What's New in Python

Release 3.2

A. M. Kuchling

May 12, 2011

Python Software Foundation
Email: docs@python.org

Contents

Author Raymond Hettinger

Release 3.2

Date May 12, 2011

This article explains the new features in Python 3.2 as compared to 3.1. It focuses on a few highlights and gives a few examples. For full details, see the Misc/NEWS file.

See Also:

PEP 392 - Python 3.2 Release Schedule

1 PEP 384: Defining a Stable ABI

In the past, extension modules built for one Python version were often not usable with other Python versions. Particularly on Windows, every feature release of Python required rebuilding all extension modules that one wanted to use. This requirement was the result of the free access to Python interpreter internals that extension modules could use.

With Python 3.2, an alternative approach becomes available: extension modules which restrict themselves to a limited API (by defining Py_LIMITED_API) cannot use many of the internals, but are constrained to a set of API functions that are promised to be stable for several releases. As a consequence, extension modules built for 3.2 in that mode will also work with 3.3, 3.4, and so on. Extension modules that make use of details of memory structures can still be built, but will need to be recompiled for every feature release.

See Also:

PEP 384 - Defining a Stable ABI PEP written by Martin von Löwis.

2 PEP 389: Argparse Command Line Parsing Module

A new module for command line parsing, `argparse`, was introduced to overcome the limitations of `optparse` which did not provide support for positional arguments (not just options), subcommands, required options and other common patterns of specifying and validating options.

This module has already had widespread success in the community as a third-party module. Being more fully featured than its predecessor, the `argparse` module is now the preferred module for command-line processing. The older module is still being kept available because of the substantial amount of legacy code that depends on it.

Here's an annotated example parser showing features like limiting results to a set of choices, specifying a *metavar* in the help screen, validating that one or more positional arguments is present, and making a required option:

```
import argparse
parser = argparse.ArgumentParser(
            description = 'Manage servers',          # main description for help
            epilog = 'Tested on Solaris and Linux')  # displayed after help
parser.add_argument('action',                        # argument name
            choices = ['deploy', 'start', 'stop'],   # three allowed values
            help = 'action on each target')          # help msg
parser.add_argument('targets',
            metavar = 'HOSTNAME',                    # var name used in help msg
            nargs = '+',                             # require one or more targets
            help = 'url for target machines')        # help msg explanation
parser.add_argument('-u', '--user',                  # -u or --user option
            required = True,                         # make it a required argument
            help = 'login as user')
```

Example of calling the parser on a command string:

```
>>> cmd  = 'deploy sneezy.example.com sleepy.example.com -u skycaptain'
>>> result = parser.parse_args(cmd.split())
>>> result.action
'deploy'
>>> result.targets
['sneezy.example.com', 'sleepy.example.com']
>>> result.user
'skycaptain'
```

Example of the parser's automatically generated help:

```
>>> parser.parse_args('-h'.split())

usage: manage_cloud.py [-h] -u USER
                       {deploy,start,stop} HOSTNAME [HOSTNAME ...]

Manage servers

positional arguments:
  {deploy,start,stop}   action on each target
  HOSTNAME              url for target machines

optional arguments:
  -h, --help            show this help message and exit
  -u USER, --user USER  login as user

Tested on Solaris and Linux
```

An especially nice `argparse` feature is the ability to define subparsers, each with their own argument patterns and help displays:

```
import argparse
parser = argparse.ArgumentParser(prog='HELM')
subparsers = parser.add_subparsers()

parser_l = subparsers.add_parser('launch', help='Launch Control')   # first subgroup
parser_l.add_argument('-m', '--missiles', action='store_true')
parser_l.add_argument('-t', '--torpedos', action='store_true')

parser_m = subparsers.add_parser('move', help='Move Vessel',        # second subgroup
                                 aliases=('steer', 'turn'))         # equivalent names
parser_m.add_argument('-c', '--course', type=int, required=True)
parser_m.add_argument('-s', '--speed', type=int, default=0)

$ ./helm.py --help                         # top level help (launch and move)
$ ./helm.py launch --help                  # help for launch options
$ ./helm.py launch --missiles              # set missiles=True and torpedos=False
$ ./helm.py steer --course 180 --speed 5   # set movement parameters
```

See Also:

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

upgrading-optparse-code for details on the differences from `optparse`.

3 PEP 391: Dictionary Based Configuration for Logging

The `logging` module provided two kinds of configuration, one style with function calls for each option or another style driven by an external file saved in a `ConfigParser` format. Those options did not provide the flexibility to create configurations from JSON or YAML files, nor did they support incremental configuration, which is needed for specifying logger options from a command line.

To support a more flexible style, the module now offers `logging.config.dictConfig()` for specifying logging configuration with plain Python dictionaries. The configuration options include formatters, handlers, filters, and loggers. Here's a working example of a configuration dictionary:

```
{"version": 1,
 "formatters": {"brief": {"format": "%(levelname)-8s: %(name)-15s: %(message)s"},
                "full": {"format": "%(asctime)s %(name)-15s %(levelname)-8s %(message)s"}
                },
 "handlers": {"console": {
                    "class": "logging.StreamHandler",
                    "formatter": "brief",
                    "level": "INFO",
                    "stream": "ext://sys.stdout"},
              "console_priority": {
                    "class": "logging.StreamHandler",
                    "formatter": "full",
                    "level": "ERROR",
                    "stream": "ext://sys.stderr"}
              },
 "root": {"level": "DEBUG", "handlers": ["console", "console_priority"]}}
```

If that dictionary is stored in a file called `conf.json`, it can be loaded and called with code like this:

```
>>> import json, logging.config
>>> with open('conf.json') as f:
        conf = json.load(f)
>>> logging.config.dictConfig(conf)
>>> logging.info("Transaction completed normally")
INFO    : root            : Transaction completed normally
>>> logging.critical("Abnormal termination")
2011-02-17 11:14:36,694 root            CRITICAL Abnormal termination
```

See Also:

PEP 391 - Dictionary Based Configuration for Logging PEP written by Vinay Sajip.

4 PEP 3148: The `concurrent.futures` module

Code for creating and managing concurrency is being collected in a new top-level namespace, *concurrent*. Its first member is a *futures* package which provides a uniform high-level interface for managing threads and processes.

The design for `concurrent.futures` was inspired by *java.util.concurrent.package*. In that model, a running call and its result are represented by a `Future` object that abstracts features common to threads, processes, and remote procedure calls. That object supports status checks (running or done), timeouts, cancellations, adding callbacks, and access to results or exceptions.

The primary offering of the new module is a pair of executor classes for launching and managing calls. The goal of the executors is to make it easier to use existing tools for making parallel calls. They save the effort needed to setup a

pool of resources, launch the calls, create a results queue, add time-out handling, and limit the total number of threads, processes, or remote procedure calls.

Ideally, each application should share a single executor across multiple components so that process and thread limits can be centrally managed. This solves the design challenge that arises when each component has its own competing strategy for resource management.

Both classes share a common interface with three methods: `submit()` for scheduling a callable and returning a `Future` object; `map()` for scheduling many asynchronous calls at a time, and `shutdown()` for freeing resources. The class is a *context manager* and can be used in a `with` statement to assure that resources are automatically released when currently pending futures are done executing.

A simple of example of `ThreadPoolExecutor` is a launch of four parallel threads for copying files:

```
import concurrent.futures, shutil
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

See Also:

PEP 3148 - Futures – Execute Computations Asynchronously PEP written by Brian Quinlan.

Code for Threaded Parallel URL reads, an example using threads to fetch multiple web pages in parallel.

Code for computing prime numbers in parallel, an example demonstrating `ProcessPoolExecutor`.

5 PEP 3147: PYC Repository Directories

Python's scheme for caching bytecode in *.pyc* files did not work well in environments with multiple Python interpreters. If one interpreter encountered a cached file created by another interpreter, it would recompile the source and overwrite the cached file, thus losing the benefits of caching.

The issue of "pyc fights" has become more pronounced as it has become commonplace for Linux distributions to ship with multiple versions of Python. These conflicts also arise with CPython alternatives such as Unladen Swallow.

To solve this problem, Python's import machinery has been extended to use distinct filenames for each interpreter. Instead of Python 3.2 and Python 3.3 and Unladen Swallow each competing for a file called "mymodule.pyc", they will now look for "mymodule.cpython-32.pyc", "mymodule.cpython-33.pyc", and "mymodule.unladen10.pyc". And to prevent all of these new files from cluttering source directories, the *pyc* files are now collected in a "__pycache__" directory stored under the package directory.

Aside from the filenames and target directories, the new scheme has a few aspects that are visible to the programmer:

- Imported modules now have a `__cached__` attribute which stores the name of the actual file that was imported:

  ```
  >>> import collections
  >>> collections.__cached__
  'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
  ```

- The tag that is unique to each interpreter is accessible from the `imp` module:

  ```
  >>> import imp
  >>> imp.get_tag()
  'cpython-32'
  ```

- Scripts that try to deduce source filename from the imported file now need to be smarter. It is no longer sufficient to simply strip the "c" from a ".pyc" filename. Instead, use the new functions in the `imp` module:

```
>>> imp.source_from_cache('c:/py32/lib/__pycache__/collections.cpython-32.pyc')
'c:/py32/lib/collections.py'
>>> imp.cache_from_source('c:/py32/lib/collections.py')
'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
```

- The `py_compile` and `compileall` modules have been updated to reflect the new naming convention and target directory. The command-line invocation of *compileall* has new options: `-i` for specifying a list of files and directories to compile and `-b` which causes bytecode files to be written to their legacy location rather than *__pycache__*.
- The `importlib.abc` module has been updated with new *abstract base classes* for loading bytecode files. The obsolete ABCs, `PyLoader` and `PyPycLoader`, have been deprecated (instructions on how to stay Python 3.1 compatible are included with the documentation).

See Also:

PEP 3147 - PYC Repository Directories PEP written by Barry Warsaw.

6 PEP 3149: ABI Version Tagged .so Files

The PYC repository directory allows multiple bytecode cache files to be co-located. This PEP implements a similar mechanism for shared object files by giving them a common directory and distinct names for each version.

The common directory is "pyshared" and the file names are made distinct by identifying the Python implementation (such as CPython, PyPy, Jython, etc.), the major and minor version numbers, and optional build flags (such as "d" for debug, "m" for pymalloc, "u" for wide-unicode). For an arbitrary package "foo", you may see these files when the distribution package is installed:

```
/usr/share/pyshared/foo.cpython-32m.so
/usr/share/pyshared/foo.cpython-33md.so
```

In Python itself, the tags are accessible from functions in the `sysconfig` module:

```
>>> import sysconfig
>>> sysconfig.get_config_var('SOABI')     # find the version tag
'cpython-32mu'
>>> sysconfig.get_config_var('SO')        # find the full filename extension
'.cpython-32mu.so'
```

See Also:

PEP 3149 - ABI Version Tagged .so Files PEP written by Barry Warsaw.

7 PEP 3333: Python Web Server Gateway Interface v1.0.1

This informational PEP clarifies how bytes/text issues are to be handled by the WGSI protocol. The challenge is that string handling in Python 3 is most conveniently handled with the `str` type even though the HTTP protocol is itself bytes oriented.

The PEP differentiates so-called *native strings* that are used for request/response headers and metadata versus *byte strings* which are used for the bodies of requests and responses.

The *native strings* are always of type `str` but are restricted to code points between *U+0000* through *U+00FF* which are translatable to bytes using *Latin-1* encoding. These strings are used for the keys and values in the environment dictionary and for response headers and statuses in the `start_response()` function. They must follow **RFC 2616** with respect to encoding. That is, they must either be *ISO-8859-1* characters or use

RFC 2047 MIME encoding.

For developers porting WSGI applications from Python 2, here are the salient points:

- If the app already used strings for headers in Python 2, no change is needed.
- If instead, the app encoded output headers or decoded input headers, then the headers will need to be re-encoded to Latin-1. For example, an output header encoded in utf-8 was using `h.encode('utf-8')` now needs to convert from bytes to native strings using `h.encode('utf-8').decode('latin-1')`.
- Values yielded by an application or sent using the `write()` method must be byte strings. The `start_response()` function and environ must use native strings. The two cannot be mixed.

For server implementers writing CGI-to-WSGI pathways or other CGI-style protocols, the users must to be able access the environment using native strings even though the underlying platform may have a different convention. To bridge this gap, the `wsgiref` module has a new function, `wsgiref.handlers.read_environ()` for transcoding CGI variables from `os.environ` into native strings and returning a new dictionary.

See Also:

PEP 3333 - Python Web Server Gateway Interface v1.0.1 PEP written by Phillip Eby.

8 Other Language Changes

Some smaller changes made to the core Python language are:

- String formatting for `format()` and `str.format()` gained new capabilities for the format character **#**. Previously, for integers in binary, octal, or hexadecimal, it caused the output to be prefixed with '0b', '0o', or '0x' respectively. Now it can also handle floats, complex, and Decimal, causing the output to always have a decimal point even when no digits follow it.

  ```
  >>> format(20, '#o')
  '0o24'
  >>> format(12.34, '#5.0f')
  '  12.'
  ```

  (Suggested by Mark Dickinson and implemented by Eric Smith in issue 7094.)

- There is also a new `str.format_map()` method that extends the capabilities of the existing `str.format()` method by accepting arbitrary *mapping* objects. This new method makes it possible to use string formatting with any of Python's many dictionary-like objects such as `defaultdict`, `Shelf`, `ConfigParser`, or `dbm`. It is also useful with custom `dict` subclasses that normalize keys before look-up or that supply a `__missing__()` method for unknown keys:

  ```
  >>> import shelve
  >>> d = shelve.open('tmp.shl')
  >>> 'The {project_name} status is {status} as of {date}'.format_map(d)
  'The testing project status is green as of February 15, 2011'

  >>> class LowerCasedDict(dict):
          def __getitem__(self, key):
              return dict.__getitem__(self, key.lower())
  >>> lcd = LowerCasedDict(part='widgets', quantity=10)
  >>> 'There are {QUANTITY} {Part} in stock'.format_map(lcd)
  'There are 10 widgets in stock'

  >>> class PlaceholderDict(dict):
          def __missing__(self, key):
  ```

```
            return '<{}>'.format(key)
>>> 'Hello {name}, welcome to {location}'.format_map(PlaceholderDict())
'Hello <name>, welcome to <location>'
```

(Suggested by Raymond Hettinger and implemented by Eric Smith in issue 6081.)

- The interpreter can now be started with a quiet option, `-q`, to prevent the copyright and version information from being displayed in the interactive mode. The option can be introspected using the `sys.flags` attribute:

```
$ python -q
>>> sys.flags
sys.flags(debug=0, division_warning=0, inspect=0, interactive=0,
optimize=0, dont_write_bytecode=0, no_user_site=0, no_site=0,
ignore_environment=0, verbose=0, bytes_warning=0, quiet=1)
```

(Contributed by Marcin Wojdyr in issue 1772833).

- The `hasattr()` function works by calling `getattr()` and detecting whether an exception is raised. This technique allows it to detect methods created dynamically by `__getattr__()` or `__getattribute__()` which would otherwise be absent from the class dictionary. Formerly, *hasattr* would catch any exception, possibly masking genuine errors. Now, *hasattr* has been tightened to only catch `AttributeError` and let other exceptions pass through:

```
>>> class A:
        @property
        def f(self):
            return 1 // 0

>>> a = A()
>>> hasattr(a, 'f')
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

(Discovered by Yury Selivanov and fixed by Benjamin Peterson; issue 9666.)

- The `str()` of a float or complex number is now the same as its `repr()`. Previously, the `str()` form was shorter but that just caused confusion and is no longer needed now that the shortest possible `repr()` is displayed by default:

```
>>> import math
>>> repr(math.pi)
'3.141592653589793'
>>> str(math.pi)
'3.141592653589793'
```

(Proposed and implemented by Mark Dickinson; issue 9337.)

- `memoryview` objects now have a `release()` method and they also now support the context manager protocol. This allows timely release of any resources that were acquired when requesting a buffer from the original object.

```
>>> with memoryview(b'abcdefgh') as v:
        print(v.tolist())
[97, 98, 99, 100, 101, 102, 103, 104]
```

(Added by Antoine Pitrou; issue 9757.)

- Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block:

```
def outer(x):
    def inner():
        return x
    inner()
    del x
```

This is now allowed. Remember that the target of an `except` clause is cleared, so this code which used to work with Python 2.6, raised a `SyntaxError` with Python 3.1 and now works again:

```
def f():
    def print_error():
        print(e)
    try:
        something
    except Exception as e:
        print_error()
        # implicit "del e" here
```

(See issue 4617.)

- The internal `structsequence` tool now creates subclasses of tuple. This means that C structures like those returned by `os.stat()`, `time.gmtime()`, and `sys.version_info` now work like a *named tuple* and now work with functions and methods that expect a tuple as an argument. This is a big step forward in making the C structures as flexible as their pure Python counterparts:

```
>>> isinstance(sys.version_info, tuple)
True
>>> 'Version %d.%d.%d %s(%d)' % sys.version_info
'Version 3.2.0 final(0)'
```

(Suggested by Arfrever Frehtes Taifersar Arahesis and implemented by Benjamin Peterson in issue 8413.)

- Warnings are now easier to control using the `PYTHONWARNINGS` environment variable as an alternative to using `-W` at the command line:

```
$ export PYTHONWARNINGS='ignore::RuntimeWarning::,once::UnicodeWarning::'
```

(Suggested by Barry Warsaw and implemented by Philip Jenvey in issue 7301.)

- A new warning category, `ResourceWarning`, has been added. It is emitted when potential issues with resource consumption or cleanup are detected. It is silenced by default in normal release builds but can be enabled through the means provided by the `warnings` module, or on the command line.

  A `ResourceWarning` is issued at interpreter shutdown if the `gc.garbage` list isn't empty, and if `gc.DEBUG_UNCOLLECTABLE` is set, all uncollectable objects are printed. This is meant to make the programmer aware that their code contains object finalization issues.

  A `ResourceWarning` is also issued when a *file object* is destroyed without having been explicitly closed. While the deallocator for such object ensures it closes the underlying operating system resource (usually, a file descriptor), the delay in deallocating the object could produce various issues, especially under Windows. Here is an example of enabling the warning from the command line:

```
$ python -q -Wdefault
>>> f = open("foo", "wb")
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.BufferedWriter name='foo'>
```

(Added by Antoine Pitrou and Georg Brandl in issue 10093 and issue 477863.)

- `range` objects now support *index* and *count* methods. This is part of an effort to make more objects fully implement the `collections.Sequence` *abstract base class*. As a result, the language will have a more

uniform API. In addition, `range` objects now support slicing and negative indices, even with values larger than `sys.maxsize`. This makes *range* more interoperable with lists:

```
>>> range(0, 100, 2).count(10)
1
>>> range(0, 100, 2).index(10)
5
>>> range(0, 100, 2)[5]
10
>>> range(0, 100, 2)[0:5]
range(0, 10, 2)
```

(Contributed by Daniel Stutzbach in issue 9213, by Alexander Belopolsky in issue 2690, and by Nick Coghlan in issue 10889.)

- The `callable()` builtin function from Py2.x was resurrected. It provides a concise, readable alternative to using an *abstract base class* in an expression like `isinstance(x, collections.Callable)`:

```
>>> callable(max)
True
>>> callable(20)
False
```

(See issue 10518.)

- Python's import mechanism can now load modules installed in directories with non-ASCII characters in the path name. This solved an aggravating problem with home directories for users with non-ASCII characters in their usernames.

(Required extensive work by Victor Stinner in issue 9425.)

9 New, Improved, and Deprecated Modules

Python's standard library has undergone significant maintenance efforts and quality improvements.

The biggest news for Python 3.2 is that the `email` package, `mailbox` module, and `nntplib` modules now work correctly with the bytes/text model in Python 3. For the first time, there is correct handling of messages with mixed encodings.

Throughout the standard library, there has been more careful attention to encodings and text versus bytes issues. In particular, interactions with the operating system are now better able to exchange non-ASCII data using the Windows MBCS encoding, locale-aware encodings, or UTF-8.

Another significant win is the addition of substantially better support for *SSL* connections and security certificates.

In addition, more classes now implement a *context manager* to support convenient and reliable resource clean-up using a `with` statement.

9.1 email

The usability of the `email` package in Python 3 has been mostly fixed by the extensive efforts of R. David Murray. The problem was that emails are typically read and stored in the form of `bytes` rather than `str` text, and they may contain multiple encodings within a single email. So, the email package had to be extended to parse and generate email messages in bytes format.

- New functions `message_from_bytes()` and `message_from_binary_file()`, and new classes `BytesFeedParser` and `BytesParser` allow binary message data to be parsed into model objects.

- Given bytes input to the model, `get_payload()` will by default decode a message body that has a *`Content-Transfer-Encoding`* of *8bit* using the charset specified in the MIME headers and return the resulting string.
- Given bytes input to the model, `Generator` will convert message bodies that have a *`Content-Transfer-Encoding`* of *8bit* to instead have a *7bit* *`Content-Transfer-Encoding`*.

