of work by Tim Peters, pymalloc in 2.3 also provides debugging features to catch memory overwrites and doubled frees in both extension modules and in the interpreter itself. To enable this support, compile a debugging version of the Python interpreter by running `configure` with `!--with-pydebug`.

To aid extension writers, a header file `Misc/pymemcompat.h` is distributed with the source to Python 2.3 that allows Python extensions to use the 2.3 interfaces to memory allocation while compiling against any version of Python since 1.5.2. You would copy the file from Python's source distribution and bundle it with the source of your extension.

https://hg.python.org/cpython/file/default/Objects/obmalloc.c
For the full details of the pymalloc implementation, see the comments at the top of the file `Objects/obmalloc.c` in the Python source code. The above link points to the file within the python.org SVN browser.

Build and C API Changes

Changes to Python's build process and to the C API include:

- The cycle detection implementation used by the garbage collection has proven to be stable, so it's now been made mandatory. You can no longer compile Python without it, and the `!--with-cycle-gc` switch to `configure` has been removed.
- Python can now optionally be built as a shared library (`libpython2.3.so`) by supplying `!--enable-shared` when running Python's `configure` script. (Contributed by Ondrej Palkovsky.)
- The `!DL_EXPORT` and `!DL_IMPORT` macros are now deprecated. Initialization functions for Python extension modules should now be declared using the new macro `PyMODINIT_FUNC`, while the Python core will generally use the `!PyAPI_FUNC` and `!PyAPI_DATA` macros.
- The interpreter can be compiled without any docstrings for the built-in functions and modules by supplying `!--without-doc-strings` to the `configure` script. This makes the Python executable about 10% smaller, but will also mean that you can't get help for Python's built-ins. (Contributed by Gustavo Niemeyer.)
- The `!PyArg_NoArgs` macro is now deprecated, and code that uses it should be changed. For Python 2.2 and later, the method definition table can specify the `METH_NOARGS` flag, signalling that there are no arguments, and the argument checking can then be removed. If compatibility with pre-2.2 versions of Python is important, the code could use `PyArg_ParseTuple(args, "")` instead, but this will be slower than using `METH_NOARGS`.
- `PyArg_ParseTuple` accepts new format characters for various sizes of unsigned integers: `B` for `unsigned char`, `H` for `unsigned short int`, `I` for `unsigned int`, and `K` for `unsigned long long`.
- A new function, `PyObject_DelItemString(mapping, char *key)` was added as shorthand for `PyObject_DelItem(mapping, PyString_New(key))`.
- File objects now manage their internal string buffer differently, increasing it exponentially when needed. This results in the benchmark tests in `Lib/test/test_bufio.py` speeding up

considerably (from 57 seconds to 1.7 seconds, according to one measurement).
- It's now possible to define class and static methods for a C extension type by setting either the `METH_CLASS` or `METH_STATIC` flags in a method's `PyMethodDef` structure.
- Python now includes a copy of the Expat XML parser's source code, removing any dependence on a system version or local installation of Expat.
- If you dynamically allocate type objects in your extension, you should be aware of a change in the rules relating to the `~type.__module__` and `~type.__name__` attributes. In summary, you will want to ensure the type's dictionary contains a `'__module__'` key; making the module name the part of the type name leading up to the final period will no longer have the desired effect. For more detail, read the API reference documentation or the source.

Port-Specific Changes

Support for a port to IBM's OS/2 using the EMX runtime environment was merged into the main Python source tree. EMX is a POSIX emulation layer over the OS/2 system APIs. The Python port for EMX tries to support all the POSIX-like capability exposed by the EMX runtime, and mostly succeeds; `!fork` and `~fcntl.fcntl` are restricted by the limitations of the underlying emulation layer. The standard OS/2 port, which uses IBM's Visual Age compiler, also gained support for case-sensitive import semantics as part of the integration of the EMX port into CVS. (Contributed by Andrew MacIntyre.)

On MacOS, most toolbox modules have been weaklinked to improve backward compatibility. This means that modules will no longer fail to load if a single routine is missing on the current OS version. Instead calling the missing routine will raise an exception. (Contributed by Jack Jansen.)

The RPM spec files, found in the `Misc/RPM/` directory in the Python source distribution, were updated for 2.3. (Contributed by Sean Reifschneider.)

Other new platforms now supported by Python include AtheOS (http://www.atheos.cx/), GNU/Hurd, and OpenVMS.

