What's New in Python

Release 3.3.0

A. M. Kuchling

September 29, 2012

Python Software Foundation
Email: docs@python.org

Contents

Author Raymond Hettinger

Release 3.3.0

Date September 29, 2012

This article explains the new features in Python 3.3, compared to 3.2. Python 3.3 was released on September 29, 2012.

1 Summary – Release highlights

New syntax features:

- New `yield from` expression for *generator delegation*.
- The `u'unicode'` syntax is accepted again for `str` objects.

New library modules:

- `faulthandler` (helps debugging low-level crashes)
- `ipaddress` (high-level objects representing IP addresses and masks)
- `lzma` (compress data using the XZ / LZMA algorithm)
- `venv` (Python *virtual environments*, as in the popular `virtualenv` package)

New built-in features:

- Reworked *I/O exception hierarchy*.

Implementation improvements:

- Rewritten *import machinery* based on `importlib`.
- More compact *unicode strings*.
- More compact *attribute dictionaries*.

Security improvements:

- Hash randomization is switched on by default.

Please read on for a comprehensive list of user-facing changes.

2 PEP 405: Virtual Environments

PEP 405 - Python Virtual Environments PEP written by Carl Meyer, implemented by Carl Meyer and Vinay Sajip.

Virtual environments help create separate Python setups while sharing a system-wide base install, for ease of maintenance. Virtual environments have their own set of private site packages (i.e. locally-installed libraries), and are optionally segregated from the system-wide site packages. Their concept and implementation are inspired by the popular `virtualenv` third-party package, but benefit from tighter integration with the interpreter core.

This PEP adds the `venv` module for programmatic access, and the *pyvenv* script for command-line access and administration. The Python interpreter becomes aware of a `pvenv.cfg` file whose existence signals the base of a virtual environment's directory tree.

3 PEP 420: Namespace Packages

Native support for package directories that don't require `__init__.py` marker files and can automatically span multiple path segments (inspired by various third party approaches to namespace packages, as described in

PEP 420)

4 PEP 3118: New memoryview implementation and buffer protocol documentation

issue 10181 **- memoryview bug fixes and features.** Written by Stefan Krah.

The new memoryview implementation comprehensively fixes all ownership and lifetime issues of dynamically allocated fields in the Py_buffer struct that led to multiple crash reports. Additionally, several functions that crashed or returned incorrect results for non-contiguous or multi-dimensional input have been fixed.

The memoryview object now has a PEP-3118 compliant getbufferproc() that checks the consumer's request type. Many new features have been added, most of them work in full generality for non-contiguous arrays and arrays with suboffsets.

The documentation has been updated, clearly spelling out responsibilities for both exporters and consumers. Buffer request flags are grouped into basic and compound flags. The memory layout of non-contiguous and multi-dimensional NumPy-style arrays is explained.

4.1 Features

- All native single character format specifiers in struct module syntax (optionally prefixed with '@') are now supported.
- With some restrictions, the cast() method allows changing of format and shape of C-contiguous arrays.
- Multi-dimensional list representations are supported for any array type.
- Multi-dimensional comparisons are supported for any array type.
- One-dimensional memoryviews of hashable (read-only) types with formats B, b or c are now hashable. (Contributed by Antoine Pitrou in issue 13411)
- Arbitrary slicing of any 1-D arrays type is supported. For example, it is now possible to reverse a memoryview in O(1) by using a negative step.

4.2 API changes

- The maximum number of dimensions is officially limited to 64.
- The representation of empty shape, strides and suboffsets is now an empty tuple instead of None.
- Accessing a memoryview element with format 'B' (unsigned bytes) now returns an integer (in accordance with the struct module syntax). For returning a bytes object the view must be cast to 'c' first.
- memoryview comparisons now use the logical structure of the operands and compare all array elements by value. All format strings in struct module syntax are supported. Views with unrecognised format strings are still permitted, but will always compare as unequal, regardless of view contents.
- For further changes see Build and C API Changes and Porting C code .

5 PEP 393: Flexible String Representation

The Unicode string type is changed to support multiple internal representations, depending on the character with the largest Unicode ordinal (1, 2, or 4 bytes) in the represented string. This allows a space-efficient representation in common cases, but gives access to full UCS-4 on all systems. For compatibility with existing APIs, several representations may exist in parallel; over time, this compatibility should be phased out.