  Headers with unencoded non-ASCII bytes are deemed to be **RFC 2047**-encoded using the *unknown-8bit* character set.
- A new class `BytesGenerator` produces bytes as output, preserving any unchanged non-ASCII data that was present in the input used to build the model, including message bodies with a *`Content-Transfer-Encoding`* of *8bit*.
- The `smtplib` `SMTP` class now accepts a byte string for the *msg* argument to the `sendmail()` method, and a new method, `send_message()` accepts a `Message` object and can optionally obtain the *from_addr* and *to_addrs* addresses directly from the object.

(Proposed and implemented by R. David Murray, issue 4661 and issue 10321.)

9.2 elementtree

The `xml.etree.ElementTree` package and its `xml.etree.cElementTree` counterpart have been updated to version 1.3.

Several new and useful functions and methods have been added:

- `xml.etree.ElementTree.fromstringlist()` which builds an XML document from a sequence of fragments
- `xml.etree.ElementTree.register_namespace()` for registering a global namespace prefix
- `xml.etree.ElementTree.tostringlist()` for string representation including all sublists
- `xml.etree.ElementTree.Element.extend()` for appending a sequence of zero or more elements
- `xml.etree.ElementTree.Element.iterfind()` searches an element and subelements
- `xml.etree.ElementTree.Element.itertext()` creates a text iterator over an element and its subelements
- `xml.etree.ElementTree.TreeBuilder.end()` closes the current element
- `xml.etree.ElementTree.TreeBuilder.doctype()` handles a doctype declaration

Two methods have been deprecated:

- `xml.etree.ElementTree.getchildren()` use `list(elem)` instead.
- `xml.etree.ElementTree.getiterator()` use `Element.iter` instead.

For details of the update, see Introducing ElementTree on Fredrik Lundh's website.

(Contributed by Florent Xicluna and Fredrik Lundh, issue 6472.)

9.3 functools

- The `functools` module includes a new decorator for caching function calls. `functools.lru_cache()` can save repeated queries to an external resource whenever the results are expected to be the same.

  For example, adding a caching decorator to a database query function can save database accesses for popular searches:

```
>>> import functools
>>> @functools.lru_cache(maxsize=300)
>>> def get_phone_number(name):
        c = conn.cursor()
        c.execute('SELECT phonenumber FROM phonelist WHERE name=?', (name,))
        return c.fetchone()[0]

>>> for name in user_requests:
        get_phone_number(name)         # cached lookup
```

To help with choosing an effective cache size, the wrapped function is instrumented for tracking cache statistics:

```
>>> get_phone_number.cache_info()
CacheInfo(hits=4805, misses=980, maxsize=300, currsize=300)
```

If the phonelist table gets updated, the outdated contents of the cache can be cleared with:

```
>>> get_phone_number.cache_clear()
```

(Contributed by Raymond Hettinger and incorporating design ideas from Jim Baker, Miki Tebeka, and Nick Coghlan; see recipe 498245, recipe 577479, issue 10586, and issue 10593.)

- The `functools.wraps()` decorator now adds a `__wrapped__` attribute pointing to the original callable function. This allows wrapped functions to be introspected. It also copies `__annotations__` if defined. And now it also gracefully skips over missing attributes such as `__doc__` which might not be defined for the wrapped callable.

  In the above example, the cache can be removed by recovering the original function:

  ```
  >>> get_phone_number = get_phone_number.__wrapped__     # uncached function
  ```

  (By Nick Coghlan and Terrence Cole; issue 9567, issue 3445, and issue 8814.)

- To help write classes with rich comparison methods, a new decorator `functools.total_ordering()` will use a existing equality and inequality methods to fill in the remaining methods.

  For example, supplying *__eq__* and *__lt__* will enable `total_ordering()` to fill-in *__le__*, *__gt__* and *__ge__*:

  ```
  @total_ordering
  class Student:
      def __eq__(self, other):
          return ((self.lastname.lower(), self.firstname.lower()) ==
                  (other.lastname.lower(), other.firstname.lower()))
      def __lt__(self, other):
          return ((self.lastname.lower(), self.firstname.lower()) <
                  (other.lastname.lower(), other.firstname.lower()))
  ```

  With the *total_ordering* decorator, the remaining comparison methods are filled in automatically.

  (Contributed by Raymond Hettinger.)

- To aid in porting programs from Python 2, the `functools.cmp_to_key()` function converts an old-style comparison function to modern *key function*:

  ```
  >>> # locale-aware sort order
  >>> sorted(iterable, key=cmp_to_key(locale.strcoll))
  ```

  For sorting examples and a brief sorting tutorial, see the Sorting HowTo tutorial.

  (Contributed by Raymond Hettinger.)

9.4 itertools

- The `itertools` module has a new `accumulate()` function modeled on APL's *scan* operator and Numpy's *accumulate* function:

```
>>> from itertools import accumulate
>>> list(accumulate([8, 2, 50]))
[8, 10, 60]

>>> prob_dist = [0.1, 0.4, 0.2, 0.3]
>>> list(accumulate(prob_dist))       # cumulative probability distribution
[0.1, 0.5, 0.7, 1.0]
```

  For an example using `accumulate()`, see the *examples for the random module*.

  (Contributed by Raymond Hettinger and incorporating design suggestions from Mark Dickinson.)

9.5 collections

- The `collections.Counter` class now has two forms of in-place subtraction, the existing `-=` operator for saturating subtraction and the new `subtract()` method for regular subtraction. The former is suitable for multisets which only have positive counts, and the latter is more suitable for use cases that allow negative counts:

```
>>> tally = Counter(dogs=5, cat=3)
>>> tally -= Counter(dogs=2, cats=8)     # saturating subtraction
>>> tally
Counter({'dogs': 3})

>>> tally = Counter(dogs=5, cats=3)
>>> tally.subtract(dogs=2, cats=8)       # regular subtraction
>>> tally
Counter({'dogs': 3, 'cats': -5})
```

  (Contributed by Raymond Hettinger.)

- The `collections.OrderedDict` class has a new method `move_to_end()` which takes an existing key and moves it to either the first or last position in the ordered sequence.

  The default is to move an item to the last position. This is equivalent of renewing an entry with `od[k] = od.pop(k)`.

  A fast move-to-end operation is useful for resequencing entries. For example, an ordered dictionary can be used to track order of access by aging entries from the oldest to the most recently accessed.

```
>>> d = OrderedDict.fromkeys(['a', 'b', 'X', 'd', 'e'])
>>> list(d)
['a', 'b', 'X', 'd', 'e']
>>> d.move_to_end('X')
>>> list(d)
['a', 'b', 'd', 'e', 'X']
```

  (Contributed by Raymond Hettinger.)

- The `collections.deque` class grew two new methods `count()` and `reverse()` that make them more substitutable for `list` objects:

```
>>> d = deque('simsalabim')
>>> d.count('s')
2
```

```
>>> d.reverse()
>>> d
deque(['m', 'i', 'b', 'a', 'l', 'a', 's', 'm', 'i', 's'])
```

(Contributed by Raymond Hettinger.)

9.6 threading

The `threading` module has a new `Barrier` synchronization class for making multiple threads wait until all of them have reached a common barrier point. Barriers are useful for making sure that a task with multiple preconditions does not run until all of the predecessor tasks are complete.

Barriers can work with an arbitrary number of threads. This is a generalization of a Rendezvous which is defined for only two threads.

Implemented as a two-phase cyclic barrier, `Barrier` objects are suitable for use in loops. The separate *filling* and *draining* phases assure that all threads get released (drained) before any one of them can loop back and re-enter the barrier. The barrier fully resets after each cycle.

Example of using barriers:

```
from threading import Barrier, Thread

def get_votes(site):
    ballots = conduct_election(site)
    all_polls_closed.wait()        # do not count until all polls are closed
    totals = summarize(ballots)
    publish(site, totals)

all_polls_closed = Barrier(len(sites))
for site in sites:
    Thread(target=get_votes, args=(site,)).start()
```

In this example, the barrier enforces a rule that votes cannot be counted at any polling site until all polls are closed. Notice how a solution with a barrier is similar to one with `threading.Thread.join()`, but the threads stay alive and continue to do work (summarizing ballots) after the barrier point is crossed.

If any of the predecessor tasks can hang or be delayed, a barrier can be created with an optional *timeout* parameter. Then if the timeout period elapses before all the predecessor tasks reach the barrier point, all waiting threads are released and a `BrokenBarrierError` exception is raised:

```
def get_votes(site):
    ballots = conduct_election(site)
    try:
        all_polls_closed.wait(timeout = midnight - time.now())
    except BrokenBarrierError:
        lockbox = seal_ballots(ballots)
        queue.put(lockbox)
    else:
        totals = summarize(ballots)
        publish(site, totals)
```

In this example, the barrier enforces a more robust rule. If some election sites do not finish before midnight, the barrier times-out and the ballots are sealed and deposited in a queue for later handling.

See Barrier Synchronization Patterns for more examples of how barriers can be used in parallel computing. Also, there is a simple but thorough explanation of barriers in The Little Book of Semaphores, *section 3.6*.

(Contributed by Kristján Valur Jónsson with an API review by Jeffrey Yasskin in issue 8777.)

9.7 datetime and time

- The `datetime` module has a new type `timezone` that implements the `tzinfo` interface by returning a fixed UTC offset and timezone name. This makes it easier to create timezone-aware datetime objects:

```
>>> from datetime import datetime, timezone

>>> datetime.now(timezone.utc)
datetime.datetime(2010, 12, 8, 21, 4, 2, 923754, tzinfo=datetime.timezone.utc)

>>> datetime.strptime("01/01/2000 12:00 +0000", "%m/%d/%Y %H:%M %z")
datetime.datetime(2000, 1, 1, 12, 0, tzinfo=datetime.timezone.utc)
```

- Also, `timedelta` objects can now be multiplied by `float` and divided by `float` and `int` objects. And `timedelta` objects can now divide one another.
- The `datetime.date.strftime()` method is no longer restricted to years after 1900. The new supported year range is from 1000 to 9999 inclusive.
- Whenever a two-digit year is used in a time tuple, the interpretation has been governed by `time.accept2dyear`. The default is *True* which means that for a two-digit year, the century is guessed according to the POSIX rules governing the `%y` strptime format.

  Starting with Py3.2, use of the century guessing heuristic will emit a `DeprecationWarning`. Instead, it is recommended that `time.accept2dyear` be set to *False* so that large date ranges can be used without guesswork:

```
>>> import time, warnings
>>> warnings.resetwarnings()       # remove the default warning filters

>>> time.accept2dyear = True       # guess whether 11 means 11 or 2011
>>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
Warning (from warnings module):
  ...
DeprecationWarning: Century info guessed for a 2-digit year.
'Fri Jan  1 12:34:56 2011'

>>> time.accept2dyear = False      # use the full range of allowable dates
>>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
'Fri Jan  1 12:34:56 11'
```

  Several functions now have significantly expanded date ranges. When `time.accept2dyear` is false, the `time.asctime()` function will accept any year that fits in a C int, while the `time.mktime()` and `time.strftime()` functions will accept the full range supported by the corresponding operating system functions.

