What's New in Python

Release 2.7.11rc1

A. M. Kuchling

December 03, 2015

Python Software Foundation
Email: docs@python.org

Contents

Author A.M. Kuchling (amk at amk.ca)

This article explains the new features in Python 2.7. Python 2.7 was released on July 3, 2010.

Numeric handling has been improved in many ways, for both floating-point numbers and for the `Decimal` class. There are some useful additions to the standard library, such as a greatly enhanced `unittest` module, the `argparse` module for parsing command-line options, convenient `OrderedDict` and `Counter` classes in the `collections` module, and many other improvements.

Python 2.7 is planned to be the last of the 2.x releases, so we worked on making it a good release for the long term. To help with porting to Python 3, several new features from the Python 3.x series have been included in 2.7.

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 2.7 at https://docs.python.org. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature or the issue on https://bugs.python.org in which a change was discussed. Whenever possible, "What's New in Python" links to the bug/patch item for each change.

1 The Future for Python 2.x

Python 2.7 is the last major release in the 2.x series, as the Python maintainers have shifted the focus of their new feature development efforts to the Python 3.x series. This means that while Python 2 continues to receive bug fixes, and to be updated to build correctly on new hardware and versions of supported operated systems, there will be no new full feature releases for the language or standard library.

However, while there is a large common subset between Python 2.7 and Python 3, and many of the changes involved in migrating to that common subset, or directly to Python 3, can be safely automated, some other changes (notably those associated with Unicode handling) may require careful consideration, and preferably robust automated regression test suites, to migrate effectively.

This means that Python 2.7 will remain in place for a long time, providing a stable and supported base platform for production systems that have not yet been ported to Python 3. The full expected lifecycle of the Python 2.7 series is detailed in **PEP 373**.

Some key consequences of the long-term significance of 2.7 are:

- As noted above, the 2.7 release has a much longer period of maintenance when compared to earlier 2.x versions. Python 2.7 is currently expected to remain supported by the core development team (receiving security updates and other bug fixes) until at least 2020 (10 years after its initial release, compared to the more typical support period of 18-24 months).

- As the Python 2.7 standard library ages, making effective use of the Python Package Index (either directly or via a redistributor) becomes more important for Python 2 users. In addition to a wide variety of third party packages for various tasks, the available packages include backports of new modules and features from the Python 3 standard library that are compatible with Python 2, as well as various tools and libraries that can make it easier to migrate to Python 3. The Python Packaging User Guide provides guidance on downloading and installing software from the Python Package Index.
- While the preferred approach to enhancing Python 2 is now the publication of new packages on the Python Package Index, this approach doesn't necessarily work in all cases, especially those related to network security. In exceptional cases that cannot be handled adequately by publishing new or updated packages on PyPI, the Python Enhancement Proposal process may be used to make the case for adding new features directly to the Python 2 standard library. Any such additions, and the maintenance releases where they were added, will be noted in the *New Features Added to Python 2.7 Maintenance Releases* section below.

For projects wishing to migrate from Python 2 to Python 3, or for library and framework developers wishing to support users on both Python 2 and Python 3, there are a variety of tools and guides available to help decide on a suitable approach and manage some of the technical details involved. The recommended starting point is the *pyporting-howto* HOWTO guide.

2 Changes to the Handling of Deprecation Warnings

For Python 2.7, a policy decision was made to silence warnings only of interest to developers by default. `DeprecationWarning` and its descendants are now ignored unless otherwise requested, preventing users from seeing warnings triggered by an application. This change was also made in the branch that became Python 3.2. (Discussed on stdlib-sig and carried out in issue 7319.)

In previous releases, `DeprecationWarning` messages were enabled by default, providing Python developers with a clear indication of where their code may break in a future major version of Python.

However, there are increasingly many users of Python-based applications who are not directly involved in the development of those applications. `DeprecationWarning` messages are irrelevant to such users, making them worry about an application that's actually working correctly and burdening application developers with responding to these concerns.

You can re-enable display of `DeprecationWarning` messages by running Python with the *`-Wdefault`* (short form: *`-Wd`*) switch, or by setting the `PYTHONWARNINGS` environment variable to `"default"` (or `"d"`) before running Python. Python code can also re-enable them by calling `warnings.simplefilter('default')`.

The `unittest` module also automatically reenables deprecation warnings when running tests.

3 Python 3.1 Features

Much as Python 2.6 incorporated features from Python 3.0, version 2.7 incorporates some of the new features in Python 3.1. The 2.x series continues to provide tools for migrating to the 3.x series.

A partial list of 3.1 features that were backported to 2.7:

- The syntax for set literals (`{1,2,3}` is a mutable set).
- Dictionary and set comprehensions (`{i: i*2 for i in range(3)}`).
- Multiple context managers in a single `with` statement.
- A new version of the `io` library, rewritten in C for performance.
- The ordered-dictionary type described in *PEP 372: Adding an Ordered Dictionary to collections*.

- The new `","` format specifier described in *PEP 378: Format Specifier for Thousands Separator*.
- The `memoryview` object.
- A small subset of the `importlib` module, described below.
- The `repr()` of a float `x` is shorter in many cases: it's now based on the shortest decimal string that's guaranteed to round back to `x`. As in previous versions of Python, it's guaranteed that `float(repr(x))` recovers `x`.
- Float-to-string and string-to-float conversions are correctly rounded. The `round()` function is also now correctly rounded.
- The `PyCapsule` type, used to provide a C API for extension modules.
- The `PyLong_AsLongAndOverflow()` C API function.

Other new Python3-mode warnings include:

- `operator.isCallable()` and `operator.sequenceIncludes()`, which are not supported in 3.x, now trigger warnings.
- The *-3* switch now automatically enables the *-Qwarn* switch that causes warnings about using classic division with integers and long integers.

4 PEP 372: Adding an Ordered Dictionary to collections

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, 2.7 introduces a new `OrderedDict` class in the `collections` module.

The `OrderedDict` API provides the same interface as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted:

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('first', 1),
...                  ('second', 2),
...                  ('third', 3)])
>>> d.items()
[('first', 1), ('second', 2), ('third', 3)]
```

If a new entry overwrites an existing entry, the original insertion position is left unchanged:

```
>>> d['second'] = 4
>>> d.items()
[('first', 1), ('second', 4), ('third', 3)]
```

Deleting an entry and reinserting it will move it to the end:

```
>>> del d['second']
>>> d['second'] = 5
>>> d.items()
[('first', 1), ('third', 3), ('second', 5)]
```

The `popitem()` method has an optional *last* argument that defaults to True. If *last* is True, the most recently added key is returned and removed; if it's False, the oldest key is selected:

```
>>> od = OrderedDict([(x,0) for x in range(20)])
>>> od.popitem()
(19, 0)
>>> od.popitem()
(18, 0)
```

```
>>> od.popitem(last=False)
(0, 0)
>>> od.popitem(last=False)
(1, 0)
```

Comparing two ordered dictionaries checks both the keys and values, and requires that the insertion order was the same:

```
>>> od1 = OrderedDict([('first', 1),
...                    ('second', 2),
...                    ('third', 3)])
>>> od2 = OrderedDict([('third', 3),
...                    ('first', 1),
...                    ('second', 2)])
>>> od1 == od2
False
>>> # Move 'third' key to the end
>>> del od2['third']; od2['third'] = 3
>>> od1 == od2
True
```

Comparing an `OrderedDict` with a regular dictionary ignores the insertion order and just compares the keys and values.

How does the `OrderedDict` work? It maintains a doubly-linked list of keys, appending new keys to the list as they're inserted. A secondary dictionary maps keys to their corresponding list node, so deletion doesn't have to traverse the entire linked list and therefore remains O(1).

The standard library now supports use of ordered dictionaries in several modules.

- The `ConfigParser` module uses them by default, meaning that configuration files can now be read, modified, and then written back in their original order.
- The `_asdict()` method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indices.
- The `json` module's `JSONDecoder` class constructor was extended with an *object_pairs_hook* parameter to allow `OrderedDict` instances to be built by the decoder. Support was also added for third-party tools like PyYAML.

See also:

PEP 372 - Adding an ordered dictionary to collections PEP written by Armin Ronacher and Raymond Hettinger; implemented by Raymond Hettinger.

5 PEP 378: Format Specifier for Thousands Separator

To make program output more readable, it can be useful to add separators to large numbers, rendering them as 18,446,744,073,709,551,616 instead of 18446744073709551616.

The fully general solution for doing this is the `locale` module, which can use different separators (“,” in North America, “.” in Europe) and different grouping sizes, but `locale` is complicated to use and unsuitable for multi-threaded applications where different threads are producing output for different locales.

Therefore, a simple comma-grouping mechanism has been added to the mini-language used by the `str.format()` method. When formatting a floating-point number, simply include a comma between the width and the precision:

```
>>> '{:20,.2f}'.format(18446744073709551616.0)
'18,446,744,073,709,551,616.00'
```

When formatting an integer, include the comma after the width:

```
>>> '{:20,d}'.format(18446744073709551616)
'18,446,744,073,709,551,616'
```

This mechanism is not adaptable at all; commas are always used as the separator and the grouping is always into three-digit groups. The comma-formatting mechanism isn't as general as the `locale` module, but it's easier to use.

See also:

PEP 378 - Format Specifier for Thousands Separator PEP written by Raymond Hettinger; implemented by Eric Smith.

6 PEP 389: The argparse Module for Parsing Command Lines

The `argparse` module for parsing command-line arguments was added as a more powerful replacement for the `optparse` module.

This means Python now supports three different modules for parsing command-line arguments: `getopt`, `optparse`, and `argparse`. The `getopt` module closely resembles the C library's `getopt()` function, so it remains useful if you're writing a Python prototype that will eventually be rewritten in C. `optparse` becomes redundant, but there are no plans to remove it because there are many scripts still using it, and there's no automated way to update these scripts. (Making the `argparse` API consistent with `optparse`'s interface was discussed but rejected as too messy and difficult.)

In short, if you're writing a new script and don't need to worry about compatibility with earlier versions of Python, use `argparse` instead of `optparse`.

Here's an example:

```
import argparse

parser = argparse.ArgumentParser(description='Command-line example.')

# Add optional switches
parser.add_argument('-v', action='store_true', dest='is_verbose',
                    help='produce verbose output')
parser.add_argument('-o', action='store', dest='output',
                    metavar='FILE',
                    help='direct output to FILE instead of stdout')
parser.add_argument('-C', action='store', type=int, dest='context',
                    metavar='NUM', default=0,
                    help='display NUM lines of added context')

# Allow any number of additional arguments.
parser.add_argument(nargs='*', action='store', dest='inputs',
                    help='input filenames (default is stdin)')

args = parser.parse_args()
print args.__dict__
```

Unless you override it, *-h* and *--help* switches are automatically added, and produce neatly formatted output:

```
-> ./python.exe argparse-example.py --help
usage: argparse-example.py [-h] [-v] [-o FILE] [-C NUM] [inputs [inputs ...]]