On the Python side, there should be no downside to this change.

On the C API side, PEP 393 is fully backward compatible. The legacy API should remain available at least five years. Applications using the legacy API will not fully benefit of the memory reduction, or - worse - may use a bit more memory, because Python may have to maintain two versions of each string (in the legacy format and in the new efficient storage).

5.1 Functionality

Changes introduced by **PEP 393** are the following:

- Python now always supports the full range of Unicode codepoints, including non-BMP ones (i.e. from `U+0000` to `U+10FFFF`). The distinction between narrow and wide builds no longer exists and Python now behaves like a wide build, even under Windows.
- With the death of narrow builds, the problems specific to narrow builds have also been fixed, for example:
  - `len()` now always returns 1 for non-BMP characters, so `len('\U0010FFFF') == 1`;
  - surrogate pairs are not recombined in string literals, so `'\uDBFF\uDFFF' != '\U0010FFFF'`;
  - indexing or slicing non-BMP characters returns the expected value, so `'\U0010FFFF'[0]` now returns `'\U0010FFFF'` and not `'\uDBFF'`;
  - all other functions in the standard library now correctly handle non-BMP codepoints.
- The value of `sys.maxunicode` is now always `1114111` (`0x10FFFF` in hexadecimal). The `PyUnicode_GetMax()` function still returns either `0xFFFF` or `0x10FFFF` for backward compatibility, and it should not be used with the new Unicode API (see issue 13054).
- The `./configure` flag `--with-wide-unicode` has been removed.

5.2 Performance and resource usage

The storage of Unicode strings now depends on the highest codepoint in the string:

- pure ASCII and Latin1 strings (`U+0000-U+00FF`) use 1 byte per codepoint;
- BMP strings (`U+0000-U+FFFF`) use 2 bytes per codepoint;
- non-BMP strings (`U+10000-U+10FFFF`) use 4 bytes per codepoint.

The net effect is that for most applications, memory usage of string storage should decrease significantly - especially compared to former wide unicode builds - as, in many cases, strings will be pure ASCII even in international contexts (because many strings store non-human language data, such as XML fragments, HTTP headers, JSON-encoded data, etc.). We also hope that it will, for the same reasons, increase CPU cache efficiency on non-trivial applications. The memory usage of Python 3.3 is two to three times smaller than Python 3.2, and a little bit better than Python 2.7, on a Django benchmark (see the PEP for details).

6 PEP 3151: Reworking the OS and IO exception hierarchy

PEP 3151 - Reworking the OS and IO exception hierarchy PEP written and implemented by Antoine Pitrou.

The hierarchy of exceptions raised by operating system errors is now both simplified and finer-grained.

You don't have to worry anymore about choosing the appropriate exception type between `OSError`, `IOError`, `EnvironmentError`, `WindowsError`, `mmap.error`, `socket.error` or `select.error`. All these exception types are now only one: `OSError`. The other names are kept as aliases for compatibility reasons.

Also, it is now easier to catch a specific error condition. Instead of inspecting the `errno` attribute (or `args[0]`) for a particular constant from the `errno` module, you can catch the adequate `OSError` subclass. The available subclasses are the following:

- `BlockingIOError`
- `ChildProcessError`
- `ConnectionError`
- `FileExistsError`
- `FileNotFoundError`
- `InterruptedError`
- `IsADirectoryError`
- `NotADirectoryError`
- `PermissionError`
- `ProcessLookupError`
- `TimeoutError`

And the `ConnectionError` itself has finer-grained subclasses:

- `BrokenPipeError`
- `ConnectionAbortedError`
- `ConnectionRefusedError`
- `ConnectionResetError`

Thanks to the new exceptions, common usages of the `errno` can now be avoided. For example, the following code written for Python 3.2:

```
from errno import ENOENT, EACCES, EPERM

try:
    with open("document.txt") as f:
        content = f.read()
except IOError as err:
    if err.errno == ENOENT:
        print("document.txt file is missing")
    elif err.errno in (EACCES, EPERM):
        print("You are not allowed to read document.txt")
    else:
        raise
```

can now be written without the `errno` import and without manual inspection of exception attributes:

```
try:
    with open("document.txt") as f:
        content = f.read()
except FileNotFoundError:
    print("document.txt file is missing")
except PermissionError:
    print("You are not allowed to read document.txt")
```

7 PEP 380: Syntax for Delegating to a Subgenerator

PEP 380 - Syntax for Delegating to a Subgenerator PEP written by Greg Ewing.