(Contributed by Alexander Belopolsky and Victor Stinner in issue 1289118, issue 5094, issue 6641, issue 2706, issue 1777412, issue 8013, and issue 10827.)

9.8 math

The `math` module has been updated with six new functions inspired by the C99 standard.

The `isfinite()` function provides a reliable and fast way to detect special values. It returns *True* for regular numbers and *False* for *Nan* or *Infinity*:

```
>>> [isfinite(x) for x in (123, 4.56, float('Nan'), float('Inf'))]
[True, True, False, False]
```

The `expm1()` function computes `e**x-1` for small values of *x* without incurring the loss of precision that usually accompanies the subtraction of nearly equal quantities:

```
>>> expm1(0.013671875)   # more accurate way to compute e**x-1 for a small x
0.013765762467652909
```

The `erf()` function computes a probability integral or Gaussian error function. The complementary error function, `erfc()`, is `1 - erf(x)`:

```
>>> erf(1.0/sqrt(2.0))   # portion of normal distribution within 1 standard deviation
0.682689492137086
>>> erfc(1.0/sqrt(2.0))  # portion of normal distribution outside 1 standard deviation
0.31731050786291404
>>> erf(1.0/sqrt(2.0)) + erfc(1.0/sqrt(2.0))
1.0
```

The `gamma()` function is a continuous extension of the factorial function. See http://en.wikipedia.org/wiki/Gamma_function for details. Because the function is related to factorials, it grows large even for small values of *x*, so there is also a `lgamma()` function for computing the natural logarithm of the gamma function:

```
>>> gamma(7.0)           # six factorial
720.0
>>> lgamma(801.0)        # log(800 factorial)
4551.950730698041
```

(Contributed by Mark Dickinson.)

9.9 abc

The `abc` module now supports `abstractclassmethod()` and `abstractstaticmethod()`.

These tools make it possible to define an *abstract base class* that requires a particular `classmethod()` or `staticmethod()` to be implemented:

```
class Temperature(metaclass=abc.ABCMeta):
    @abc.abstractclassmethod
    def from_fahrenheit(cls, t):
        ...
    @abc.abstractclassmethod
    def from_celsius(cls, t):
        ...
```

(Patch submitted by Daniel Urban; issue 5867.)

9.10 io

The `io.BytesIO` has a new method, `getbuffer()`, which provides functionality similar to `memoryview()`. It creates an editable view of the data without making a copy. The buffer's random access and support for slice notation are well-suited to in-place editing:

```
>>> REC_LEN, LOC_START, LOC_LEN = 34, 7, 11

>>> def change_location(buffer, record_number, location):
        start = record_number * REC_LEN + LOC_START
        buffer[start: start+LOC_LEN] = location
```

```
>>> import io

>>> byte_stream = io.BytesIO(
    b'G3805  storeroom  Main chassis    '
    b'X7899  shipping   Reserve cog     '
    b'L6988  receiving  Primary sprocket'
)
>>> buffer = byte_stream.getbuffer()
>>> change_location(buffer, 1, b'warehouse  ')
>>> change_location(buffer, 0, b'showroom   ')
>>> print(byte_stream.getvalue())
b'G3805  showroom   Main chassis    '
b'X7899  warehouse  Reserve cog     '
b'L6988  receiving  Primary sprocket'
```

(Contributed by Antoine Pitrou in issue 5506.)

9.11 reprlib

When writing a `__repr__()` method for a custom container, it is easy to forget to handle the case where a member refers back to the container itself. Python's builtin objects such as `list` and `set` handle self-reference by displaying "..." in the recursive part of the representation string.

To help write such `__repr__()` methods, the `reprlib` module has a new decorator, `recursive_repr()`, for detecting recursive calls to `__repr__()` and substituting a placeholder string instead:

```
>>> class MyList(list):
        @recursive_repr()
        def __repr__(self):
            return '<' + '|'.join(map(repr, self)) + '>'

>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

(Contributed by Raymond Hettinger in issue 9826 and issue 9840.)

9.12 logging

In addition to dictionary-based configuration described above, the `logging` package has many other improvements.

The logging documentation has been augmented by a *basic tutorial*, an *advanced tutorial*, and a *cookbook* of logging recipes. These documents are the fastest way to learn about logging.

The `logging.basicConfig()` set-up function gained a *style* argument to support three different types of string formatting. It defaults to "%" for traditional %-formatting, can be set to "{" for the new `str.format()` style, or can be set to "$" for the shell-style formatting provided by `string.Template`. The following three configurations are equivalent:

```
>>> from logging import basicConfig
>>> basicConfig(style='%', format="%(name)s -> %(levelname)s: %(message)s")
>>> basicConfig(style='{', format="{name} -> {levelname} {message}")
>>> basicConfig(style='$', format="$name -> $levelname: $message")
```

If no configuration is set-up before a logging event occurs, there is now a default configuration using a `StreamHandler` directed to `sys.stderr` for events of `WARNING` level or higher. Formerly, an event occurring before a configuration was set-up would either raise an exception or silently drop the event depending on the value of `logging.raiseExceptions`. The new default handler is stored in `logging.lastResort`.

The use of filters has been simplified. Instead of creating a `Filter` object, the predicate can be any Python callable that returns *True* or *False*.

There were a number of other improvements that add flexibility and simplify configuration. See the module documentation for a full listing of changes in Python 3.2.

9.13 csv

The `csv` module now supports a new dialect, `unix_dialect`, which applies quoting for all fields and a traditional Unix style with `'\n'` as the line terminator. The registered dialect name is `unix`.

The `csv.DictWriter` has a new method, `writeheader()` for writing-out an initial row to document the field names:

```
>>> import csv, sys
>>> w = csv.DictWriter(sys.stdout, ['name', 'dept'], dialect='unix')
>>> w.writeheader()
"name","dept"
>>> w.writerows([
        {'name': 'tom', 'dept': 'accounting'},
        {'name': 'susan', 'dept': 'Salesl'}])
"tom","accounting"
"susan","sales"
```

(New dialect suggested by Jay Talbot in issue 5975, and the new method suggested by Ed Abraham in issue 1537721.)

9.14 contextlib

There is a new and slightly mind-blowing tool `ContextDecorator` that is helpful for creating a *context manager* that does double duty as a function decorator.

As a convenience, this new functionality is used by `contextmanager()` so that no extra effort is needed to support both roles.

The basic idea is that both context managers and function decorators can be used for pre-action and post-action wrappers. Context managers wrap a group of statements using a `with` statement, and function decorators wrap a group of statements enclosed in a function. So, occasionally there is a need to write a pre-action or post-action wrapper that can be used in either role.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, the `contextmanager()` provides both capabilities in a single definition:

```
from contextlib import contextmanager
import logging

logging.basicConfig(level=logging.INFO)

@contextmanager
def track_entry_and_exit(name):
    logging.info('Entering: {}'.format(name))
```

```
    yield
    logging.info('Exiting: {}'.format(name))
```

Formerly, this would have only been usable as a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Now, it can be used as a decorator as well:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Trying to fulfill two roles at once places some limitations on the technique. Context managers normally have the flexibility to return an argument usable by a `with` statement, but there is no parallel for function decorators.

In the above example, there is not a clean way for the *track_entry_and_exit* context manager to return a logging instance for use in the body of enclosed statements.

(Contributed by Michael Foord in issue 9110.)

9.15 decimal and fractions

Mark Dickinson crafted an elegant and efficient scheme for assuring that different numeric datatypes will have the same hash value whenever their actual values are equal (issue 8188):

```
assert hash(Fraction(3, 2)) == hash(1.5) == \
       hash(Decimal("1.5")) == hash(complex(1.5, 0))
```

Some of the hashing details are exposed through a new attribute, `sys.hash_info`, which describes the bit width of the hash value, the prime modulus, the hash values for *infinity* and *nan*, and the multiplier used for the imaginary part of a number:

```
>>> sys.hash_info
sys.hash_info(width=64, modulus=2305843009213693951, inf=314159, nan=0, imag=1000003)
```

An early decision to limit the inter-operability of various numeric types has been relaxed. It is still unsupported (and ill-advised) to have implicit mixing in arithmetic expressions such as `Decimal('1.1') + float('1.1')` because the latter loses information in the process of constructing the binary float. However, since existing floating point value can be converted losslessly to either a decimal or rational representation, it makes sense to add them to the constructor and to support mixed-type comparisons.

- The `decimal.Decimal` constructor now accepts `float` objects directly so there in no longer a need to use the `from_float()` method (issue 8257).
- Mixed type comparisons are now fully supported so that `Decimal` objects can be directly compared with `float` and `fractions.Fraction` (issue 2531 and issue 8188).

Similar changes were made to `fractions.Fraction` so that the `from_float()` and `from_decimal()` methods are no longer needed (issue 8294):

```
>>> Decimal(1.1)
Decimal('1.100000000000000088817841970012523233890533447265625')
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

Another useful change for the `decimal` module is that the `Context.clamp` attribute is now public. This is useful in creating contexts that correspond to the decimal interchange formats specified in IEEE 754 (see issue 8540).

(Contributed by Mark Dickinson and Raymond Hettinger.)

9.16 ftp

The `ftplib.FTP` class now supports the context manager protocol to unconditionally consume `socket.error` exceptions and to close the FTP connection when done:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
        ftp.login()
        ftp.dir()

'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
```

Other file-like objects such as `mmap.mmap` and `fileinput.input()` also grew auto-closing context managers:

```
with fileinput.input(files=('log1.txt', 'log2.txt')) as f:
    for line in f:
        process(line)
```

(Contributed by Tarek Ziadé and Giampaolo Rodolà in issue 4972, and by Georg Brandl in issue 8046 and issue 1286.)

The `FTP_TLS` class now accepts a *context* parameter, which is a `ssl.SSLContext` object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; issue 8806.)

9.17 popen

The `os.popen()` and `subprocess.Popen()` functions now support `with` statements for auto-closing of the file descriptors.

(Contributed by Antoine Pitrou and Brian Curtin in issue 7461 and issue 10554.)

9.18 select

The `select` module now exposes a new, constant attribute, `PIPE_BUF`, which gives the minimum number of bytes which are guaranteed not to block when `select.select()` says a pipe is ready for writing.