Command-line example.
```

```
positional arguments:
  inputs      input filenames (default is stdin)

optional arguments:
  -h, --help  show this help message and exit
  -v          produce verbose output
  -o FILE     direct output to FILE instead of stdout
  -C NUM      display NUM lines of added context
```

As with `optparse`, the command-line switches and arguments are returned as an object with attributes named by the *dest* parameters:

```
-> ./python.exe argparse-example.py -v
{'output': None,
 'is_verbose': True,
 'context': 0,
 'inputs': []}

-> ./python.exe argparse-example.py -v -o /tmp/output -C 4 file1 file2
{'output': '/tmp/output',
 'is_verbose': True,
 'context': 4,
 'inputs': ['file1', 'file2']}
```

`argparse` has much fancier validation than `optparse`; you can specify an exact number of arguments as an integer, 0 or more arguments by passing `'*'`, 1 or more by passing `'+'`, or an optional argument with `'?'`. A top-level parser can contain sub-parsers to define subcommands that have different sets of switches, as in `svn commit`, `svn checkout`, etc. You can specify an argument's type as `FileType`, which will automatically open files for you and understands that `'-'` means standard input or output.

See also:

`argparse` documentation The documentation page of the argparse module.

argparse-from-optparse Part of the Python documentation, describing how to convert code that uses `optparse`.

PEP 389 - argparse - New Command Line Parsing Module PEP written and implemented by Steven Bethard.

7 PEP 391: Dictionary-Based Configuration For Logging

The `logging` module is very flexible; applications can define a tree of logging subsystems, and each logger in this tree can filter out certain messages, format them differently, and direct messages to a varying number of handlers.

All this flexibility can require a lot of configuration. You can write Python statements to create objects and set their properties, but a complex set-up requires verbose but boring code. `logging` also supports a `fileConfig()` function that parses a file, but the file format doesn't support configuring filters, and it's messier to generate programmatically.

Python 2.7 adds a `dictConfig()` function that uses a dictionary to configure logging. There are many ways to produce a dictionary from different sources: construct one with code; parse a file containing JSON; or use a YAML parsing library if one is installed. For more information see *logging-config-api*.

The following example configures two loggers, the root logger and a logger named "network". Messages sent to the root logger will be sent to the system log using the syslog protocol, and messages to the "network" logger will be written to a `network.log` file that will be rotated once the log reaches 1MB.

```
import logging
import logging.config

configdict = {
 'version': 1,    # Configuration schema in use; must be 1 for now
 'formatters': {
     'standard': {
         'format': ('%(asctime)s %(name)-15s '
                    '%(levelname)-8s %(message)s')}},

 'handlers': {'netlog': {'backupCount': 10,
                      'class': 'logging.handlers.RotatingFileHandler',
                      'filename': '/logs/network.log',
                      'formatter': 'standard',
                      'level': 'INFO',
                      'maxBytes': 1000000},
              'syslog': {'class': 'logging.handlers.SysLogHandler',
                         'formatter': 'standard',
                         'level': 'ERROR'}},

 # Specify all the subordinate loggers
 'loggers': {
             'network': {
                         'handlers': ['netlog']
             }
 },
 # Specify properties of the root logger
 'root': {
          'handlers': ['syslog']
 },
}

# Set up configuration
logging.config.dictConfig(configdict)

# As an example, log two error messages
logger = logging.getLogger('/')
logger.error('Database not found')

netlogger = logging.getLogger('network')
netlogger.error('Connection failed')
```

Three smaller enhancements to the `logging` module, all implemented by Vinay Sajip, are:

- The `SysLogHandler` class now supports syslogging over TCP. The constructor has a *socktype* parameter giving the type of socket to use, either `socket.SOCK_DGRAM` for UDP or `socket.SOCK_STREAM` for TCP. The default protocol remains UDP.
- `Logger` instances gained a `getChild()` method that retrieves a descendant logger using a relative path. For example, once you retrieve a logger by doing `log = getLogger('app')`, calling `log.getChild('network.listen')` is equivalent to `getLogger('app.network.listen')`.
- The `LoggerAdapter` class gained a `isEnabledFor()` method that takes a *level* and returns whether the underlying logger would process a message of that level of importance.

See also:

PEP 391 - Dictionary-Based Configuration For Logging PEP written and implemented by Vinay Sajip.

8 PEP 3106: Dictionary Views

The dictionary methods `keys()`, `values()`, and `items()` are different in Python 3.x. They return an object called a *view* instead of a fully materialized list.

It's not possible to change the return values of `keys()`, `values()`, and `items()` in Python 2.7 because too much code would break. Instead the 3.x versions were added under the new names `viewkeys()`, `viewvalues()`, and `viewitems()`.

```
>>> d = dict((i*10, chr(65+i)) for i in range(26))
>>> d
{0: 'A', 130: 'N', 10: 'B', 140: 'O', 20: ..., 250: 'Z'}
>>> d.viewkeys()
dict_keys([0, 130, 10, 140, 20, 150, 30, ..., 250])
```

Views can be iterated over, but the key and item views also behave like sets. The `&` operator performs intersection, and `|` performs a union:

```
>>> d1 = dict((i*10, chr(65+i)) for i in range(26))
>>> d2 = dict((i**.5, i) for i in range(1000))
>>> d1.viewkeys() & d2.viewkeys()
set([0.0, 10.0, 20.0, 30.0])
>>> d1.viewkeys() | range(0, 30)
set([0, 1, 130, 3, 4, 5, 6, ..., 120, 250])
```

The view keeps track of the dictionary and its contents change as the dictionary is modified:

```
>>> vk = d.viewkeys()
>>> vk
dict_keys([0, 130, 10, ..., 250])
>>> d[260] = '&'
>>> vk
dict_keys([0, 130, 260, 10, ..., 250])
```

However, note that you can't add or remove keys while you're iterating over the view:

```
>>> for k in vk:
...     d[k*2] = k
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

You can use the view methods in Python 2.x code, and the 2to3 converter will change them to the standard `keys()`, `values()`, and `items()` methods.

See also:

PEP 3106 - Revamping dict.keys(), .values() and .items() PEP written by Guido van Rossum. Backported to 2.7 by Alexandre Vassalotti; issue 1967.

9 PEP 3137: The memoryview Object

The `memoryview` object provides a view of another object's memory content that matches the `bytes` type's interface.

```
>>> import string
>>> m = memoryview(string.letters)
>>> m
<memory at 0x37f850>
>>> len(m)           # Returns length of underlying object
52
>>> m[0], m[25], m[26]    # Indexing returns one byte
('a', 'z', 'A')
>>> m2 = m[0:26]          # Slicing returns another memoryview
>>> m2
<memory at 0x37f080>
```

The content of the view can be converted to a string of bytes or a list of integers:

```
>>> m2.tobytes()
'abcdefghijklmnopqrstuvwxyz'
>>> m2.tolist()
[97, 98, 99, 100, 101, 102, 103, ... 121, 122]
>>>
```

`memoryview` objects allow modifying the underlying object if it's a mutable object.

```
>>> m2[0] = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> b = bytearray(string.letters)  # Creating a mutable object
>>> b
bytearray(b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
>>> mb = memoryview(b)
>>> mb[0] = '*'          # Assign to view, changing the bytearray.
>>> b[0:5]               # The bytearray has been changed.
bytearray(b'*bcde')
>>>
```

See also:

PEP 3137 - Immutable Bytes and Mutable Buffer PEP written by Guido van Rossum. Implemented by Travis Oliphant, Antoine Pitrou and others. Backported to 2.7 by Antoine Pitrou; issue 2396.

10 Other Language Changes

Some smaller changes made to the core Python language are:

- The syntax for set literals has been backported from Python 3.x. Curly brackets are used to surround the contents of the resulting mutable set; set literals are distinguished from dictionaries by not containing colons and values. `{}` continues to represent an empty dictionary; use `set()` for an empty set.

  ```
  >>> {1, 2, 3, 4, 5}
  set([1, 2, 3, 4, 5])
  >>> set() # empty set
  ```

```
set([])
>>> {}     # empty dict
{}
```

Backported by Alexandre Vassalotti; issue 2335.

- Dictionary and set comprehensions are another feature backported from 3.x, generalizing list/generator comprehensions to use the literal syntax for sets and dictionaries.

```
>>> {x: x*x for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> {('a'*x) for x in range(6)}
set(['', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa'])
```

Backported by Alexandre Vassalotti; issue 2333.

- The `with` statement can now use multiple context managers in one statement. Context managers are processed from left to right and each one is treated as beginning a new `with` statement. This means that:

```
with A() as a, B() as b:
    ... suite of statements ...
```

is equivalent to:

```
with A() as a:
    with B() as b:
        ... suite of statements ...
```

The `contextlib.nested()` function provides a very similar function, so it's no longer necessary and has been deprecated.

(Proposed in https://codereview.appspot.com/53094; implemented by Georg Brandl.)

- Conversions between floating-point numbers and strings are now correctly rounded on most platforms. These conversions occur in many different places: `str()` on floats and complex numbers; the `float` and `complex` constructors; numeric formatting; serializing and deserializing floats and complex numbers using the `marshal`, `pickle` and `json` modules; parsing of float and imaginary literals in Python code; and `Decimal`-to-float conversion.

  Related to this, the `repr()` of a floating-point number *x* now returns a result based on the shortest decimal string that's guaranteed to round back to *x* under correct rounding (with round-half-to-even rounding mode). Previously it gave a string based on rounding x to 17 decimal digits.

  The rounding library responsible for this improvement works on Windows and on Unix platforms using the gcc, icc, or suncc compilers. There may be a small number of platforms where correct operation of this code cannot be guaranteed, so the code is not used on such systems. You can find out which code is being used by checking `sys.float_repr_style`, which will be `short` if the new code is in use and `legacy` if it isn't.

  Implemented by Eric Smith and Mark Dickinson, using David Gay's `dtoa.c` library; issue 7117.

- Conversions from long integers and regular integers to floating point now round differently, returning the floating-point number closest to the number. This doesn't matter for small integers that can be converted exactly, but for large numbers that will unavoidably lose precision, Python 2.7 now approximates more closely. For example, Python 2.6 computed the following:

```
>>> n = 29514790517935283391
>>> float(n)
2.9514790517935283e+20
>>> n - long(float(n))
65535L
```

Python 2.7's floating-point result is larger, but much closer to the true value:

```
>>> n = 29514790517935289l391
>>> float(n)
2.9514790517935289e+20
>>> n - long(float(n))
-1L
```

(Implemented by Mark Dickinson; issue 3166.)