PEP 380 adds the `yield from` expression, allowing a generator to delegate part of its operations to another generator. This allows a section of code containing 'yield' to be factored out and placed in another generator. Additionally, the subgenerator is allowed to return with a value, and the value is made available to the delegating generator.

While designed primarily for use in delegating to a subgenerator, the `yield from` expression actually allows delegation to arbitrary subiterators.

For simple iterators, `yield from iterable` is essentially just a shortened form of `for item in iterable: yield item`:

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

However, unlike an ordinary loop, `yield from` allows subgenerators to receive sent and thrown values directly from the calling scope, and return a final value to the outer generator:

```
>>> def accumulate(start=0):
...     tally = start
...     while 1:
...         next = yield
...         if next is None:
...             return tally
...         tally += next
...
>>> def gather_tallies(tallies, start=0):
...     while 1:
...         tally = yield from accumulate()
...         tallies.append(tally)
...
>>> tallies = []
>>> acc = gather_tallies(tallies)
>>> next(acc) # Ensure the accumulator is ready to accept values
>>> for i in range(10):
...     acc.send(i)
...
>>> acc.send(None) # Finish the first tally
>>> for i in range(5):
...     acc.send(i)
...
>>> acc.send(None) # Finish the second tally
>>> tallies
[45, 10]
```

The main principle driving this change is to allow even generators that are designed to be used with the `send` and `throw` methods to be split into multiple subgenerators as easily as a single large function can be split into multiple subfunctions.

(Implementation by Greg Ewing, integrated into 3.3 by Renaud Blanch, Ryan Kelly and Nick Coghlan, documentation by Zbigniew Jędrzejewski-Szmek and Nick Coghlan)

8 PEP 409: Suppressing exception context

PEP 409 - Suppressing exception context PEP written by Ethan Furman, implemented by Ethan Furman and Nick Coghlan.

PEP 409 introduces new syntax that allows the display of the chained exception context to be disabled. This allows cleaner error messages in applications that convert between exception types:

```
>>> class D:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr) from None...
>>>
D({}).x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

Without the `from None` suffix to suppress the cause, the original exception would be displayed by default:

```
>>> class C:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr)
...
>>> C({}).x
Traceback (most recent call last):
  File "<stdin>", line 6, in __getattr__
KeyError: 'x'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

No debugging capability is lost, as the original exception context remains available if needed (for example, if an intervening library has incorrectly suppressed valuable underlying details):

```
>>> try:
...     D({}).x
... except AttributeError as exc:
...     print(repr(exc.__context__))
...
KeyError('x',)
```

9 PEP 414: Explicit Unicode literals

PEP 414 - Explicit Unicode literals PEP written by Armin Ronacher.

To ease the transition from Python 2 for Unicode aware Python applications that make heavy use of Unicode literals, Python 3.3 once again supports the "u" prefix for string literals. This prefix has no semantic significance in Python 3, it is provided solely to reduce the number of purely mechanical changes in migrating to Python 3, making it easier for developers to focus on the more significant semantic changes (such as the stricter default separation of binary and text data).

10 PEP 3155: Qualified name for classes and functions

PEP 3155 - Qualified name for classes and functions PEP written and implemented by Antoine Pitrou.

Functions and class objects have a new `__qualname__` attribute representing the "path" from the module top-level to their definition. For global functions and classes, this is the same as `__name__`. For other functions and classes, it provides better information about where they were actually defined, and how they might be accessible from the global scope.

Example with (non-bound) methods:

```
>>> class C:
...     def meth(self):
...         pass
>>> C.meth.__name__
'meth'
>>> C.meth.__qualname__
'C.meth'
```

Example with nested classes:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.D.__name__
'D'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__name__
'meth'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Example with nested functions:

```
>>> def outer():
...     def inner():
...         pass
...     return inner
...
>>> outer().__name__
'inner'
>>> outer().__qualname__
'outer.<locals>.inner'
```

The string representation of those objects is also changed to include the new, more precise information:

```
>>> str(C.D)
"<class '__main__.C.D'>"
>>> str(C.D.meth)
'<function C.D.meth at 0x7f46b9fe31e0>'
```

11 PEP 412: Key-Sharing Dictionary

PEP 412 - Key-Sharing Dictionary PEP written and implemented by Mark Shannon.