```
>>> import select
>>> select.PIPE_BUF
512
```

(Available on Unix systems. Patch by Sébastien Sablé in issue 9862)

9.19 gzip and zipfile

`gzip.GzipFile` now implements the `io.BufferedIOBase` *abstract base class* (except for `truncate()`). It also has a `peek()` method and supports unseekable as well as zero-padded file objects.

The `gzip` module also gains the `compress()` and `decompress()` functions for easier in-memory compression and decompression. Keep in mind that text needs to be encoded as `bytes` before compressing and decompressing:

```
>>> s = 'Three shall be the number thou shalt count, '
>>> s += 'and the number of the counting shall be three'
>>> b = s.encode()                        # convert to utf-8
>>> len(b)
89
>>> c = gzip.compress(b)
>>> len(c)
77
>>> gzip.decompress(c).decode()[:42]      # decompress and convert to text
'Three shall be the number thou shalt count,'
```

(Contributed by Anand B. Pillai in issue 3488; and by Antoine Pitrou, Nir Aides and Brian Curtin in issue 9962, issue 1675951, issue 7471 and issue 2846.)

Also, the `zipfile.ZipExtFile` class was reworked internally to represent files stored inside an archive. The new implementation is significantly faster and can be wrapped in a `io.BufferedReader` object for more speedups. It also solves an issue where interleaved calls to *read* and *readline* gave the wrong results.

(Patch submitted by Nir Aides in issue 7610.)

9.20 tarfile

The `TarFile` class can now be used as a context manager. In addition, its `add()` method has a new option, *filter*, that controls which files are added to the archive and allows the file metadata to be edited.

The new *filter* option replaces the older, less flexible *exclude* parameter which is now deprecated. If specified, the optional *filter* parameter needs to be a *keyword argument*. The user-supplied filter function accepts a `TarInfo` object and returns an updated `TarInfo` object, or if it wants the file to be excluded, the function can return *None*:

```
>>> import tarfile, glob

>>> def myfilter(tarinfo):
       if tarinfo.isfile():             # only save real files
            tarinfo.uname = 'monty'     # redact the user name
            return tarinfo

>>> with tarfile.open(name='myarchive.tar.gz', mode='w:gz') as tf:
        for filename in glob.glob('*.txt'):
            tf.add(filename, filter=myfilter)
        tf.list()
-rw-r--r-- monty/501        902 2011-01-26 17:59:11 annotations.txt
-rw-r--r-- monty/501        123 2011-01-26 17:59:11 general_questions.txt
-rw-r--r-- monty/501       3514 2011-01-26 17:59:11 prion.txt
-rw-r--r-- monty/501        124 2011-01-26 17:59:11 py_todo.txt
-rw-r--r-- monty/501       1399 2011-01-26 17:59:11 semaphore_notes.txt
```

(Proposed by Tarek Ziadé and implemented by Lars Gustäbel in issue 6856.)

9.21 hashlib

The `hashlib` module has two new constant attributes listing the hashing algorithms guaranteed to be present in all implementations and those available on the current implementation:

```
>>> import hashlib

>>> hashlib.algorithms_guaranteed
```

```
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}

>>> hashlib.algorithms_available
{'md2', 'SHA256', 'SHA512', 'dsaWithSHA', 'mdc2', 'SHA224', 'MD4', 'sha256',
'sha512', 'ripemd160', 'SHA1', 'MDC2', 'SHA', 'SHA384', 'MD2',
'ecdsa-with-SHA1','md4', 'md5', 'sha1', 'DSA-SHA', 'sha224',
'dsaEncryption', 'DSA', 'RIPEMD160', 'sha', 'MD5', 'sha384'}
```

(Suggested by Carl Chenet in issue 7418.)

9.22 ast

The `ast` module has a wonderful a general-purpose tool for safely evaluating expression strings using the Python literal syntax. The `ast.literal_eval()` function serves as a secure alternative to the builtin `eval()` function which is easily abused. Python 3.2 adds `bytes` and `set` literals to the list of supported types: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and None.

```
>>> from ast import literal_eval

>>> request = "{'req': 3, 'func': 'pow', 'args': (2, 0.5)}"
>>> literal_eval(request)
{'args': (2, 0.5), 'req': 3, 'func': 'pow'}

>>> request = "os.system('do something harmful')"
>>> literal_eval(request)
Traceback (most recent call last):
  ...
ValueError: malformed node or string: <_ast.Call object at 0x101739a10>
```

(Implemented by Benjamin Peterson and Georg Brandl.)

9.23 os

Different operating systems use various encodings for filenames and environment variables. The `os` module provides two new functions, `fsencode()` and `fsdecode()`, for encoding and decoding filenames:

```
>>> filename = 'Sehenswürdigkeiten'
>>> os.fsencode(filename)
b'Sehensw\xc3\xbcrdigkeiten'
```

Some operating systems allow direct access to encoded bytes in the environment. If so, the `os.supports_bytes_environ` constant will be true.

For direct access to encoded environment variables (if available), use the new `os.getenvb()` function or use `os.environb` which is a bytes version of `os.environ`.

(Contributed by Victor Stinner.)

9.24 shutil

The `shutil.copytree()` function has two new options:

- *ignore_dangling_symlinks*: when `symlinks=False` so that the function copies a file pointed to by a symlink, not the symlink itself. This option will silence the error raised if the file doesn't exist.
- *copy_function*: is a callable that will be used to copy files. `shutil.copy2()` is used by default.

(Contributed by Tarek Ziadé.)

In addition, the `shutil` module now supports *archiving operations* for zipfiles, uncompressed tarfiles, gzipped tarfiles, and bzipped tarfiles. And there are functions for registering additional archiving file formats (such as xz compressed tarfiles or custom formats).

The principal functions are `make_archive()` and `unpack_archive()`. By default, both operate on the current directory (which can be set by `os.chdir()`) and on any sub-directories. The archive filename needs to be specified with a full pathname. The archiving step is non-destructive (the original files are left unchanged).

```
>>> import shutil, pprint

>>> os.chdir('mydata')                                  # change to the source directory
>>> f = shutil.make_archive('/var/backup/mydata',
                            'zip')                      # archive the current directory
>>> f                                                   # show the name of archive
'/var/backup/mydata.zip'
>>> os.chdir('tmp')                                     # change to an unpacking
>>> shutil.unpack_archive('/var/backup/mydata.zip')     # recover the data

>>> pprint.pprint(shutil.get_archive_formats())        # display known formats
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]

>>> shutil.register_archive_format(                    # register a new archive format
        name = 'xz',
        function = xz.compress,                         # callable archiving function
        extra_args = [('level', 8)],                    # arguments to the function
        description = 'xz compression'
)
```

(Contributed by Tarek Ziadé.)

9.25 sqlite3

The `sqlite3` module was updated to pysqlite version 2.6.0. It has two new capabilities.

- The `sqlite3.Connection.in_transit` attribute is true if there is an active transaction for uncommitted changes.
- The `sqlite3.Connection.enable_load_extension()` and `sqlite3.Connection.load_extension()` methods allows you to load SQLite extensions from ".so" files. One well-known extension is the fulltext-search extension distributed with SQLite.

(Contributed by R. David Murray and Shashwat Anand; issue 8845.)

9.26 html

A new `html` module was introduced with only a single function, `escape()`, which is used for escaping reserved characters from HTML markup:

```
>>> import html
>>> html.escape('x > 2 && x < 7')
'x &gt; 2 &amp;&amp; x &lt; 7'
```

9.27 socket

The `socket` module has two new improvements.

- Socket objects now have a `detach()` method which puts the socket into closed state without actually closing the underlying file descriptor. The latter can then be reused for other purposes. (Added by Antoine Pitrou; issue 8524.)
- `socket.create_connection()` now supports the context manager protocol to unconditionally consume `socket.error` exceptions and to close the socket when done. (Contributed by Giampaolo Rodolà; issue 9794.)

9.28 ssl

The `ssl` module added a number of features to satisfy common requirements for secure (encrypted, authenticated) internet connections:

- A new class, `SSLContext`, serves as a container for persistent SSL data, such as protocol settings, certificates, private keys, and various other options. It includes a `wrap_socket()` for creating an SSL socket from an SSL context.
- A new function, `ssl.match_hostname()`, supports server identity verification for higher-level protocols by implementing the rules of HTTPS (from **RFC 2818**) which are also suitable for other protocols.
- The `ssl.wrap_socket()` constructor function now takes a *ciphers* argument. The *ciphers* string lists the allowed encryption algorithms using the format described in the OpenSSL documentation.
- When linked against recent versions of OpenSSL, the `ssl` module now supports the Server Name Indication extension to the TLS protocol, allowing multiple "virtual hosts" using different certificates on a single IP port. This extension is only supported in client mode, and is activated by passing the *server_hostname* argument to `ssl.SSLContext.wrap_socket()`.
- Various options have been added to the `ssl` module, such as `OP_NO_SSLv2` which disables the insecure and obsolete SSLv2 protocol.
- The extension now loads all the OpenSSL ciphers and digest algorithms. If some SSL certificates cannot be verified, they are reported as an "unknown algorithm" error.
- The version of OpenSSL being used is now accessible using the module attributes `ssl.OPENSSL_VERSION` (a string), `ssl.OPENSSL_VERSION_INFO` (a 5-tuple), and `ssl.OPENSSL_VERSION_NUMBER` (an integer).

(Contributed by Antoine Pitrou in issue 8850, issue 1589, issue 8322, issue 5639, issue 4870, issue 8484, and issue 8321.)

9.29 nntp

The `nntplib` module has a revamped implementation with better bytes and text semantics as well as more practical APIs. These improvements break compatibility with the nntplib version in Python 3.1, which was partly dysfunctional in itself.

Support for secure connections through both implicit (using `nntplib.NNTP_SSL`) and explicit (using `nntplib.NNTP.starttls()`) TLS has also been added.

(Contributed by Antoine Pitrou in issue 9360 and Andrew Vant in issue 1926.)

9.30 certificates

`http.client.HTTPSConnection`, `urllib.request.HTTPSHandler` and `urllib.request.urlopen()` now take optional arguments to allow for server certificate checking against a set of Certificate Authorities, as recommended in public uses of HTTPS.

(Added by Antoine Pitrou, issue 9003.)

9.31 imaplib

Support for explicit TLS on standard IMAP4 connections has been added through the new `imaplib.IMAP4.starttls` method.

(Contributed by Lorenzo M. Catucci and Antoine Pitrou, issue 4471.)

9.32 http.client

There were a number of small API improvements in the `http.client` module. The old-style HTTP 0.9 simple responses are no longer supported and the *strict* parameter is deprecated in all classes.

The `HTTPConnection` and `HTTPSConnection` classes now have a *source_address* parameter for a (host, port) tuple indicating where the HTTP connection is made from.

Support for certificate checking and HTTPS virtual hosts were added to `HTTPSConnection`.

The `request()` method on connection objects allowed an optional *body* argument so that a *file object* could be used to supply the content of the request. Conveniently, the *body* argument now also accepts an *iterable* object so long as it includes an explicit `Content-Length` header. This extended interface is much more flexible than before.

To establish an HTTPS connection through a proxy server, there is a new `set_tunnel()` method that sets the host and port for HTTP Connect tunneling.

To match the behavior of `http.server`, the HTTP client library now also encodes headers with ISO-8859-1 (Latin-1) encoding. It was already doing that for incoming headers, so now the behavior is consistent for both incoming and outgoing traffic. (See work by Armin Ronacher in issue 10980.)

9.33 unittest

The unittest module has a number of improvements supporting test discovery for packages, easier experimentation at the interactive prompt, new testcase methods, improved diagnostic messages for test failures, and better method names.

- The command-line call `python -m unittest` can now accept file paths instead of module names for running specific tests (issue 10620). The new test discovery can find tests within packages, locating any test importable from the top-level directory. The top-level directory can be specified with the *-t* option, a pattern for matching files with `-p`, and a directory to start discovery with `-s`:

  ```
  $ python -m unittest discover -s my_proj_dir -p _test.py
  ```

  (Contributed by Michael Foord.)

- Experimentation at the interactive prompt is now easier because the `unittest.case.TestCase` class can now be instantiated without arguments:

  ```
  >>> TestCase().assertEqual(pow(2, 3), 8)
  ```

  (Contributed by Michael Foord.)

- The `unittest` module has two new methods, `assertWarns()` and `assertWarnsRegex()` to verify that a given warning type is triggered by the code under test:

  ```
  with self.assertWarns(DeprecationWarning):
      legacy_function('XYZ')
  ```

  (Contributed by Antoine Pitrou, issue 9754.)

  Another new method, `assertCountEqual()` is used to compare two iterables to determine if their element counts are equal (whether the same elements are present with the same number of occurrences regardless of order):

  ```
  def test_anagram(self):
      self.assertCountEqual('algorithm', 'logarithm')
  ```

  (Contributed by Raymond Hettinger.)

- A principal feature of the unittest module is an effort to produce meaningful diagnostics when a test fails. When possible, the failure is recorded along with a diff of the output. This is especially helpful for analyzing log files of failed test runs. However, since diffs can sometime be voluminous, there is a new `maxDiff` attribute that sets maximum length of diffs displayed.

- In addition, the method names in the module have undergone a number of clean-ups.

  For example, `assertRegex()` is the new name for `assertRegexpMatches()` which was misnamed because the test uses `re.search()`, not `re.match()`. Other methods using regular expressions are now named using short form "Regex" in preference to "Regexp" – this matches the names used in other unittest implementations, matches Python's old name for the `re` module, and it has unambiguous camel-casing.

  (Contributed by Raymond Hettinger and implemented by Ezio Melotti.)

- To improve consistency, some long-standing method aliases are being deprecated in favor of the preferred names:

Old Name	Preferred Name
`assert_()`	`assertTrue()`
`assertEquals()`	`assertEqual()`
`assertNotEquals()`	`assertNotEqual()`
`assertAlmostEquals()`	`assertAlmostEqual()`
`assertNotAlmostEquals()`	`assertNotAlmostEqual()`

  Likewise, the `TestCase.fail*` methods deprecated in Python 3.1 are expected to be removed in Python 3.3. Also see the *deprecated-aliases* section in the `unittest` documentation.

  (Contributed by Ezio Melotti; issue 9424.)

- The `assertDictContainsSubset()` method was deprecated because it was misimplemented with the arguments in the wrong order. This created hard-to-debug optical illusions where tests like `TestCase().assertDictContainsSubset({'a':1, 'b':2}, {'a':1})` would fail.

  (Contributed by Raymond Hettinger.)

9.34 random

The integer methods in the `random` module now do a better job of producing uniform distributions. Previously, they computed selections with `int(n*random())` which had a slight bias whenever *n* was not a power of two. Now, multiple selections are made from a range up to the next power of two and a selection is kept only when it falls within the range `0 <= x < n`. The functions and methods affected are `randrange()`, `randint()`, `choice()`, `shuffle()` and `sample()`.

(Contributed by Raymond Hettinger; issue 9025.)

9.35 poplib

`POP3_SSL` class now accepts a *context* parameter, which is a `ssl.SSLContext` object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; issue 8807.)

9.36 asyncore

`asyncore.dispatcher` now provides a `handle_accepted()` method returning a *(sock, addr)* pair which is called when a connection has actually been established with a new remote endpoint. This is supposed to be used as a replacement for old `handle_accept()` and avoids the user to call `accept()` directly.

(Contributed by Giampaolo Rodolà; issue 6706.)

9.37 tempfile

The `tempfile` module has a new context manager, `TemporaryDirectory` which provides easy deterministic cleanup of temporary directories:

```
with tempfile.TemporaryDirectory() as tmpdirname:
    print('created temporary dir:', tmpdirname)
```

(Contributed by Neil Schemenauer and Nick Coghlan; issue 5178.)

9.38 inspect

- The `inspect` module has a new function `getgeneratorstate()` to easily identify the current state of a generator-iterator:

  ```
  >>> from inspect import getgeneratorstate
  >>> def gen():
          yield 'demo'
  >>> g = gen()
  >>> getgeneratorstate(g)
  'GEN_CREATED'
  >>> next(g)
  'demo'
  >>> getgeneratorstate(g)
  'GEN_SUSPENDED'
  >>> next(g, None)
  >>> getgeneratorstate(g)
  'GEN_CLOSED'
  ```

  (Contributed by Rodolpho Eckhardt and Nick Coghlan, issue 10220.)

- To support lookups without the possibility of activating a dynamic attribute, the `inspect` module has a new function, `getattr_static()`. Unlike `hasattr()`, this is a true read-only search, guaranteed not to change state while it is searching:

  ```
  >>> class A:
          @property
          def f(self):
              print('Running')
              return 10
  ```

```
>>> a = A()
>>> getattr(a, 'f')
Running
10
>>> inspect.getattr_static(a, 'f')
<property object at 0x1022bd788>
```

(Contributed by Michael Foord.)

9.39 pydoc

The `pydoc` module now provides a much-improved Web server interface, as well as a new command-line option `-b` to automatically open a browser window to display that server:

```
$ pydoc3.2 -b
```

(Contributed by Ron Adam; issue 2001.)

9.40 dis

The `dis` module gained two new functions for inspecting code, `code_info()` and `show_code()`. Both provide detailed code object information for the supplied function, method, source code string or code object. The former returns a string and the latter prints it:

```
>>> import dis, random
>>> dis.show_code(random.choice)
Name:              choice
Filename:          /Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/random.p
Argument count:    2
Kw-only arguments: 0
Number of locals:  3
Stack size:        11
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: 'Choose a random element from a non-empty sequence.'
   1: 'Cannot choose from an empty sequence'
Names:
   0: _randbelow
   1: len
   2: ValueError
   3: IndexError
Variable names:
   0: self
   1: seq
   2: i
```

In addition, the `dis()` function now accepts string arguments so that the common idiom `dis(compile(s, '', 'eval'))` can be shortened to `dis(s)`:

```
>>> dis('3*x+1 if x%2==1 else x//2')
  1           0 LOAD_NAME                0 (x)
              3 LOAD_CONST               0 (2)
              6 BINARY_MODULO
              7 LOAD_CONST               1 (1)
```

```
             10 COMPARE_OP               2 (==)
             13 POP_JUMP_IF_FALSE       28
             16 LOAD_CONST               2 (3)
             19 LOAD_NAME                0 (x)
             22 BINARY_MULTIPLY
             23 LOAD_CONST               1 (1)
             26 BINARY_ADD
             27 RETURN_VALUE
        >>   28 LOAD_NAME                0 (x)
             31 LOAD_CONST               0 (2)
             34 BINARY_FLOOR_DIVIDE
             35 RETURN_VALUE
```

Taken together, these improvements make it easier to explore how CPython is implemented and to see for yourself what the language syntax does under-the-hood.

(Contributed by Nick Coghlan in issue 9147.)

9.41 dbm

All database modules now support the `get()` and `setdefault()` methods.

(Suggested by Ray Allen in issue 9523.)

9.42 ctypes

A new type, `ctypes.c_ssize_t` represents the C `ssize_t` datatype.

9.43 site

The `site` module has three new functions useful for reporting on the details of a given Python installation.

- `getsitepackages()` lists all global site-packages directories.
- `getuserbase()` reports on the user's base directory where data can be stored.
- `getusersitepackages()` reveals the user-specific site-packages directory path.

```
>>> import site
>>> site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-packages',
 '/Library/Frameworks/Python.framework/Versions/3.2/lib/site-python',
 '/Library/Python/3.2/site-packages']
>>> site.getuserbase()
'/Users/raymondhettinger/Library/Python/3.2'
>>> site.getusersitepackages()
'/Users/raymondhettinger/Library/Python/3.2/lib/python/site-packages'
```

Conveniently, some of site's functionality is accessible directly from the command-line:

```
$ python -m site --user-base
/Users/raymondhettinger/.local
$ python -m site --user-site
/Users/raymondhettinger/.local/lib/python3.2/site-packages
```

(Contributed by Tarek Ziadé in issue 6693.)

9.44 sysconfig

The new `sysconfig` module makes it straightforward to discover installation paths and configuration variables that vary across platforms and installations.

The module offers access simple access functions for platform and version information:

- `get_platform()` returning values like *linux-i586* or *macosx-10.6-ppc*.
- `get_python_version()` returns a Python version string such as "3.2".

It also provides access to the paths and variables corresponding to one of seven named schemes used by `distutils`. Those include *posix_prefix*, *posix_home*, *posix_user*, *nt*, *nt_user*, *os2*, *os2_home*:

- `get_paths()` makes a dictionary containing installation paths for the current installation scheme.
- `get_config_vars()` returns a dictionary of platform specific variables.

There is also a convenient command-line interface:

```
C:\Python32>python -m sysconfig
Platform: "win32"
Python version: "3.2"
Current installation scheme: "nt"

Paths:
        data = "C:\Python32"
        include = "C:\Python32\Include"
        platinclude = "C:\Python32\Include"
        platlib = "C:\Python32\Lib\site-packages"
        platstdlib = "C:\Python32\Lib"
        purelib = "C:\Python32\Lib\site-packages"
        scripts = "C:\Python32\Scripts"
        stdlib = "C:\Python32\Lib"