Integer division is also more accurate in its rounding behaviours. (Also implemented by Mark Dickinson; issue 1811.)

- Implicit coercion for complex numbers has been removed; the interpreter will no longer ever attempt to call a `__coerce__()` method on complex objects. (Removed by Meador Inge and Mark Dickinson; issue 5211.)
- The `str.format()` method now supports automatic numbering of the replacement fields. This makes using `str.format()` more closely resemble using `%s` formatting:

```
>>> '{}:{}:{}'.format(2009, 04, 'Sunday')
'2009:4:Sunday'
>>> '{}:{}:{day}'.format(2009, 4, day='Sunday')
'2009:4:Sunday'
```

The auto-numbering takes the fields from left to right, so the first `{...}` specifier will use the first argument to `str.format()`, the next specifier will use the next argument, and so on. You can't mix auto-numbering and explicit numbering – either number all of your specifier fields or none of them – but you can mix auto-numbering and named fields, as in the second example above. (Contributed by Eric Smith; issue 5237.)

Complex numbers now correctly support usage with `format()`, and default to being right-aligned. Specifying a precision or comma-separation applies to both the real and imaginary parts of the number, but a specified field width and alignment is applied to the whole of the resulting `1.5+3j` output. (Contributed by Eric Smith; issue 1588 and issue 7988.)

The 'F' format code now always formats its output using uppercase characters, so it will now produce 'INF' and 'NAN'. (Contributed by Eric Smith; issue 3382.)

A low-level change: the `object.__format__()` method now triggers a `PendingDeprecationWarning` if it's passed a format string, because the `__format__()` method for `object` converts the object to a string representation and formats that. Previously the method silently applied the format string to the string representation, but that could hide mistakes in Python code. If you're supplying formatting information such as an alignment or precision, presumably you're expecting the formatting to be applied in some object-specific way. (Fixed by Eric Smith; issue 7994.)

- The `int()` and `long()` types gained a `bit_length` method that returns the number of bits necessary to represent its argument in binary:

```
>>> n = 37
>>> bin(n)
'0b100101'
>>> n.bit_length()
6
>>> n = 2**123-1
>>> n.bit_length()
123
>>> (n+1).bit_length()
124
```

(Contributed by Fredrik Johansson and Victor Stinner; issue 3439.)

- The `import` statement will no longer try an absolute import if a relative import (e.g. `from .os import sep`) fails. This fixes a bug, but could possibly break certain `import` statements that were only working by

accident. (Fixed by Meador Inge; issue 7902.)

- It's now possible for a subclass of the built-in `unicode` type to override the `__unicode__()` method. (Implemented by Victor Stinner; issue 1583863.)
- The `bytearray` type's `translate()` method now accepts `None` as its first argument. (Fixed by Georg Brandl; issue 4759.)
- When using `@classmethod` and `@staticmethod` to wrap methods as class or static methods, the wrapper object now exposes the wrapped function as their `__func__` attribute. (Contributed by Amaury Forgeot d'Arc, after a suggestion by George Sakkis; issue 5982.)
- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise `AttributeError` as you would expect. Fixed by Benjamin Peterson; issue 7604.)
- Two new encodings are now supported: "cp720", used primarily for Arabic text; and "cp858", a variant of CP 850 that adds the euro symbol. (CP720 contributed by Alexander Belchenko and Amaury Forgeot d'Arc in issue 1616979; CP858 contributed by Tim Hatch in issue 8016.)
- The `file` object will now set the `filename` attribute on the `IOError` exception when trying to open a directory on POSIX platforms (noted by Jan Kaliszewski; issue 4764), and now explicitly checks for and forbids writing to read-only file objects instead of trusting the C library to catch and report the error (fixed by Stefan Krah; issue 5677).
- The Python tokenizer now translates line endings itself, so the `compile()` built-in function now accepts code using any line-ending convention. Additionally, it no longer requires that the code end in a newline.
- Extra parentheses in function definitions are illegal in Python 3.x, meaning that you get a syntax error from `def f((x)): pass`. In Python3-warning mode, Python 2.7 will now warn about this odd usage. (Noted by James Lingard; issue 7362.)
- It's now possible to create weak references to old-style class objects. New-style classes were always weak-referenceable. (Fixed by Antoine Pitrou; issue 8268.)
- When a module object is garbage-collected, the module's dictionary is now only cleared if no one else is holding a reference to the dictionary (issue 7140).

10.1 Interpreter Changes

A new environment variable, `PYTHONWARNINGS`, allows controlling warnings. It should be set to a string containing warning settings, equivalent to those used with the `-W` switch, separated by commas. (Contributed by Brian Curtin; issue 7301.)

For example, the following setting will print warnings every time they occur, but turn warnings from the `Cookie` module into an error. (The exact syntax for setting an environment variable varies across operating systems and shells.)

```
export PYTHONWARNINGS=all,error:::Cookie:0
```

10.2 Optimizations

Several performance enhancements have been added:

- A new opcode was added to perform the initial setup for `with` statements, looking up the `__enter__()` and `__exit__()` methods. (Contributed by Benjamin Peterson.)
- The garbage collector now performs better for one common usage pattern: when many objects are being allocated without deallocating any of them. This would previously take quadratic time for garbage collection, but now the number of full garbage collections is reduced as the number of objects on the heap grows. The new

logic only performs a full garbage collection pass when the middle generation has been collected 10 times and when the number of survivor objects from the middle generation exceeds 10% of the number of objects in the oldest generation. (Suggested by Martin von Löwis and implemented by Antoine Pitrou; issue 4074.)

- The garbage collector tries to avoid tracking simple containers which can't be part of a cycle. In Python 2.7, this is now true for tuples and dicts containing atomic types (such as ints, strings, etc.). Transitively, a dict containing tuples of atomic types won't be tracked either. This helps reduce the cost of each garbage collection by decreasing the number of objects to be considered and traversed by the collector. (Contributed by Antoine Pitrou; issue 4688.)
- Long integers are now stored internally either in base 2**15 or in base 2**30, the base being determined at build time. Previously, they were always stored in base 2**15. Using base 2**30 gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base 2**30 on 64-bit machines and base 2**15 on 32-bit machines; on Unix, there's a new configure option `--enable-big-digits` that can be used to override this default.

  Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there's a new structseq `sys.long_info` that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

  ```
  >>> import sys
  >>> sys.long_info
  sys.long_info(bits_per_digit=30, sizeof_digit=4)
  ```

  (Contributed by Mark Dickinson; issue 4258.)

  Another set of changes made long objects a few bytes smaller: 2 bytes smaller on 32-bit systems and 6 bytes on 64-bit. (Contributed by Mark Dickinson; issue 5260.)
- The division algorithm for long integers has been made faster by tightening the inner loop, doing shifts instead of multiplications, and fixing an unnecessary extra iteration. Various benchmarks show speedups of between 50% and 150% for long integer divisions and modulo operations. (Contributed by Mark Dickinson; issue 5512.) Bitwise operations are also significantly faster (initial patch by Gregory Smith; issue 1087418).
- The implementation of `%` checks for the left-side operand being a Python string and special-cases it; this results in a 1-3% performance increase for applications that frequently use `%` with strings, such as templating libraries. (Implemented by Collin Winter; issue 5176.)
- List comprehensions with an `if` condition are compiled into faster bytecode. (Patch by Antoine Pitrou, backported to 2.7 by Jeffrey Yasskin; issue 4715.)
- Converting an integer or long integer to a decimal string was made faster by special-casing base 10 instead of using a generalized conversion function that supports arbitrary bases. (Patch by Gawain Bolton; issue 6713.)
- The `split()`, `replace()`, `rindex()`, `rpartition()`, and `rsplit()` methods of string-like types (strings, Unicode strings, and `bytearray` objects) now use a fast reverse-search algorithm instead of a character-by-character scan. This is sometimes faster by a factor of 10. (Added by Florent Xicluna; issue 7462 and issue 7622.)
- The `pickle` and `cPickle` modules now automatically intern the strings used for attribute names, reducing memory usage of the objects resulting from unpickling. (Contributed by Jake McGuire; issue 5084.)
- The `cPickle` module now special-cases dictionaries, nearly halving the time required to pickle them. (Contributed by Collin Winter; issue 5670.)

11 New and Improved Modules

As in every release, 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 Subversion logs for all the details.

- The `bdb` module's base debugging class `Bdb` gained a feature for skipping modules. The constructor now takes an iterable containing glob-style patterns such as `django.*`; the debugger will not step into stack frames from a module that matches one of these patterns. (Contributed by Maru Newby after a suggestion by Senthil Kumaran; issue 5142.)

- The `binascii` module now supports the buffer API, so it can be used with `memoryview` instances and other similar buffer objects. (Backported from 3.x by Florent Xicluna; issue 7703.)

- Updated module: the `bsddb` module has been updated from 4.7.2devel9 to version 4.8.4 of the pybsddb package. The new version features better Python 3.x compatibility, various bug fixes, and adds several new BerkeleyDB flags and methods. (Updated by Jesús Cea Avión; issue 8156. The pybsddb changelog can be read at http://hg.jcea.es/pybsddb/file/tip/ChangeLog.)

- The `bz2` module's `BZ2File` now supports the context management protocol, so you can write `with bz2.BZ2File(...) as f:`. (Contributed by Hagen Fürstenau; issue 3860.)

- New class: the `Counter` class in the `collections` module is useful for tallying data. `Counter` instances behave mostly like dictionaries but return zero for missing keys instead of raising a `KeyError`:

```
>>> from collections import Counter
>>> c = Counter()
>>> for letter in 'here is a sample of english text':
...   c[letter] += 1
...
>>> c
Counter({' ': 6, 'e': 5, 's': 3, 'a': 2, 'i': 2, 'h': 2,
'l': 2, 't': 2, 'g': 1, 'f': 1, 'm': 1, 'o': 1, 'n': 1,
'p': 1, 'r': 1, 'x': 1})
>>> c['e']
5
>>> c['z']
0
```

  There are three additional `Counter` methods. `most_common()` returns the N most common elements and their counts. `elements()` returns an iterator over the contained elements, repeating each element as many times as its count. `subtract()` takes an iterable and subtracts one for each element instead of adding; if the argument is a dictionary or another `Counter`, the counts are subtracted.

```
>>> c.most_common(5)
[(' ', 6), ('e', 5), ('s', 3), ('a', 2), ('i', 2)]
>>> c.elements() ->
   'a', 'a', ' ', ' ', ' ', ' ', ' ', ' ',
   'e', 'e', 'e', 'e', 'e', 'g', 'f', 'i', 'i',
   'h', 'h', 'm', 'l', 'l', 'o', 'n', 'p', 's',
   's', 's', 'r', 't', 't', 'x'
>>> c['e']
5
>>> c.subtract('very heavy on the letter e')
>>> c['e']     # Count is now lower
-1
```

  Contributed by Raymond Hettinger; issue 1696199.