Dictionaries used for the storage of objects' attributes are now able to share part of their internal storage between each other (namely, the part which stores the keys and their respective hashes). This reduces the memory consumption of programs creating many instances of non-builtin types.

12 PEP 362: Function Signature Object

PEP 362: - Function Signature Object PEP written by Brett Cannon, Yury Selivanov, Larry Hastings, Jiwon Seo. Implemented by Yury Selivanov.

A new function `inspect.signature()` makes introspection of python callables easy and straightforward. A broad range of callables is supported: python functions, decorated or not, classes, and `functools.partial()` objects. New classes `inspect.Signature`, `inspect.Parameter` and `inspect.BoundArguments` hold information about the call signatures, such as, annotations, default values, parameters kinds, and bound arguments, which considerably simplifies writing decorators and any code that validates or amends calling signatures or arguments.

13 Using importlib as the Implementation of Import

issue 2377 - Replace __import__ w/ importlib.__import__ issue 13959 - Re-implement parts of `imp` in pure Python issue 14605 - Make import machinery explicit issue 14646 - Require loaders set __loader__ and __package__

(Written by Brett Cannon)

The `__import__()` function is now powered by `importlib.__import__()`. This work leads to the completion of "phase 2" of **PEP 302**. There are multiple benefits to this change. First, it has allowed for more of the machinery powering import to be exposed instead of being implicit and hidden within the C code. It also provides a single implementation for all Python VMs supporting Python 3.3 to use, helping to end any VM-specific deviations in import semantics. And finally it eases the maintenance of import, allowing for future growth to occur.

For the common user, this change should result in no visible change in semantics. Any possible changes required in one's code to handle this change should read the Porting Python code section of this document to see what needs to be changed, but it will only affect those that currently manipulate import or try calling it programmatically.

13.1 New APIs

One of the large benefits of this work is the exposure of what goes into making the import statement work. That means the various importers that were once implicit are now fully exposed as part of the `importlib` package.

The abstract base classes defined in `importlib.abc` have been expanded to properly delineate between *meta path finders* and *path entry finders* by introducing `importlib.abc.MetaPathFinder` and `importlib.abc.PathEntryFinder`, respectively. The old ABC of `importlib.abc.Finder` is now only provided for backwards-compatibility and does not enforce any method requirements.

In terms of finders, `importlib.machinery.FileFinder` exposes the mechanism used to search for source and bytecode files of a module. Previously this class was an implicit member of `sys.path_hooks`.

For loaders, the new abstract base class `importlib.abc.FileLoader` helps write a loader that uses the file system as the storage mechanism for a module's code. The loader for source files (`importlib.machinery.SourceFileLoader`), sourceless bytecode files (`importlib.machinery.SourcelessFileLoader`), and extension modules (`importlib.machinery.ExtensionFileLoader`) are now available for direct use.

`ImportError` now has `name` and `path` attributes which are set when there is relevant data to provide. The message for failed imports will also provide the full name of the module now instead of just the tail end of the module's name.

The `importlib.invalidate_caches()` function will now call the method with the same name on all finders cached in `sys.path_importer_cache` to help clean up any stored state as necessary.

13.2 Visible Changes

[For potential required changes to code, see the Porting Python code section]

Beyond the expanse of what `importlib` now exposes, there are other visible changes to import. The biggest is that `sys.meta_path` and `sys.path_hooks` now store all of the meta path finders and path entry hooks used by import. Previously the finders were implicit and hidden within the C code of import instead of being directly exposed. This means that one can now easily remove or change the order of the various finders to fit one's needs.

Another change is that all modules have a `__loader__` attribute, storing the loader used to create the module. **PEP 302** has been updated to make this attribute mandatory for loaders to implement, so in the future once 3rd-party loaders have been updated people will be able to rely on the existence of the attribute. Until such time, though, import is setting the module post-load.