Variables:
        BINDIR = "C:\Python32"
        BINLIBDEST = "C:\Python32\Lib"
        EXE = ".exe"
        INCLUDEPY = "C:\Python32\Include"
        LIBDEST = "C:\Python32\Lib"
        SO = ".pyd"
        VERSION = "32"
        abiflags = ""
        base = "C:\Python32"
        exec_prefix = "C:\Python32"
        platbase = "C:\Python32"
        prefix = "C:\Python32"
        projectbase = "C:\Python32"
        py_version = "3.2"
        py_version_nodot = "32"
        py_version_short = "3.2"
        srcdir = "C:\Python32"
        userbase = "C:\Documents and Settings\Raymond\Application Data\Python"
```

(Moved out of Distutils by Tarek Ziadé.)

9.45 pdb

The pdb debugger module gained a number of usability improvements:

- pdb.py now has a -c option that executes commands as given in a .pdbrc script file.
- A .pdbrc script file can contain continue and next commands that continue debugging.
- The Pdb class constructor now accepts a *nosigint* argument.
- New commands: l(list), ll(long list) and source for listing source code.
- New commands: display and undisplay for showing or hiding the value of an expression if it has changed.
- New command: interact for starting an interactive interpreter containing the global and local names found in the current scope.
- Breakpoints can be cleared by breakpoint number.

(Contributed by Georg Brandl, Antonio Cuni and Ilya Sandler.)

9.46 configparser

The configparser module was modified to improve usability and predictability of the default parser and its supported INI syntax. The old ConfigParser class was removed in favor of SafeConfigParser which has in turn been renamed to ConfigParser. Support for inline comments is now turned off by default and section or option duplicates are not allowed in a single configuration source.

Config parsers gained a new API based on the mapping protocol:

```
>>> parser = ConfigParser()
>>> parser.read_string("""
[DEFAULT]
location = upper left
visible = yes
editable = no
color = blue

[main]
title = Main Menu
color = green

[options]
title = Options
""")
>>> parser['main']['color']
'green'
>>> parser['main']['editable']
'no'
>>> section = parser['options']
>>> section['title']
'Options'
>>> section['title'] = 'Options (editable: %(editable)s)'
>>> section['title']
'Options (editable: no)'
```

The new API is implemented on top of the classical API, so custom parser subclasses should be able to use it without modifications.

The INI file structure accepted by config parsers can now be customized. Users can specify alternative option/value delimiters and comment prefixes, change the name of the *DEFAULT* section or switch the interpolation syntax.

There is support for pluggable interpolation including an additional interpolation handler `ExtendedInterpolation`:

```
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> parser.read_dict({'buildout': {'directory': '/home/ambv/zope9'},
                      'custom': {'prefix': '/usr/local'}})
>>> parser.read_string("""
    [buildout]
    parts =
      zope9
      instance
    find-links =
      ${buildout:directory}/downloads/dist

    [zope9]
    recipe = plone.recipe.zope9install
    location = /opt/zope

    [instance]
    recipe = plone.recipe.zope9instance
    zope9-location = ${zope9:location}
    zope-conf = ${custom:prefix}/etc/zope.conf
    """)
>>> parser['buildout']['find-links']
'\n/home/ambv/zope9/downloads/dist'
>>> parser['instance']['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance = parser['instance']
>>> instance['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance['zope9-location']
'/opt/zope'
```

A number of smaller features were also introduced, like support for specifying encoding in read operations, specifying fallback values for get-functions, or reading directly from dictionaries and strings.

(All changes contributed by Łukasz Langa.)

9.47 urllib.parse

A number of usability improvements were made for the `urllib.parse` module.

The `urlparse()` function now supports IPv6 addresses as described in **RFC 2732**:

```
>>> import urllib.parse
>>> urllib.parse.urlparse('http://[dead:beef:cafe:5417:affe:8FA3:deaf:feed]/foo/')
ParseResult(scheme='http',
            netloc='[dead:beef:cafe:5417:affe:8FA3:deaf:feed]',
            path='/foo/',
            params='',
            query='',
            fragment='')
```

The `urldefrag()` function now returns a *named tuple*:

```
>>> r = urllib.parse.urldefrag('http://python.org/about/#target')
>>> r
DefragResult(url='http://python.org/about/', fragment='target')
>>> r[0]
'http://python.org/about/'
>>> r.fragment
'target'
```

And, the `urlencode()` function is now much more flexible, accepting either a string or bytes type for the *query* argument. If it is a string, then the *safe*, *encoding*, and *error* parameters are sent to `quote_plus()` for encoding:

```
>>> urllib.parse.urlencode([
         ('type', 'telenovela'),
         ('name', '¿Dónde Está Elisa?')],
         encoding='latin-1')
'type=telenovela&name=%BFD%F3nde+Est%E1+Elisa%3F'
```

As detailed in *parsing-ascii-encoded-bytes*, all the `urllib.parse` functions now accept ASCII-encoded byte strings as input, so long as they are not mixed with regular strings. If ASCII-encoded byte strings are given as parameters, the return types will also be an ASCII-encoded byte strings:

```
>>> urllib.parse.urlparse(b'http://www.python.org:80/about/')
ParseResultBytes(scheme=b'http', netloc=b'www.python.org:80',
                 path=b'/about/', params=b'', query=b'', fragment=b'')
```

(Work by Nick Coghlan, Dan Mahn, and Senthil Kumaran in issue 2987, issue 5468, and issue 9873.)

9.48 mailbox

Thanks to a concerted effort by R. David Murray, the `mailbox` module has been fixed for Python 3.2. The challenge was that mailbox had been originally designed with a text interface, but email messages are best represented with `bytes` because various parts of a message may have different encodings.

The solution harnessed the `email` package's binary support for parsing arbitrary email messages. In addition, the solution required a number of API changes.

As expected, the `add()` method for `mailbox.Mailbox` objects now accepts binary input.

`StringIO` and text file input are deprecated. Also, string input will fail early if non-ASCII characters are used. Previously it would fail when the email was processed in a later step.

There is also support for binary output. The `get_file()` method now returns a file in the binary mode (where it used to incorrectly set the file to text-mode). There is also a new `get_bytes()` method that returns a `bytes` representation of a message corresponding to a given *key*.

It is still possible to get non-binary output using the old API's `get_string()` method, but that approach is not very useful. Instead, it is best to extract messages from a `Message` object or to load them from binary input.

(Contributed by R. David Murray, with efforts from Steffen Daode Nurpmeso and an initial patch by Victor Stinner in issue 9124.)

9.49 turtledemo

The demonstration code for the `turtle` module was moved from the *Demo* directory to main library. It includes over a dozen sample scripts with lively displays. Being on `sys.path`, it can now be run directly from the command-line:

```
$ python -m turtledemo
```

(Moved from the Demo directory by Alexander Belopolsky in issue 10199.)

10 Multi-threading

- The mechanism for serializing execution of concurrently running Python threads (generally known as the *GIL* or *Global Interpreter Lock*) has been rewritten. Among the objectives were more predictable switching intervals and reduced overhead due to lock contention and the number of ensuing system calls. The notion of a "check interval" to allow thread switches has been abandoned and replaced by an absolute duration expressed in seconds. This parameter is tunable through `sys.setswitchinterval()`. It currently defaults to 5 milliseconds.

  Additional details about the implementation can be read from a python-dev mailing-list message (however, "priority requests" as exposed in this message have not been kept for inclusion).

  (Contributed by Antoine Pitrou.)

- Regular and recursive locks now accept an optional *timeout* argument to their `acquire()` method. (Contributed by Antoine Pitrou; issue 7316.)

- Similarly, `threading.Semaphore.acquire()` also gained a *timeout* argument. (Contributed by Torsten Landschoff; issue 850728.)

- Regular and recursive lock acquisitions can now be interrupted by signals on platforms using Pthreads. This means that Python programs that deadlock while acquiring locks can be successfully killed by repeatedly sending SIGINT to the process (by pressing `Ctrl+C` in most shells). (Contributed by Reid Kleckner; issue 8844.)

11 Optimizations

A number of small performance enhancements have been added:

- Python's peephole optimizer now recognizes patterns such `x in {1, 2, 3}` as being a test for membership in a set of constants. The optimizer recasts the `set` as a `frozenset` and stores the pre-built constant.

  Now that the speed penalty is gone, it is practical to start writing membership tests using set-notation. This style is both semantically clear and operationally fast:

  ```
  extension = name.rpartition('.')[2]
  if extension in {'xml', 'html', 'xhtml', 'css'}:
      handle(name)
  ```

  (Patch and additional tests contributed by Dave Malcolm; issue 6690).

- Serializing and unserializing data using the `pickle` module is now several times faster.

  (Contributed by Alexandre Vassalotti, Antoine Pitrou and the Unladen Swallow team in issue 9410 and issue 3873.)

- The Timsort algorithm used in `list.sort()` and `sorted()` now runs faster and uses less memory when called with a *key function*. Previously, every element of a list was wrapped with a temporary object that remembered the key value associated with each element. Now, two arrays of keys and values are sorted in parallel. This saves the memory consumed by the sort wrappers, and it saves time lost to delegating comparisons.

  (Patch by Daniel Stutzbach in issue 9915.)

- JSON decoding performance is improved and memory consumption is reduced whenever the same string is repeated for multiple keys. Also, JSON encoding now uses the C speedups when the `sort_keys` argument is true.

  (Contributed by Antoine Pitrou in issue 7451 and by Raymond Hettinger and Antoine Pitrou in issue 10314.)

- Recursive locks (created with the `threading.RLock()` API) now benefit from a C implementation which makes them as fast as regular locks, and between 10x and 15x faster than their previous pure Python implementation.

(Contributed by Antoine Pitrou; issue 3001.)

- The fast-search algorithm in stringlib is now used by the `split()`, `splitlines()` and `replace()` methods on `bytes`, `bytearray` and `str` objects. Likewise, the algorithm is also used by `rfind()`, `rindex()`, `rsplit()` and `rpartition()`.

  (Patch by Florent Xicluna in issue 7622 and issue 7462.)

- String to integer conversions now work two "digits" at a time, reducing the number of division and modulo operations.

  (issue 6713 by Gawain Bolton, Mark Dickinson, and Victor Stinner.)

There were several other minor optimizations. Set differencing now runs faster when one operand is much larger than the other (patch by Andress Bennetts in issue 8685). The `array.repeat()` method has a faster implementation (issue 1569291 by Alexander Belopolsky). The `BaseHTTPRequestHandler` has more efficient buffering (issue 3709 by Andrew Schaaf). The `operator.attrgetter()` function has been sped-up (issue 10160 by Christos Georgiou). And `ConfigParser` loads multi-line arguments a bit faster (issue 7113 by Łukasz Langa).

12 Unicode

Python has been updated to Unicode 6.0.0. The update to the standard adds over 2,000 new characters including emoji symbols which are important for mobile phones.

In addition, the updated standard has altered the character properties for two Kannada characters (U+0CF1, U+0CF2) and one New Tai Lue numeric character (U+19DA), making the former eligible for use in identifiers while disqualifying the latter. For more information, see Unicode Character Database Changes.

13 Codecs

Support was added for *cp720* Arabic DOS encoding (issue 1616979).

MBCS encoding no longer ignores the error handler argument. In the default strict mode, it raises an `UnicodeDecodeError` when it encounters an undecodable byte sequence and an `UnicodeEncodeError` for an unencodable character.

The MBCS codec supports `'strict'` and `'ignore'` error handlers for decoding, and `'strict'` and `'replace'` for encoding.

To emulate Python3.1 MBCS encoding, select the `'ignore'` handler for decoding and the `'replace'` handler for encoding.

On Mac OS X, Python decodes command line arguments with `'utf-8'` rather than the locale encoding.

By default, `tarfile` uses `'utf-8'` encoding on Windows (instead of `'mbcs'`) and the `'surrogateescape'` error handler on all operating systems.

14 Documentation

The documentation continues to be improved.

- A table of quick links has been added to the top of lengthy sections such as *built-in-funcs*. In the case of `itertools`, the links are accompanied by tables of cheatsheet-style summaries to provide an overview and memory jog without having to read all of the docs.

- In some cases, the pure Python source code can be a helpful adjunct to the documentation, so now many modules now feature quick links to the latest version of the source code. For example, the `functools` module documentation has a quick link at the top labeled:

  Source code Lib/functools.py.

  (Contributed by Raymond Hettinger; see rationale.)

- The docs now contain more examples and recipes. In particular, `re` module has an extensive section, *re-examples*. Likewise, the `itertools` module continues to be updated with new *itertools-recipes*.

- The `datetime` module now has an auxiliary implementation in pure Python. No functionality was changed. This just provides an easier-to-read alternate implementation.

  (Contributed by Alexander Belopolsky in issue 9528.)

- The unmaintained `Demo` directory has been removed. Some demos were integrated into the documentation, some were moved to the `Tools/demo` directory, and others were removed altogether.

  (Contributed by Georg Brandl in issue 7962.)

15 IDLE

- The format menu now has an option to clean source files by stripping trailing whitespace.

  (Contributed by Raymond Hettinger; issue 5150.)

- IDLE on Mac OS X now works with both Carbon AquaTk and Cocoa AquaTk.

  (Contributed by Kevin Walzer, Ned Deily, and Ronald Oussoren; issue 6075.)

16 Code Repository

In addition to the existing Subversion code repository at http://svn.python.org there is now a Mercurial repository at http://hg.python.org/.

After the 3.2 release, there are plans to switch to Mercurial as the primary repository. This distributed version control system should make it easier for members of the community to create and share external changesets. See

PEP 385 for details.

To learn the new version control system, see the tutorial by Joel Spolsky or the Guide to Mercurial Workflows.

17 Build and C API Changes

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

- The *idle*, *pydoc* and *2to3* scripts are now installed with a version-specific suffix on `make altinstall` (issue 10679).

- The C functions that access the Unicode Database now accept and return characters from the full Unicode range, even on narrow unicode builds (Py_UNICODE_TOLOWER, Py_UNICODE_ISDECIMAL, and others). A visible difference in Python is that `unicodedata.numeric()` now returns the correct value for large code points, and `repr()` may consider more characters as printable.

  (Reported by Bupjoe Lee and fixed by Amaury Forgeot D'Arc; issue 5127.)

- Computed gotos are now enabled by default on supported compilers (which are detected by the configure script). They can still be disabled selectively by specifying `--without-computed-gotos`.

  (Contributed by Antoine Pitrou; issue 9203.)

- The option `--with-wctype-functions` was removed. The built-in unicode database is now used for all functions.

  (Contributed by Amaury Forgeot D'Arc; issue 9210.)

- Hash values are now values of a new type, `Py_hash_t`, which is defined to be the same size as a pointer. Previously they were of type long, which on some 64-bit operating systems is still only 32 bits long. As a result of this fix, `set` and `dict` can now hold more than `2**32` entries on builds with 64-bit pointers (previously, they could grow to that size but their performance degraded catastrophically).

  (Suggested by Raymond Hettinger and implemented by Benjamin Peterson; issue 9778.)

- A new macro `Py_VA_COPY` copies the state of the variable argument list. It is equivalent to C99 *va_copy* but available on all Python platforms (issue 2443).

- A new C API function `PySys_SetArgvEx()` allows an embedded interpreter to set `sys.argv` without also modifying `sys.path` (issue 5753).

- `PyEval_CallObject` is now only available in macro form. The function declaration, which was kept for backwards compatibility reasons, is now removed – the macro was introduced in 1997 (issue 8276).

- There is a new function `PyLong_AsLongLongAndOverflow()` which is analogous to `PyLong_AsLongAndOverflow()`. They both serve to convert Python `int` into a native fixed-width type while providing detection of cases where the conversion won't fit (issue 7767).

- The `PyUnicode_CompareWithASCIIString()` function now returns *not equal* if the Python string is *NUL* terminated.

- There is a new function `PyErr_NewExceptionWithDoc()` that is like `PyErr_NewException()` but allows a docstring to be specified. This lets C exceptions have the same self-documenting capabilities as their pure Python counterparts (issue 7033).

- When compiled with the `--with-valgrind` option, the pymalloc allocator will be automatically disabled when running under Valgrind. This gives improved memory leak detection when running under Valgrind, while taking advantage of pymalloc at other times (issue 2422).

- Removed the `O?` format from the *PyArg_Parse* functions. The format is no longer used and it had never been documented (issue 8837).

There were a number of other small changes to the C-API. See the Misc/NEWS file for a complete list.

Also, there were a number of updates to the Mac OS X build, see Mac/BuildScript/README.txt for details. For users running a 32/64-bit build, there is a known problem with the default Tcl/Tk on Mac OS X 10.6. Accordingly, we recommend installing an updated alternative such as ActiveState Tcl/Tk 8.5.9. See http://www.python.org/download/mac/tcltk/ for additional details.

18 Porting to Python 3.2

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

- The `configparser` module has a number of clean-ups. The major change is to replace the old `ConfigParser` class with long-standing preferred alternative `SafeConfigParser`. In addition there are a number of smaller incompatibilities:

  - The interpolation syntax is now validated on `get()` and `set()` operations. In the default interpolation scheme, only two tokens with percent signs are valid: `%(name)s` and `%%`, the latter being an escaped percent sign.
  - The `set()` and `add_section()` methods now verify that values are actual strings. Formerly, unsupported types could be introduced unintentionally.
  - Duplicate sections or options from a single source now raise either `DuplicateSectionError` or `DuplicateOptionError`. Formerly, duplicates would silently overwrite a previous entry.
  - Inline comments are now disabled by default so now the **;** character can be safely used in values.
  - Comments now can be indented. Consequently, for **;** or **#** to appear at the start of a line in multiline values, it has to be interpolated. This keeps comment prefix characters in values from being mistaken as comments.
  - `""` is now a valid value and is no longer automatically converted to an empty string. For empty strings, use `"option ="` in a line.

- The `nntplib` module was reworked extensively, meaning that its APIs are often incompatible with the 3.1 APIs.
- `bytearray` objects can no longer be used as filenames; instead, they should be converted to `bytes`.
- The `array.tostring()` and `array.fromstring()` have been renamed to `array.tobytes()` and `array.frombytes()` for clarity. The old names have been deprecated. (See issue 8990.)
- `PyArg_Parse*()` functions:
  - "t#" format has been removed: use "s#" or "s*" instead
  - "w" and "w#" formats has been removed: use "w*" instead
- The `PyCObject` type, deprecated in 3.1, has been removed. To wrap opaque C pointers in Python objects, the `PyCapsule` API should be used instead; the new type has a well-defined interface for passing typing safety information and a less complicated signature for calling a destructor.
- The `sys.setfilesystemencoding()` function was removed because it had a flawed design.
- The `random.seed()` function and method now salt string seeds with an sha512 hash function. To access the previous version of *seed* in order to reproduce Python 3.1 sequences, set the *version* argument to *1*, `random.seed(s, version=1)`.
- The previously deprecated `string.maketrans()` function has been removed in favor of the static methods `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the `string` module. Now, `str`, `bytes`, and `bytearray` each have their own **maketrans** and **translate** methods with intermediate translation tables of the appropriate type.

  (Contributed by Georg Brandl; issue 5675.)
- The previously deprecated `contextlib.nested()` function has been removed in favor of a plain `with` statement which can accept multiple context managers. The latter technique is faster (because it is built-in), and it does a better job finalizing multiple context managers when one of them raises an exception:

```
with open('mylog.txt') as infile, open('a.out', 'w') as outfile:
    for line in infile:
        if '<critical>' in line:
            outfile.write(line)
```

  (Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094.)
- `struct.pack()` now only allows bytes for the `s` string pack code. Formerly, it would accept text arguments and implicitly encode them to bytes using UTF-8. This was problematic because it made assumptions about

the correct encoding and because a variable-length encoding can fail when writing to fixed length segment of a structure.

Code such as `struct.pack('<6sHHBBB', 'GIF87a', x, y)` should be rewritten with to use bytes instead of text, `struct.pack('<6sHHBBB', b'GIF87a', x, y)`.

(Discovered by David Beazley and fixed by Victor Stinner; issue 10783.)

- The `xml.etree.ElementTree` class now raises an `xml.etree.ElementTree.ParseError` when a parse fails. Previously it raised a `xml.parsers.expat.ExpatError`.
- The new, longer `str()` value on floats may break doctests which rely on the old output format.
- In `subprocess.Popen`, the default value for *close_fds* is now `True` under Unix; under Windows, it is `True` if the three standard streams are set to `None`, `False` otherwise. Previously, *close_fds* was always `False` by default, which produced difficult to solve bugs or race conditions when open file descriptors would leak into the child process.
- Support for legacy HTTP 0.9 has been removed from `urllib.request` and `http.client`. Such support is still present on the server side (in `http.server`).

  (Contributed by Antoine Pitrou, issue 10711.)
- SSL sockets in timeout mode now raise `socket.timeout` when a timeout occurs, rather than a generic `SSLError`.

  (Contributed by Antoine Pitrou, issue 10272.)
- The misleading functions `PyEval_AcquireLock()` and `PyEval_ReleaseLock()` have been officially deprecated. The thread-state aware APIs (such as `PyEval_SaveThread()` and `PyEval_RestoreThread()`) should be used instead.
- Due to security risks, `asyncore.handle_accept()` has been deprecated, and a new function, `asyncore.handle_accepted()`, was added to replace it.

  (Contributed by Giampaolo Rodola in issue 6706.)
- Due to the new *GIL* implementation, `PyEval_InitThreads()` cannot be called before `Py_Initialize()` anymore.

Index