  New class: `OrderedDict` is described in the earlier section *PEP 372: Adding an Ordered Dictionary to collections*.

  New method: The `deque` data type now has a `count()` method that returns the number of contained elements

equal to the supplied argument *x*, and a `reverse()` method that reverses the elements of the deque in-place. `deque` also exposes its maximum length as the read-only `maxlen` attribute. (Both features added by Raymond Hettinger.)

The `namedtuple` class now has an optional *rename* parameter. If *rename* is true, field names that are invalid because they've been repeated or aren't legal Python identifiers will be renamed to legal names that are derived from the field's position within the list of fields:

```
>>> from collections import namedtuple
>>> T = namedtuple('T', ['field1', '$illegal', 'for', 'field2'], rename=True)
>>> T._fields
('field1', '_1', '_2', 'field2')
```

(Added by Raymond Hettinger; issue 1818.)

Finally, the `Mapping` abstract base class now returns `NotImplemented` if a mapping is compared to another type that isn't a `Mapping`. (Fixed by Daniel Stutzbach; issue 8729.)

- Constructors for the parsing classes in the `ConfigParser` module now take an *allow_no_value* parameter, defaulting to false; if true, options without values will be allowed. For example:

```
>>> import ConfigParser, StringIO
>>> sample_config = """
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-bdb
... """
>>> config = ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(StringIO.StringIO(sample_config))
>>> config.get('mysqld', 'user')
'mysql'
>>> print config.get('mysqld', 'skip-bdb')
None
>>> print config.get('mysqld', 'unknown')
Traceback (most recent call last):
  ...
NoOptionError: No option 'unknown' in section: 'mysqld'
```

(Contributed by Mats Kindahl; issue 7005.)

- Deprecated function: `contextlib.nested()`, which allows handling more than one context manager with a single `with` statement, has been deprecated, because the `with` statement now supports multiple context managers.
- The `cookielib` module now ignores cookies that have an invalid version field, one that doesn't contain an integer value. (Fixed by John J. Lee; issue 3924.)
- The `copy` module's `deepcopy()` function will now correctly copy bound instance methods. (Implemented by Robert Collins; issue 1515.)
- The `ctypes` module now always converts `None` to a C NULL pointer for arguments declared as pointers. (Changed by Thomas Heller; issue 4606.) The underlying libffi library has been updated to version 3.0.9, containing various fixes for different platforms. (Updated by Matthias Klose; issue 8142.)
- New method: the `datetime` module's `timedelta` class gained a `total_seconds()` method that returns the number of seconds in the duration. (Contributed by Brian Quinlan; issue 5788.)
- New method: the `Decimal` class gained a `from_float()` class method that performs an exact conversion of a floating-point number to a `Decimal`. This exact conversion strives for the clos-

est decimal approximation to the floating-point representation's value; the resulting decimal value will therefore still include the inaccuracy, if any. For example, `Decimal.from_float(0.1)` returns `Decimal('0.1000000000000000055511151231257827021181583404541015625')`. (Implemented by Raymond Hettinger; issue 4796.)

Comparing instances of `Decimal` with floating-point numbers now produces sensible results based on the numeric values of the operands. Previously such comparisons would fall back to Python's default rules for comparing objects, which produced arbitrary results based on their type. Note that you still cannot combine `Decimal` and floating-point in other operations such as addition, since you should be explicitly choosing how to convert between float and `Decimal`. (Fixed by Mark Dickinson; issue 2531.)

The constructor for `Decimal` now accepts floating-point numbers (added by Raymond Hettinger; issue 8257) and non-European Unicode characters such as Arabic-Indic digits (contributed by Mark Dickinson; issue 6595).

Most of the methods of the `Context` class now accept integers as well as `Decimal` instances; the only exceptions are the `canonical()` and `is_canonical()` methods. (Patch by Juan José Conti; issue 7633.)

When using `Decimal` instances with a string's `format()` method, the default alignment was previously left-alignment. This has been changed to right-alignment, which is more sensible for numeric types. (Changed by Mark Dickinson; issue 6857.)

Comparisons involving a signaling NaN value (or `sNAN`) now signal `InvalidOperation` instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or `NaN`) are now hashable. (Fixed by Mark Dickinson; issue 7279.)

- The `difflib` module now produces output that is more compatible with modern **diff/patch** tools through one small change, using a tab character instead of spaces as a separator in the header giving the filename. (Fixed by Anatoly Techtonik; issue 7585.)
- The Distutils `sdist` command now always regenerates the `MANIFEST` file, since even if the `MANIFEST.in` or `setup.py` files haven't been modified, the user might have created some new files that should be included. (Fixed by Tarek Ziadé; issue 8688.)
- The `doctest` module's `IGNORE_EXCEPTION_DETAIL` flag will now ignore the name of the module containing the exception being tested. (Patch by Lennart Regebro; issue 7490.)
- The `email` module's `Message` class will now accept a Unicode-valued payload, automatically converting the payload to the encoding specified by `output_charset`. (Added by R. David Murray; issue 1368247.)
- The `Fraction` class now accepts a single float or `Decimal` instance, or two rational numbers, as arguments to its constructor. (Implemented by Mark Dickinson; rationals added in issue 5812, and float/decimal in issue 8294.)

  Ordering comparisons (<, <=, >, >=) between fractions and complex numbers now raise a `TypeError`. This fixes an oversight, making the `Fraction` match the other numeric types.
- New class: `FTP_TLS` in the `ftplib` module provides secure FTP connections using TLS encapsulation of authentication as well as subsequent control and data transfers. (Contributed by Giampaolo Rodola; issue 2054.)

  The `storbinary()` method for binary uploads can now restart uploads thanks to an added *rest* parameter (patch by Pablo Mouzo; issue 6845.)
- New class decorator: `total_ordering()` in the `functools` module takes a class that defines an `__eq__()` method and one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`, and generates the missing comparison methods. Since the `__cmp__()` method is being deprecated in Python 3.x, this decorator makes it easier to define ordered classes. (Added by Raymond Hettinger; issue 5479.)

  New function: `cmp_to_key()` will take an old-style comparison function that expects two arguments and return a new callable that can be used as the *key* parameter to functions such as `sorted()`, `min()` and `max()`, etc. The primary intended use is to help with making code compatible with Python 3.x. (Added by Raymond Hettinger.)

- New function: the `gc` module's `is_tracked()` returns true if a given instance is tracked by the garbage collector, false otherwise. (Contributed by Antoine Pitrou; issue 4688.)

- The `gzip` module's `GzipFile` now supports the context management protocol, so you can write `with gzip.GzipFile(...) as f:` (contributed by Hagen Fürstenau; issue 3860), and it now implements the `io.BufferedIOBase` ABC, so you can wrap it with `io.BufferedReader` for faster processing (contributed by Nir Aides; issue 7471). It's also now possible to override the modification time recorded in a gzipped file by providing an optional timestamp to the constructor. (Contributed by Jacques Frechet; issue 4272.)

  Files in gzip format can be padded with trailing zero bytes; the `gzip` module will now consume these trailing bytes. (Fixed by Tadek Pietraszek and Brian Curtin; issue 2846.)

- New attribute: the `hashlib` module now has an `algorithms` attribute containing a tuple naming the supported algorithms. In Python 2.7, `hashlib.algorithms` contains `('md5', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512')`. (Contributed by Carl Chenet; issue 7418.)

- The default `HTTPResponse` class used by the `httplib` module now supports buffering, resulting in much faster reading of HTTP responses. (Contributed by Kristján Valur Jónsson; issue 4879.)

  The `HTTPConnection` and `HTTPSConnection` classes now support a *source_address* parameter, a `(host, port)` 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; issue 3972.)

- The `ihooks` module now supports relative imports. Note that `ihooks` is an older module for customizing imports, superseded by the `imputil` module added in Python 2.0. (Relative import support added by Neil Schemenauer.)

- The `imaplib` module now supports IPv6 addresses. (Contributed by Derek Morr; issue 1655.)

- New function: the `inspect` module's `getcallargs()` takes a callable and its positional and keyword arguments, and figures out which of the callable's parameters will receive each argument, returning a dictionary mapping argument names to their values. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'b': 2, 'pos': (3,), 'named': {}}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'b': 1, 'pos': (), 'named': {'x': 4}}
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() takes at least 1 argument (0 given)
```

  Contributed by George Sakkis; issue 3135.

- Updated module: The `io` library has been upgraded to the version shipped with Python 3.1. For 3.1, the I/O library was entirely rewritten in C and is 2 to 20 times faster depending on the task being performed. The original Python version was renamed to the `_pyio` module.

  One minor resulting change: the `io.TextIOBase` class now has an `errors` attribute giving the error setting used for encoding and decoding errors (one of `'strict'`, `'replace'`, `'ignore'`).

  The `io.FileIO` class now raises an `OSError` when passed an invalid file descriptor. (Implemented by Benjamin Peterson; issue 4991.) The `truncate()` method now preserves the file position; previously it would change the file position to the end of the new file. (Fixed by Pascal Chambon; issue 6939.)

- New function: `itertools.compress(data, selectors)` takes two iterators. Elements of *data* are returned if the corresponding value in *selectors* is true:

```
itertools.compress('ABCDEF', [1,0,1,0,1,1]) =>
  A, C, E, F
```

New function: `itertools.combinations_with_replacement(iter, r)` returns all the possible *r*-length combinations of elements from the iterable *iter*. Unlike `combinations()`, individual elements can be repeated in the generated combinations:

```
itertools.combinations_with_replacement('abc', 2) =>
  ('a', 'a'), ('a', 'b'), ('a', 'c'),
  ('b', 'b'), ('b', 'c'), ('c', 'c')
```

Note that elements are treated as unique depending on their position in the input, not their actual values.

The `itertools.count()` function now has a *step* argument that allows incrementing by values other than 1. `count()` also now allows keyword arguments, and using non-integer values such as floats or `Decimal` instances. (Implemented by Raymond Hettinger; issue 5032.)

`itertools.combinations()` and `itertools.product()` previously raised `ValueError` for values of *r* larger than the input iterable. This was deemed a specification error, so they now return an empty iterator. (Fixed by Raymond Hettinger; issue 4816.)

- Updated module: The `json` module was upgraded to version 2.0.9 of the simplejson package, which includes a C extension that makes encoding and decoding faster. (Contributed by Bob Ippolito; issue 4136.)

  To support the new `collections.OrderedDict` type, `json.load()` now has an optional *object_pairs_hook* parameter that will be called with any object literal that decodes to a list of pairs. (Contributed by Raymond Hettinger; issue 5381.)

- The `mailbox` module's `Maildir` class now records the timestamp on the directories it reads, and only re-reads them if the modification time has subsequently changed. This improves performance by avoiding unneeded directory scans. (Fixed by A.M. Kuchling and Antoine Pitrou; issue 1607951, issue 6896.)

- New functions: the `math` module gained `erf()` and `erfc()` for the error function and the complementary error function, `expm1()` which computes `e**x - 1` with more precision than using `exp()` and subtracting 1, `gamma()` for the Gamma function, and `lgamma()` for the natural log of the Gamma function. (Contributed by Mark Dickinson and nirinA raseliarison; issue 3366.)

- The `multiprocessing` module's `Manager*` classes can now be passed a callable that will be called whenever a subprocess is started, along with a set of arguments that will be passed to the callable. (Contributed by lekma; issue 5585.)

  The `Pool` class, which controls a pool of worker processes, now has an optional *maxtasksperchild* parameter. Worker processes will perform the specified number of tasks and then exit, causing the `Pool` to start a new worker. This is useful if tasks may leak memory or other resources, or if some tasks will cause the worker to become very large. (Contributed by Charles Cazabon; issue 6963.)

- The `nntplib` module now supports IPv6 addresses. (Contributed by Derek Morr; issue 1664.)

- New functions: the `os` module wraps the following POSIX system calls: `getresgid()` and `getresuid()`, which return the real, effective, and saved GIDs and UIDs; `setresgid()` and `setresuid()`, which set real, effective, and saved GIDs and UIDs to new values; `initgroups()`, which initialize the group access list for the current process. (GID/UID functions contributed by Travis H.; issue 6508. Support for initgroups added by Jean-Paul Calderone; issue 7333.)

  The `os.fork()` function now re-initializes the import lock in the child process; this fixes problems on Solaris when `fork()` is called from a thread. (Fixed by Zsolt Cserna; issue 7242.)

- In the `os.path` module, the `normpath()` and `abspath()` functions now preserve Unicode; if their input path is a Unicode string, the return value is also a Unicode string. (`normpath()` fixed by Matt Giuca in issue 5827; `abspath()` fixed by Ezio Melotti in issue 3426.)

- The pydoc module now has help for the various symbols that Python uses. You can now do help('<<') or help('@'), for example. (Contributed by David Laban; issue 4739.)

- The re module's split(), sub(), and subn() now accept an optional *flags* argument, for consistency with the other functions in the module. (Added by Gregory P. Smith.)

- New function: run_path() in the runpy module will execute the code at a provided *path* argument. *path* can be the path of a Python source file (example.py), a compiled bytecode file (example.pyc), a directory (./package/), or a zip archive (example.zip). If a directory or zip path is provided, it will be added to the front of sys.path and the module __main__ will be imported. It's expected that the directory or zip contains a __main__.py; if it doesn't, some other __main__.py might be imported from a location later in sys.path. This makes more of the machinery of runpy available to scripts that want to mimic the way Python's command line processes an explicit path name. (Added by Nick Coghlan; issue 6816.)

- New function: in the shutil module, make_archive() takes a filename, archive type (zip or tar-format), and a directory path, and creates an archive containing the directory's contents. (Added by Tarek Ziadé.)

  shutil's copyfile() and copytree() functions now raise a SpecialFileError exception when asked to copy a named pipe. Previously the code would treat named pipes like a regular file by opening them for reading, and this would block indefinitely. (Fixed by Antoine Pitrou; issue 3002.)

- The signal module no longer re-installs the signal handler unless this is truly necessary, which fixes a bug that could make it impossible to catch the EINTR signal robustly. (Fixed by Charles-Francois Natali; issue 8354.)

- New functions: in the site module, three new functions return various site- and user-specific paths. getsitepackages() returns a list containing all global site-packages directories, getusersitepackages() returns the path of the user's site-packages directory, and getuserbase() returns the value of the USER_BASE environment variable, giving the path to a directory that can be used to store data. (Contributed by Tarek Ziadé; issue 6693.)

  The site module now reports exceptions occurring when the sitecustomize module is imported, and will no longer catch and swallow the KeyboardInterrupt exception. (Fixed by Victor Stinner; issue 3137.)

- The create_connection() function gained a *source_address* parameter, a (host, port) 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; issue 3972.)

  The recv_into() and recvfrom_into() methods will now write into objects that support the buffer API, most usefully the bytearray and memoryview objects. (Implemented by Antoine Pitrou; issue 8104.)

- The SocketServer module's TCPServer class now supports socket timeouts and disabling the Nagle algorithm. The disable_nagle_algorithm class attribute defaults to False; if overridden to be True, new request connections will have the TCP_NODELAY option set to prevent buffering many small sends into a single TCP packet. The timeout class attribute can hold a timeout in seconds that will be applied to the request socket; if no request is received within that time, handle_timeout() will be called and handle_request() will return. (Contributed by Kristján Valur Jónsson; issue 6192 and issue 6267.)

- Updated module: the sqlite3 module has been updated to version 2.6.0 of the pysqlite package. Version 2.6.0 includes a number of bugfixes, and adds the ability to load SQLite extensions from shared libraries. Call the enable_load_extension(True) method to enable extensions, and then call load_extension() to load a particular shared library. (Updated by Gerhard Häring.)

- The ssl module's SSLSocket objects now support the buffer API, which fixed a test suite failure (fix by Antoine Pitrou; issue 7133) and automatically set OpenSSL's SSL_MODE_AUTO_RETRY, which will prevent an error code being returned from recv() operations that trigger an SSL renegotiation (fix by Antoine Pitrou; issue 8222).

  The ssl.wrap_socket() constructor function now takes a *ciphers* argument that's a string listing the encryption algorithms to be allowed; the format of the string is described in the OpenSSL documentation. (Added by Antoine Pitrou; issue 8322.)

Another change makes the extension load all of OpenSSL's ciphers and digest algorithms so that they're all available. Some SSL certificates couldn't be verified, reporting an "unknown algorithm" error. (Reported by Beda Kosata, and fixed by Antoine Pitrou; issue 8484.)

The version of OpenSSL being used is now available as the module attributes `ssl.OPENSSL_VERSION` (a string), `ssl.OPENSSL_VERSION_INFO` (a 5-tuple), and `ssl.OPENSSL_VERSION_NUMBER` (an integer). (Added by Antoine Pitrou; issue 8321.)

- The `struct` module will no longer silently ignore overflow errors when a value is too large for a particular integer format code (one of `bBhHiIlLqQ`); it now always raises a `struct.error` exception. (Changed by Mark Dickinson; issue 1523.) The `pack()` function will also attempt to use `__index__()` to convert and pack non-integers before trying the `__int__()` method or reporting an error. (Changed by Mark Dickinson; issue 8300.)

- New function: the `subprocess` module's `check_output()` runs a command with a specified set of arguments and returns the command's output as a string when the command runs without error, or raises a `CalledProcessError` exception otherwise.

```
>>> subprocess.check_output(['df', '-h', '.'])
'Filesystem     Size   Used  Avail Capacity  Mounted on\n
/dev/disk0s2    52G    49G   3.0G    94%    /\n'

>>> subprocess.check_output(['df', '-h', '/bogus'])
  ...
subprocess.CalledProcessError: Command '['df', '-h', '/bogus']' returned non-zero exit
```

(Contributed by Gregory P. Smith.)

The `subprocess` module will now retry its internal system calls on receiving an `EINTR` signal. (Reported by several people; final patch by Gregory P. Smith in issue 1068268.)

- New function: `is_declared_global()` in the `symtable` module returns true for variables that are explicitly declared to be global, false for ones that are implicitly global. (Contributed by Jeremy Hylton.)

- The `syslog` module will now use the value of `sys.argv[0]` as the identifier instead of the previous default value of `'python'`. (Changed by Sean Reifschneider; issue 8451.)

- The `sys.version_info` value is now a named tuple, with attributes named `major`, `minor`, `micro`, `releaselevel`, and `serial`. (Contributed by Ross Light; issue 4285.)

  `sys.getwindowsversion()` also returns a named tuple, with attributes named `major`, `minor`, `build`, `platform`, `service_pack`, `service_pack_major`, `service_pack_minor`, `suite_mask`, and `product_type`. (Contributed by Brian Curtin; issue 7766.)

- The `tarfile` module's default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; issue 7357.)

  `tarfile` now supports filtering the `TarInfo` objects being added to a tar file. When you call `add()`, you may supply an optional *filter* argument that's a callable. The *filter* callable will be passed the `TarInfo` for every file being added, and can modify and return it. If the callable returns `None`, the file will be excluded from the resulting archive. This is more powerful than the existing *exclude* argument, which has therefore been deprecated. (Added by Lars Gustäbel; issue 6856.) The `TarFile` class also now supports the context management protocol. (Added by Lars Gustäbel; issue 7232.)

- The `wait()` method of the `threading.Event` class now returns the internal flag on exit. This means the method will usually return true because `wait()` is supposed to block until the internal flag becomes true. The return value will only be false if a timeout was provided and the operation timed out. (Contributed by Tim Lesher; issue 1674032.)

- The Unicode database provided by the `unicodedata` module is now used internally to determine which characters are numeric, whitespace, or represent line breaks. The database also includes information from the `Unihan.txt` data file (patch by Anders Chrigström and Amaury Forgeot d'Arc; issue 1571184) and has been updated to version 5.2.0 (updated by Florent Xicluna; issue 8024).

- The `urlparse` module's `urlsplit()` now handles unknown URL schemes in a fashion compliant with **RFC 3986**: if the URL is of the form `"<something>://..."`, the text before the `://` is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

  ```
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', '', '//host/filename?query', '', '')
  ```

  Python 2.7 (and Python 2.6.5) will return:

  ```
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', 'host', '/filename?query', '', '')
  ```

  (Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

  The `urlparse` module also supports IPv6 literal addresses as defined by **RFC 2732** (contributed by Senthil Kumaran; issue 2987).