Loaders are also now expected to set the `__package__` attribute from

PEP 366. Once again, import itself is already setting this on all loaders from `importlib` and import itself is setting the attribute post-load.

`None` is now inserted into `sys.path_importer_cache` when no finder can be found on `sys.path_hooks`. Since `imp.NullImporter` is not directly exposed on `sys.path_hooks` it could no longer be relied upon to always be available to use as a value representing no finder found.

All other changes relate to semantic changes which should be taken into consideration when updating code for Python 3.3, and thus should be read about in the Porting Python code section of this document.

14 New Email Package Features

14.1 Policy Framework

The email package now has a `policy` framework. A `Policy` is an object with several methods and properties that control how the email package behaves. The primary policy for Python 3.3 is the `Compat32` policy, which provides backward compatibility with the email package in Python 3.2. A `policy` can be specified when an email message is parsed by a `parser`, or when a `Message` object is created, or when an email is serialized using a `generator`. Unless overridden, a policy passed to a `parser` is inherited by all the `Message` object and sub-objects created by the `parser`. By default a `generator` will use the policy of the `Message` object it is serializing. The default policy is `compat32`.

The minimum set of controls implemented by all `policy` objects are:

max_line_length	The maximum length, excluding the linesep character(s), individual lines may have when a `Message` is serialized. Defaults to 78.
linesep	The character used to separate individual lines when a `Message` is serialized. Defaults to `\n`.
cte_type	`7bit` or `8bit`. `8bit` applies only to a `Bytes generator`, and means that non-ASCII may be used where allowed by the protocol (or where it exists in the original input).
raise_on_defect	Causes a `parser` to raise error when defects are encountered instead of adding them to the `Message` object's `defects` list.

A new policy instance, with new settings, is created using the `clone()` method of policy objects. `clone` takes any of the above controls as keyword arguments. Any control not specified in the call retains its default value. Thus you can create a policy that uses `\r\n` linesep characters like this:

```
mypolicy = compat32.clone(linesep='\r\n')
```

Policies can be used to make the generation of messages in the format needed by your application simpler. Instead of having to remember to specify `linesep='\r\n'` in all the places you call a `generator`, you can specify it once, when you set the policy used by the `parser` or the `Message`, whichever your program uses to create `Message` objects. On the other hand, if you need to generate messages in multiple forms, you can still specify the parameters in the appropriate `generator` call. Or you can have custom policy instances for your different cases, and pass those in when you create the `generator`.

14.2 Provisional Policy with New Header API

While the policy framework is worthwhile all by itself, the main motivation for introducing it is to allow the creation of new policies that implement new features for the email package in a way that maintains backward compatibility for those who do not use the new policies. Because the new policies introduce a new API, we are releasing them in Python 3.3 as a *provisional policy*. Backwards incompatible changes (up to and including removal of the code) may occur if deemed necessary by the core developers.

The new policies are instances of `EmailPolicy`, and add the following additional controls:

refold_source	Controls whether or not headers parsed by a `parser` are refolded by the `generator`. It can be `none`, `long`, or `all`. The default is `long`, which means that source headers with a line longer than `max_line_length` get refolded. `none` means no line get refolded, and `all` means that all lines get refolded.
header_factory	A callable that take a `name` and `value` and produces a custom header object.

The `header_factory` is the key to the new features provided by the new policies. When one of the new policies is used, any header retrieved from a `Message` object is an object produced by the `header_factory`, and any time you set a header on a `Message` it becomes an object produced by `header_factory`. All such header objects have a `name` attribute equal to the header name. Address and Date headers have additional attributes that give you access to the parsed data of the header. This means you can now do things like this:

```
>>> m = Message(policy=SMTP)
>>> m['To'] = 'Éric <foo@example.com>'
>>> m['to']
'Éric <foo@example.com>'
>>> m['to'].addresses
(Address(display_name='Éric', username='foo', domain='example.com'),)
>>> m['to'].addresses[0].username
'foo'
>>> m['to'].addresses[0].display_name
'Éric'
>>> m['Date'] = email.utils.localtime()
>>> m['Date'].datetime
datetime.datetime(2012, 5, 25, 21, 39, 24, 465484, tzinfo=datetime.timezone(datetime.tim
>>> m['Date']
'Fri, 25 May 2012 21:44:27 -0400'
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
```

You will note that the unicode display name is automatically encoded as `utf-8` when the message is serialized, but that when the header is accessed directly, you get the unicode version. This eliminates any need to deal with the `email.header` `decode_header()` or `make_header()` functions.