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 523 patches applied and 514 bugs fixed between Python 2.2 and 2.3. Both figures are likely to be underestimates.

Some of the more notable changes are:

- If the `PYTHONINSPECT` environment variable is set, the Python interpreter will enter the interactive prompt after running a Python program, as if Python had been invoked with

the `-i` option. The environment variable can be set before running the Python interpreter, or it can be set by the Python program as part of its execution.
- The `regrtest.py` script now provides a way to allow "all resources except *foo*." A resource name passed to the `!-u` option can now be prefixed with a hyphen (`'-'`) to mean "remove this resource." For example, the option '`-uall,-bsddb`' could be used to enable the use of all resources except `bsddb`.
- The tools used to build the documentation now work under Cygwin as well as Unix.
- The `SET_LINENO` opcode has been removed. Back in the mists of time, this opcode was needed to produce line numbers in tracebacks and support trace functions (for, e.g., `pdb`). Since Python 1.5, the line numbers in tracebacks have been computed using a different mechanism that works with "python -O". For Python 2.3 Michael Hudson implemented a similar scheme to determine when to call the trace function, removing the need for `SET_LINENO` entirely.
  It would be difficult to detect any resulting difference from Python code, apart from a slight speed up when Python is run without `-O`.
  C extensions that access the `~frame.f_lineno` field of frame objects should instead call `PyCode_Addr2Line(f->f_code, f->f_lasti)`. This will have the added effect of making the code work as desired under "python -O" in earlier versions of Python.
  A nifty new feature is that trace functions can now assign to the `~frame.f_lineno` attribute of frame objects, changing the line that will be executed next. A `jump` command has been added to the `pdb` debugger taking advantage of this new feature. (Implemented by Richie Hindle.)

Porting to Python 2.3

This section lists previously described changes that may require changes to your code:

- `yield` is now always a keyword; if it's used as a variable name in your code, a different name must be chosen.
- For strings *X* and *Y*, `X in Y` now works if *X* is more than one character long.
- The `int` type constructor will now return a long integer instead of raising an `OverflowError` when a string or floating-point number is too large to fit into an integer.
- If you have Unicode strings that contain 8-bit characters, you must declare the file's encoding (UTF-8, Latin-1, or whatever) by adding a comment to the top of the file. See section `section-encodings` for more information.
- Calling Tcl methods through `!_tkinter` no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a `!_tkinter.Tcl_Obj` object if no Python equivalent exists.
- Large octal and hex literals such as `0xffffffff` now trigger a `FutureWarning`. Currently they're stored as 32-bit numbers and result in a negative value, but in Python 2.4 they'll become positive long integers.
  There are a few ways to fix this warning. If you really need a positive number, just add an `L` to the end of the literal. If you're trying to get a 32-bit integer with low bits set and

have previously used an expression such as `~(1 << 31)`, it's probably clearest to start with all bits set and clear the desired upper bits. For example, to clear just the top bit (bit 31), you could write `0xffffffffL &~(1L<<31)`.

- You can no longer disable assertions by assigning to `__debug__`.
- The Distutils `!setup` function has gained various new keyword arguments such as *depends*. Old versions of the Distutils will abort if passed unknown keywords. A solution is to check for the presence of the new `!get_distutil_options` function in your `setup.py` and only uses the new keywords with a version of the Distutils that supports them:

```
from distutils import core


kw = {'sources': 'foo.c', ...}
if hasattr(core, 'get_distutil_options'):
    kw['depends'] = ['foo.h']
ext = Extension(**kw)
```

- Using `None` as a variable name will now result in a `SyntaxWarning` warning.
- Names of extension types defined by the modules included with Python now contain the module and a `'.'` in front of the type name.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Jeff Bauer, Simon Brunning, Brett Cannon, Michael Chermside, Andrew Dalke, Scott David Daniels, Fred L. Drake, Jr., David Fraser, Kelly Gerber, Raymond Hettinger, Michael Hudson, Chris Lambert, Detlef Lannert, Martin von Löwis, Andrew MacIntyre, Lalo Martins, Chad Netzer, Gustavo Niemeyer, Neal Norwitz, Hans Nowak, Chris Reedy, Francesco Ricciardi, Vinay Sajip, Neil Schemenauer, Roman Suzi, Jason Tishler, Just van Rossum.

□□□ #2
□ □□□ □□□ 2026-10-31 17:28:11 UTC
□ □□□ □□□ 2026-10-31 17:28:14 UTC