  ```
  >>> urlparse.urlparse('http://[1080::8:800:200C:417A]/foo')
  ParseResult(scheme='http', netloc='[1080::8:800:200C:417A]',
              path='/foo', params='', query='', fragment='')
  ```

- New class: the `WeakSet` class in the `weakref` module is a set that only holds weak references to its elements; elements will be removed once there are no references pointing to them. (Originally implemented in Python 3.x by Raymond Hettinger, and backported to 2.7 by Michael Foord.)

- The ElementTree library, `xml.etree`, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!-- comment -->`). (Patch by Neil Muller; issue 2746.)

- The XML-RPC client and server, provided by the `xmlrpclib` and `SimpleXMLRPCServer` modules, have improved performance by supporting HTTP/1.1 keep-alive and by optionally using gzip encoding to compress the XML being exchanged. The gzip compression is controlled by the `encode_threshold` attribute of `SimpleXMLRPCRequestHandler`, which contains a size in bytes; responses larger than this will be compressed. (Contributed by Kristján Valur Jónsson; issue 6267.)

- The `zipfile` module's `ZipFile` now supports the context management protocol, so you can write `with zipfile.ZipFile(...) as f:`. (Contributed by Brian Curtin; issue 5511.)

  `zipfile` now also supports archiving empty directories and extracts them correctly. (Fixed by Kuba Wieczorek; issue 4710.) Reading files out of an archive is faster, and interleaving `read()` and `readline()` now works correctly. (Contributed by Nir Aides; issue 7610.)

  The `is_zipfile()` function now accepts a file object, in addition to the path names accepted in earlier versions. (Contributed by Gabriel Genellina; issue 4756.)

  The `writestr()` method now has an optional *compress_type* parameter that lets you override the default compression method specified in the `ZipFile` constructor. (Contributed by Ronald Oussoren; issue 6003.)

11.1 New module: importlib

Python 3.1 includes the `importlib` package, a re-implementation of the logic underlying Python's `import` statement. `importlib` is useful for implementors of Python interpreters and to users who wish to write new importers

that can participate in the import process. Python 2.7 doesn't contain the complete `importlib` package, but instead has a tiny subset that contains a single function, `import_module()`.

`import_module(name, package=None)` imports a module. *name* is a string containing the module or package's name. It's possible to do relative imports by providing a string that begins with a `.` character, such as `..utils.errors`. For relative imports, the *package* argument must be provided and is the name of the package that will be used as the anchor for the relative import. `import_module()` both inserts the imported module into `sys.modules` and returns the module object.

Here are some examples:

```
>>> from importlib import import_module
>>> anydbm = import_module('anydbm')  # Standard absolute import
>>> anydbm
<module 'anydbm' from '/p/python/Lib/anydbm.py'>
>>> # Relative import
>>> file_util = import_module('..file_util', 'distutils.command')
>>> file_util
<module 'distutils.file_util' from '/python/Lib/distutils/file_util.pyc'>
```

`importlib` was implemented by Brett Cannon and introduced in Python 3.1.

11.2 New module: sysconfig

The `sysconfig` module has been pulled out of the Distutils package, becoming a new top-level module in its own right. `sysconfig` provides functions for getting information about Python's build process: compiler switches, installation paths, the platform name, and whether Python is running from its source directory.

Some of the functions in the module are:

- `get_config_var()` returns variables from Python's Makefile and the `pyconfig.h` file.
- `get_config_vars()` returns a dictionary containing all of the configuration variables.
- `get_path()` returns the configured path for a particular type of module: the standard library, site-specific modules, platform-specific modules, etc.
- `is_python_build()` returns true if you're running a binary from a Python source tree, and false otherwise.

Consult the `sysconfig` documentation for more details and for a complete list of functions.

The Distutils package and `sysconfig` are now maintained by Tarek Ziadé, who has also started a Distutils2 package (source repository at https://hg.python.org/distutils2/) for developing a next-generation version of Distutils.

11.3 ttk: Themed Widgets for Tk

Tcl/Tk 8.5 includes a set of themed widgets that re-implement basic Tk widgets but have a more customizable appearance and can therefore more closely resemble the native platform's widgets. This widget set was originally called Tile, but was renamed to Ttk (for "themed Tk") on being added to Tcl/Tck release 8.5.

To learn more, read the `ttk` module documentation. You may also wish to read the Tcl/Tk manual page describing the Ttk theme engine, available at http://www.tcl.tk/man/tcl8.5/TkCmd/ttk_intro.htm. Some screenshots of the Python/Ttk code in use are at http://code.google.com/p/python-ttk/wiki/Screenshots.

The `ttk` module was written by Guilherme Polo and added in issue 2983. An alternate version called `Tile.py`, written by Martin Franklin and maintained by Kevin Walzer, was proposed for inclusion in issue 2618, but the authors argued that Guilherme Polo's work was more comprehensive.

11.4 Updated module: unittest

The `unittest` module was greatly enhanced; many new features were added. Most of these features were implemented by Michael Foord, unless otherwise noted. The enhanced version of the module is downloadable separately for use with Python versions 2.4 to 2.6, packaged as the `unittest2` package, from https://pypi.python.org/pypi/unittest2.

When used from the command line, the module can automatically discover tests. It's not as fancy as py.test or nose, but provides a simple way to run tests kept within a set of package directories. For example, the following command will search the `test/` subdirectory for any importable test files named `test*.py`:

```
python -m unittest discover -s test
```

Consult the `unittest` module documentation for more details. (Developed in issue 6001.)

The `main()` function supports some other new options:

- *`-b`* or *`--buffer`* will buffer the standard output and standard error streams during each test. If the test passes, any resulting output will be discarded; on failure, the buffered output will be displayed.
- *`-c`* or *`--catch`* will cause the control-C interrupt to be handled more gracefully. Instead of interrupting the test process immediately, the currently running test will be completed and then the partial results up to the interruption will be reported. If you're impatient, a second press of control-C will cause an immediate interruption.

  This control-C handler tries to avoid causing problems when the code being tested or the tests being run have defined a signal handler of their own, by noticing that a signal handler was already set and calling it. If this doesn't work for you, there's a `removeHandler()` decorator that can be used to mark tests that should have the control-C handling disabled.
- *`-f`* or *`--failfast`* makes test execution stop immediately when a test fails instead of continuing to execute further tests. (Suggested by Cliff Dyer and implemented by Michael Foord; issue 8074.)

The progress messages now show 'x' for expected failures and 'u' for unexpected successes when run in verbose mode. (Contributed by Benjamin Peterson.)

Test cases can raise the `SkipTest` exception to skip a test (issue 1034053).

The error messages for `assertEqual()`, `assertTrue()`, and `assertFalse()` failures now provide more information. If you set the `longMessage` attribute of your `TestCase` classes to True, both the standard error message and any additional message you provide will be printed for failures. (Added by Michael Foord; issue 5663.)

The `assertRaises()` method now returns a context handler when called without providing a callable object to run. For example, you can write this:

```
with self.assertRaises(KeyError):
    {}['foo']
```

(Implemented by Antoine Pitrou; issue 4444.)

Module- and class-level setup and teardown fixtures are now supported. Modules can contain `setUpModule()` and `tearDownModule()` functions. Classes can have `setUpClass()` and `tearDownClass()` methods that must be defined as class methods (using `@classmethod` or equivalent). These functions and methods are invoked when the test runner switches to a test case in a different module or class.

The methods `addCleanup()` and `doCleanups()` were added. `addCleanup()` lets you add cleanup functions that will be called unconditionally (after `setUp()` if `setUp()` fails, otherwise after `tearDown()`). This allows for much simpler resource allocation and deallocation during tests (issue 5679).

A number of new methods were added that provide more specialized tests. Many of these methods were written by Google engineers for use in their test suites; Gregory P. Smith, Michael Foord, and GvR worked on merging them into Python's version of `unittest`.

- `assertIsNone()` and `assertIsNotNone()` take one expression and verify that the result is or is not `None`.
- `assertIs()` and `assertIsNot()` take two values and check whether the two values evaluate to the same object or not. (Added by Michael Foord; issue 2578.)
- `assertIsInstance()` and `assertNotIsInstance()` check whether the resulting object is an instance of a particular class, or of one of a tuple of classes. (Added by Georg Brandl; issue 7031.)
- `assertGreater()`, `assertGreaterEqual()`, `assertLess()`, and `assertLessEqual()` compare two quantities.
- `assertMultiLineEqual()` compares two strings, and if they're not equal, displays a helpful comparison that highlights the differences in the two strings. This comparison is now used by default when Unicode strings are compared with `assertEqual()`.
- `assertRegexpMatches()` and `assertNotRegexpMatches()` checks whether the first argument is a string matching or not matching the regular expression provided as the second argument (issue 8038).
- `assertRaisesRegexp()` checks whether a particular exception is raised, and then also checks that the string representation of the exception matches the provided regular expression.
- `assertIn()` and `assertNotIn()` tests whether *first* is or is not in *second*.
- `assertItemsEqual()` tests whether two provided sequences contain the same elements.
- `assertSetEqual()` compares whether two sets are equal, and only reports the differences between the sets in case of error.
- Similarly, `assertListEqual()` and `assertTupleEqual()` compare the specified types and explain any differences without necessarily printing their full values; these methods are now used by default when comparing lists and tuples using `assertEqual()`. More generally, `assertSequenceEqual()` compares two sequences and can optionally check whether both sequences are of a particular type.
- `assertDictEqual()` compares two dictionaries and reports the differences; it's now used by default when you compare two dictionaries using `assertEqual()`. `assertDictContainsSubset()` checks whether all of the key/value pairs in *first* are found in *second*.
- `assertAlmostEqual()` and `assertNotAlmostEqual()` test whether *first* and *second* are approximately equal. This method can either round their difference to an optionally-specified number of *places* (the default is 7) and compare it to zero, or require the difference to be smaller than a supplied *delta* value.
- `loadTestsFromName()` properly honors the `suiteClass` attribute of the `TestLoader`. (Fixed by Mark Roddy; issue 6866.)
- A new hook lets you extend the `assertEqual()` method to handle new data types. The `addTypeEqualityFunc()` method takes a type object and a function. The function will be used when both of the objects being compared are of the specified type. This function should compare the two objects and raise an exception if they don't match; it's a good idea for the function to provide additional information about why the two objects aren't matching, much as the new sequence comparison methods do.

`unittest.main()` now takes an optional `exit` argument. If False, `main()` doesn't call `sys.exit()`, allowing `main()` to be used from the interactive interpreter. (Contributed by J. Pablo Fernández; issue 3379.)

`TestResult` has new `startTestRun()` and `stopTestRun()` methods that are called immediately before and after a test run. (Contributed by Robert Collins; issue 5728.)

With all these changes, the `unittest.py` was becoming awkwardly large, so the module was turned into a package and the code split into several files (by Benjamin Peterson). This doesn't affect how the module is imported or used.

See also:

http://www.voidspace.org.uk/python/articles/unittest2.shtml Describes the new features, how to use them, and the rationale for various design decisions. (By Michael Foord.)

11.5 Updated module: ElementTree 1.3

The version of the ElementTree library included with Python was updated to version 1.3. Some of the new features are:

- The various parsing functions now take a *parser* keyword argument giving an `XMLParser` instance that will be used. This makes it possible to override the file's internal encoding:

  ```
  p = ET.XMLParser(encoding='utf-8')
  t = ET.XML("""<root/>""", parser=p)
  ```

  Errors in parsing XML now raise a `ParseError` exception, whose instances have a `position` attribute containing a (*line*, *column*) tuple giving the location of the problem.

- ElementTree's code for converting trees to a string has been significantly reworked, making it roughly twice as fast in many cases. The `ElementTree.write()` and `Element.write()` methods now have a *method* parameter that can be "xml" (the default), "html", or "text". HTML mode will output empty elements as `<empty></empty>` instead of `<empty/>`, and text mode will skip over elements and only output the text chunks. If you set the `tag` attribute of an element to `None` but leave its children in place, the element will be omitted when the tree is written out, so you don't need to do more extensive rearrangement to remove a single element.

  Namespace handling has also been improved. All `xmlns:<whatever>` declarations are now output on the root element, not scattered throughout the resulting XML. You can set the default namespace for a tree by setting the `default_namespace` attribute and can register new prefixes with `register_namespace()`. In XML mode, you can use the true/false *xml_declaration* parameter to suppress the XML declaration.

- New `Element` method: `extend()` appends the items from a sequence to the element's children. Elements themselves behave like sequences, so it's easy to move children from one element to another:

  ```
  from xml.etree import ElementTree as ET

  t = ET.XML("""<list>
    <item>1</item> <item>2</item>  <item>3</item>
  </list>""")
  new = ET.XML('<root/>')
  new.extend(t)

  # Outputs <root><item>1</item>...</root>
  print ET.tostring(new)
  ```

- New `Element` method: `iter()` yields the children of the element as a generator. It's also possible to write `for child in elem:` to loop over an element's children. The existing method `getiterator()` is now deprecated, as is `getchildren()` which constructs and returns a list of children.

- New `Element` method: `itertext()` yields all chunks of text that are descendants of the element. For example:

  ```
  t = ET.XML("""<list>
    <item>1</item> <item>2</item>  <item>3</item>
  </list>""")

  # Outputs ['\n  ', '1', ' ', '2', '  ', '3', '\n']
  print list(t.itertext())
  ```

- Deprecated: using an element as a Boolean (i.e., `if elem:`) would return true if the element had any children, or false if there were no children. This behaviour is confusing – `None` is false, but so is a childless element? – so it will now trigger a `FutureWarning`. In your code, you should be explicit: write `len(elem) != 0` if you're interested in the number of children, or `elem is not None`.

Fredrik Lundh develops ElementTree and produced the 1.3 version; you can read his article describing 1.3 at http://effbot.org/zone/elementtree-13-intro.htm. Florent Xicluna updated the version included with Python, after discussions on python-dev and in issue 6472.)

12 Build and C API Changes

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

- The latest release of the GNU Debugger, GDB 7, can be scripted using Python. When you begin debugging an executable program P, GDB will look for a file named `P-gdb.py` and automatically read it. Dave Malcolm contributed a `python-gdb.py` that adds a number of commands useful when debugging Python itself. For example, `py-up` and `py-down` go up or down one Python stack frame, which usually corresponds to several C stack frames. `py-print` prints the value of a Python variable, and `py-bt` prints the Python stack trace. (Added as a result of issue 8032.)
- If you use the `.gdbinit` file provided with Python, the "pyo" macro in the 2.7 version now works correctly when the thread being debugged doesn't hold the GIL; the macro now acquires it before printing. (Contributed by Victor Stinner; issue 3632.)
- `Py_AddPendingCall()` is now thread-safe, letting any worker thread submit notifications to the main Python thread. This is particularly useful for asynchronous IO operations. (Contributed by Kristján Valur Jónsson; issue 4293.)
- New function: `PyCode_NewEmpty()` creates an empty code object; only the filename, function name, and first line number are required. This is useful for extension modules that are attempting to construct a more useful traceback stack. Previously such extensions needed to call `PyCode_New()`, which had many more arguments. (Added by Jeffrey Yasskin.)
- New function: `PyErr_NewExceptionWithDoc()` creates a new exception class, just as the existing `PyErr_NewException()` does, but takes an extra `char *` argument containing the docstring for the new exception class. (Added by 'lekma' on the Python bug tracker; issue 7033.)
- New function: `PyFrame_GetLineNumber()` takes a frame object and returns the line number that the frame is currently executing. Previously code would need to get the index of the bytecode instruction currently executing, and then look up the line number corresponding to that address. (Added by Jeffrey Yasskin.)
- New functions: `PyLong_AsLongAndOverflow()` and `PyLong_AsLongLongAndOverflow()` approximates a Python long integer as a C `long` or `long long`. If the number is too large to fit into the output type, an *overflow* flag is set and returned to the caller. (Contributed by Case Van Horsen; issue 7528 and issue 7767.)
- New function: stemming from the rewrite of string-to-float conversion, a new `PyOS_string_to_double()` function was added. The old `PyOS_ascii_strtod()` and `PyOS_ascii_atof()` functions are now deprecated.
- New function: `PySys_SetArgvEx()` sets the value of `sys.argv` and can optionally update `sys.path` to include the directory containing the script named by `sys.argv[0]` depending on the value of an *updatepath* parameter.

  This function was added to close a security hole for applications that embed Python. The old function, `PySys_SetArgv()`, would always update `sys.path`, and sometimes it would add the current directory.

This meant that, if you ran an application embedding Python in a directory controlled by someone else, attackers could put a Trojan-horse module in the directory (say, a file named `os.py`) that your application would then import and run.

If you maintain a C/C++ application that embeds Python, check whether you're calling `PySys_SetArgv()` and carefully consider whether the application should be using `PySys_SetArgvEx()` with *updatepath* set to false.

Security issue reported as CVE-2008-5983; discussed in issue 5753, and fixed by Antoine Pitrou.

- New macros: the Python header files now define the following macros: `Py_ISALNUM`, `Py_ISALPHA`, `Py_ISDIGIT`, `Py_ISLOWER`, `Py_ISSPACE`, `Py_ISUPPER`, `Py_ISXDIGIT`, `Py_TOLOWER`, and `Py_TOUPPER`. All of these functions are analogous to the C standard macros for classifying characters, but ignore the current locale setting, because in several places Python needs to analyze characters in a locale-independent way. (Added by Eric Smith; issue 5793.)

- Removed function: `PyEval_CallObject` is now only available as a macro. A function version was being kept around to preserve ABI linking compatibility, but that was in 1997; it can certainly be deleted by now. (Removed by Antoine Pitrou; issue 8276.)

- New format codes: the `PyFormat_FromString()`, `PyFormat_FromStringV()`, and `PyErr_Format()` functions now accept `%lld` and `%llu` format codes for displaying C's `long long` types. (Contributed by Mark Dickinson; issue 7228.)

- The complicated interaction between threads and process forking has been changed. Previously, the child process created by `os.fork()` might fail because the child is created with only a single thread running, the thread performing the `os.fork()`. If other threads were holding a lock, such as Python's import lock, when the fork was performed, the lock would still be marked as "held" in the new process. But in the child process nothing would ever release the lock, since the other threads weren't replicated, and the child process would no longer be able to perform imports.

  Python 2.7 acquires the import lock before performing an `os.fork()`, and will also clean up any locks created using the `threading` module. C extension modules that have internal locks, or that call `fork()` themselves, will not benefit from this clean-up.

  (Fixed by Thomas Wouters; issue 1590864.)

- The `Py_Finalize()` function now calls the internal `threading._shutdown()` function; this prevents some exceptions from being raised when an interpreter shuts down. (Patch by Adam Olsen; issue 1722344.)

- When using the `PyMemberDef` structure to define attributes of a type, Python will no longer let you try to delete or set a `T_STRING_INPLACE` attribute.

- Global symbols defined by the `ctypes` module are now prefixed with `Py`, or with `_ctypes`. (Implemented by Thomas Heller; issue 3102.)

- New configure option: the *`--with-system-expat`* switch allows building the `pyexpat` module to use the system Expat library. (Contributed by Arfrever Frehtes Taifersar Arahesis; issue 7609.)

- New configure option: the *`--with-valgrind`* option will now disable the pymalloc allocator, which is difficult for the Valgrind memory-error detector to analyze correctly. Valgrind will therefore be better at detecting memory leaks and overruns. (Contributed by James Henstridge; issue 2422.)

- New configure option: you can now supply an empty string to *`--with-dbmliborder=`* in order to disable all of the various DBM modules. (Added by Arfrever Frehtes Taifersar Arahesis; issue 6491.)

- The **configure** script now checks for floating-point rounding bugs on certain 32-bit Intel chips and defines a `X87_DOUBLE_ROUNDING` preprocessor definition. No code currently uses this definition, but it's available if anyone wishes to use it. (Added by Mark Dickinson; issue 2937.)

  configure also now sets a `LDCXXSHARED` Makefile variable for supporting C++ linking. (Contributed by Arfrever Frehtes Taifersar Arahesis; issue 1222585.)

- The build process now creates the necessary files for pkg-config support. (Contributed by Clinton Roy; issue 3585.)
- The build process now supports Subversion 1.7. (Contributed by Arfrever Frehtes Taifersar Arahesis; issue 6094.)

12.1 Capsules

Python 3.1 adds a new C datatype, `PyCapsule`, for providing a C API to an extension module. A capsule is essentially the holder of a C `void *` pointer, and is made available as a module attribute; for example, the `socket` module's API is exposed as `socket.CAPI`, and `unicodedata` exposes `ucnhash_CAPI`. Other extensions can import the module, access its dictionary to get the capsule object, and then get the `void *` pointer, which will usually point to an array of pointers to the module's various API functions.

There is an existing data type already used for this, `PyCObject`, but it doesn't provide type safety. Evil code written in pure Python could cause a segmentation fault by taking a `PyCObject` from module A and somehow substituting it for the `PyCObject` in module B. Capsules know their own name, and getting the pointer requires providing the name:

```
void *vtable;

if (!PyCapsule_IsValid(capsule, "mymodule.CAPI") {
        PyErr_SetString(PyExc_ValueError, "argument type invalid");
        return NULL;
}

vtable = PyCapsule_GetPointer(capsule, "mymodule.CAPI");
```

You are assured that `vtable` points to whatever you're expecting. If a different capsule was passed in, `PyCapsule_IsValid()` would detect the mismatched name and return false. Refer to *using-capsules* for more information on using these objects.

Python 2.7 now uses capsules internally to provide various extension-module APIs, but the `PyCObject_AsVoidPtr()` was modified to handle capsules, preserving compile-time compatibility with the `CObject` interface. Use of `PyCObject_AsVoidPtr()` will signal a `PendingDeprecationWarning`, which is silent by default.

Implemented in Python 3.1 and backported to 2.7 by Larry Hastings; discussed in issue 5630.

12.2 Port-Specific Changes: Windows

- The `msvcrt` module now contains some constants from the `crtassem.h` header file: `CRT_ASSEMBLY_VERSION`, `VC_ASSEMBLY_PUBLICKEYTOKEN`, and `LIBRARIES_ASSEMBLY_NAME_PREFIX`. (Contributed by David Cournapeau; issue 4365.)
- The `_winreg` module for accessing the registry now implements the `CreateKeyEx()` and `DeleteKeyEx()` functions, extended versions of previously-supported functions that take several extra arguments. The `DisableReflectionKey()`, `EnableReflectionKey()`, and `QueryReflectionKey()` were also tested and documented. (Implemented by Brian Curtin: issue 7347.)
- The new `_beginthreadex()` API is used to start threads, and the native thread-local storage functions are now used. (Contributed by Kristján Valur Jónsson; issue 3582.)
- The `os.kill()` function now works on Windows. The signal value can be the constants `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, or any integer. The first two constants will send `Control-C` and `Control-Break`

keystroke events to subprocesses; any other value will use the `TerminateProcess()` API. (Contributed by Miki Tebeka; issue 1220212.)

- The `os.listdir()` function now correctly fails for an empty path. (Fixed by Hirokazu Yamamoto; issue 5913.)
- The `mimelib` module will now read the MIME database from the Windows registry when initializing. (Patch by Gabriel Genellina; issue 4969.)

12.3 Port-Specific Changes: Mac OS X

- The path `/Library/Python/2.7/site-packages` is now appended to `sys.path`, in order to share added packages between the system installation and a user-installed copy of the same version. (Changed by Ronald Oussoren; issue 4865.)

12.4 Port-Specific Changes: FreeBSD

- FreeBSD 7.1's `SO_SETFIB` constant, used with `getsockopt()`/`setsockopt()` to select an alternate routing table, is now available in the `socket` module. (Added by Kyle VanderBeek; issue 8235.)

13 Other Changes and Fixes

- Two benchmark scripts, `iobench` and `ccbench`, were added to the `Tools` directory. `iobench` measures the speed of the built-in file I/O objects returned by `open()` while performing various operations, and `ccbench` is a concurrency benchmark that tries to measure computing throughput, thread switching latency, and IO processing bandwidth when performing several tasks using a varying number of threads.
- The `Tools/i18n/msgfmt.py` script now understands plural forms in `.po` files. (Fixed by Martin von Löwis; issue 5464.)
- When importing a module from a `.pyc` or `.pyo` file with an existing `.py` counterpart, the `co_filename` attributes of the resulting code objects are overwritten when the original filename is obsolete. This can happen if the file has been renamed, moved, or is accessed through different paths. (Patch by Ziga Seilnacht and Jean-Paul Calderone; issue 1180193.)
- The `regrtest.py` script now takes a *`--randseed=`* switch that takes an integer that will be used as the random seed for the *`-r`* option that executes tests in random order. The *`-r`* option also reports the seed that was used (Added by Collin Winter.)
- Another `regrtest.py` switch is *`-j`*, which takes an integer specifying how many tests run in parallel. This allows reducing the total runtime on multi-core machines. This option is compatible with several other options, including the *`-R`* switch which is known to produce long runtimes. (Added by Antoine Pitrou, issue 6152.) This can also be used with a new *`-F`* switch that runs selected tests in a loop until they fail. (Added by Antoine Pitrou; issue 7312.)
- When executed as a script, the `py_compile.py` module now accepts `'-'` as an argument, which will read standard input for the list of filenames to be compiled. (Contributed by Piotr Ożarowski; issue 8233.)

14 Porting to Python 2.7

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

- The `range()` function processes its arguments more consistently; it will now call `__int__()` on non-float, non-integer arguments that are supplied to it. (Fixed by Alexander Belopolsky; issue 1533.)
- The string `format()` method changed the default precision used for floating-point and complex numbers from 6 decimal places to 12, which matches the precision used by `str()`. (Changed by Eric Smith; issue 5920.)
- Because of an optimization for the `with` statement, the special methods `__enter__()` and `__exit__()` must belong to the object's type, and cannot be directly attached to the object's instance. This affects new-style classes (derived from `object`) and C extension types. (issue 6101.)
- Due to a bug in Python 2.6, the *exc_value* parameter to `__exit__()` methods was often the string representation of the exception, not an instance. This was fixed in 2.7, so *exc_value* will be an instance as expected. (Fixed by Florent Xicluna; issue 7853.)
- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise `AttributeError` as you would expect. Fixed by Benjamin Peterson; issue 7604.)

In the standard library:

- Operations with `datetime` instances that resulted in a year falling outside the supported range didn't always raise `OverflowError`. Such errors are now checked more carefully and will now raise the exception. (Reported by Mark Leander, patch by Anand B. Pillai and Alexander Belopolsky; issue 7150.)
- When using `Decimal` instances with a string's `format()` method, the default alignment was previously left-alignment. This has been changed to right-alignment, which might change the output of your programs. (Changed by Mark Dickinson; issue 6857.)

  Comparisons involving a signaling NaN value (or `sNAN`) now signal `InvalidOperation` instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or `NaN`) are now hashable. (Fixed by Mark Dickinson; issue 7279.)
- The ElementTree library, `xml.etree`, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like *<?xml-stylesheet href="#style1"?>*) or comment (which looks like *<!– comment –>*). (Patch by Neil Muller; issue 2746.)
- The `readline()` method of `StringIO` objects now does nothing when a negative length is requested, as other file-like objects do. (issue 7348).
- The `syslog` module will now use the value of `sys.argv[0]` as the identifier instead of the previous default value of `'python'`. (Changed by Sean Reifschneider; issue 8451.)
- The `tarfile` module's default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; issue 7357.)
- The `urlparse` module's `urlsplit()` now handles unknown URL schemes in a fashion compliant with **RFC 3986**: if the URL is of the form `"<something>://..."`, the text before the `://` is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

  ```
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', '', '//host/filename?query', '', '')
  ```

  Python 2.7 (and Python 2.6.5) will return:

  ```
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', 'host', '/filename?query', '', '')
  ```

  (Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

For C extensions:

- C extensions that use integer format codes with the `PyArg_Parse*` family of functions will now raise a `TypeError` exception instead of triggering a `DeprecationWarning` (issue 5080).
- Use the new `PyOS_string_to_double()` function instead of the old `PyOS_ascii_strtod()` and `PyOS_ascii_atof()` functions, which are now deprecated.

For applications that embed Python:

- The `PySys_SetArgvEx()` function was added, letting applications close a security hole when the existing `PySys_SetArgv()` function was used. Check whether you're calling `PySys_SetArgv()` and carefully consider whether the application should be using `PySys_SetArgvEx()` with *updatepath* set to false.

15 New Features Added to Python 2.7 Maintenance Releases

New features may be added to Python 2.7 maintenance releases when the situation genuinely calls for it. Any such additions must go through the Python Enhancement Proposal process, and make a compelling case for why they can't be adequately addressed by either adding the new feature solely to Python 3, or else by publishing it on the Python Package Index.

In addition to the specific proposals listed below, there is a general exemption allowing new `-3` warnings to be added in any Python 2.7 maintenance release.

15.1 PEP 434: IDLE Enhancement Exception for All Branches

PEP 434 describes a general exemption for changes made to the IDLE development environment shipped along with Python. This exemption makes it possible for the IDLE developers to provide a more consistent user experience across all supported versions of Python 2 and 3.

For details of any IDLE changes, refer to the NEWS file for the specific release.

15.2 PEP 466: Network Security Enhancements for Python 2.7

PEP 466 describes a number of network security enhancement proposals that have been approved for inclusion in Python 2.7 maintenance releases, with the first of those changes appearing in the Python 2.7.7 release.

PEP 466 related features added in Python 2.7.7:

- `hmac.compare_digest()` was backported from Python 3 to make a timing attack resistant comparison operation available to Python 2 applications. (Contributed by Alex Gaynor; issue 21306.)
- OpenSSL 1.0.1g was upgraded in the official Windows installers published on python.org. (Contributed by Zachary Ware; issue 21462.)

PEP 466 related features added in Python 2.7.8:

- `hashlib.pbkdf2_hmac()` was backported from Python 3 to make a hashing algorithm suitable for secure password storage broadly available to Python 2 applications. (Contributed by Alex Gaynor; issue 21304.)
- OpenSSL 1.0.1h was upgraded for the official Windows installers published on python.org. (contributed by Zachary Ware in issue 21671 for CVE-2014-0224)

PEP 466 related features added in Python 2.7.9:

- Most of Python 3.4's `ssl` module was backported. This means `ssl` now supports Server Name Indication, TLS1.x settings, access to the platform certificate store, the `SSLContext` class, and other features. (Contributed by Alex Gaynor and David Reid; issue 21308.)

  Refer to the "Version added: 2.7.9" notes in the module documentation for specific details.

- `os.urandom()` was changed to cache a file descriptor to `/dev/urandom` instead of reopening `/dev/urandom` on every call. (Contributed by Alex Gaynor; issue 21305.)

- `hashlib.algorithms_guaranteed` and `hashlib.algorithms_available` were backported from Python 3 to make it easier for Python 2 applications to select the strongest available hash algorithm. (Contributed by Alex Gaynor in issue 21307)

15.3 PEP 477: Backport ensurepip (PEP 453) to Python 2.7

PEP 477 approves the inclusion of the **PEP 453** ensurepip module and the improved documentation that was enabled by it in the Python 2.7 maintenance releases, appearing first in the the Python 2.7.9 release.

Bootstrapping pip By Default

The new `ensurepip` module (defined in **PEP 453**) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations. The version of `pip` included with Python 2.7.9 is `pip` 1.5.6, and future 2.7.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pip`, `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies.

For CPython *source builds on POSIX systems*, the `make install` and `make altinstall` commands do not bootstrap `pip` by default. This behaviour can be controlled through configure options, and overridden through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As discussed in the PEP, platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

Documentation Changes

As part of this change, the *installing-index* and *distributing-index* sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide and the documentation of the individual projects.

However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as *install-index* and *distutils-index*.

See also:

PEP 453 – Explicit bootstrapping of pip in Python installations PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.

15.4 PEP 476: Enabling certificate verification by default for stdlib http clients

`httplib` and modules which use it, such as `urllib2` and `xmlrpclib`, will now verify that the server presents a certificate which is signed by a CA in the platform trust store and whose hostname matches the hostname being requested by default, significantly improving security for many applications.

For applications which require the old previous behavior, they can pass an alternate context:

```
import urllib2
import ssl

# This disables all verification
context = ssl._create_unverified_context()

# This allows using a specific certificate for the host, which doesn't need
# to be in the trust store
context = ssl.create_default_context(cafile="/path/to/file.crt")

urllib2.urlopen("https://invalid-cert", context=context)
```

16 Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Nick Coghlan, Philip Jenvey, Ryan Lovett, R. David Murray, Hugh Secker-Walker.

Index