You can also create addresses from parts:

```
>>> m['cc'] = [Group('pals', [Address('Bob', 'bob', 'example.com'),
...                           Address('Sally', 'sally', 'example.com')]),
...            Address('Bonzo', addr_spec='bonz@laugh.com')]
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
```

```
Date: Fri, 25 May 2012 21:44:27 -0400
cc: pals: Bob <bob@example.com>, Sally <sally@example.com>;, Bonzo <bonz@laugh.com>
```

Decoding to unicode is done automatically:

```
>>> m2 = message_from_string(str(m))
>>> m2['to']
'Éric <foo@example.com>'
```

When you parse a message, you can use the `addresses` and `groups` attributes of the header objects to access the groups and individual addresses:

```
>>> m2['cc'].addresses
(Address(display_name='Bob', username='bob', domain='example.com'), Address(display_name
>>> m2['cc'].groups
(Group(display_name='pals', addresses=(Address(display_name='Bob', username='bob', domai
```

In summary, if you use one of the new policies, header manipulation works the way it ought to: your application works with unicode strings, and the email package transparently encodes and decodes the unicode to and from the RFC standard Content Transfer Encodings.

15 Other Language Changes

Some smaller changes made to the core Python language are:

- Added support for Unicode name aliases and named sequences. Both `unicodedata.lookup()` and `'\N{...}'` now resolve name aliases, and `unicodedata.lookup()` resolves named sequences too.

  (Contributed by Ezio Melotti in issue 12753)

- Equality comparisons on `range()` objects now return a result reflecting the equality of the underlying sequences generated by those range objects.

  (issue 13201)

- The `count()`, `find()`, `rfind()`, `index()` and `rindex()` methods of `bytes` and `bytearray` objects now accept an integer between 0 and 255 as their first argument.

  (Contributed by Petri Lehtinen in issue 12170)

- New methods have been added to `list` and `bytearray`: `copy()` and `clear()`.

  (issue 10516)

- Raw bytes literals can now be written `rb"..."` as well as `br"..."`. (Contributed by Antoine Pitrou in issue 13748.)

- `dict.setdefault()` now does only one lookup for the given key, making it atomic when used with built-in types. (Contributed by Filip Gruszczyński in issue 13521.)

16 A Finer-Grained Import Lock

Previous versions of CPython have always relied on a global import lock. This led to unexpected annoyances, such as deadlocks when importing a module would trigger code execution in a different thread as a side-effect. Clumsy workarounds were sometimes employed, such as the `PyImport_ImportModuleNoBlock()` C API function.

In Python 3.3, importing a module takes a per-module lock. This correctly serializes importation of a given module from multiple threads (preventing the exposure of incompletely initialized modules), while eliminating the aforementioned annoyances.

(contributed by Antoine Pitrou in issue 9260.)

17 Builtin functions and types

- `open()` gets a new *opener* parameter: the underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). It can be used to use custom flags like `os.O_CLOEXEC` for example. The `'x'` mode was added: open for exclusive creation, failing if the file already exists.
- `print()`: added the *flush* keyword argument. If the *flush* keyword argument is true, the stream is forcibly flushed.
- `hash()`: hash randomization is enabled by default, see `object.__hash__()` and `PYTHONHASHSEED`.
- The `str` type gets a new `casefold()` method: return a casefolded copy of the string, casefolded strings may be used for caseless matching. For example, `'ß'.casefold()` returns `'ss'`.
- The sequence documentation has been substantially rewritten to better explain the binary/text sequence distinction and to provide specific documentation sections for the individual builtin sequence types (issue 4966)

18 New Modules

18.1 faulthandler

This new debug module contains functions to dump Python tracebacks explicitly, on a fault (a crash like a segmentation fault), after a timeout, or on a user signal. Call `faulthandler.enable()` to install fault handlers for the `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS`, and `SIGILL` signals. You can also enable them at startup by setting the

`PYTHONFAULTHANDLER` environment variable or by using *`-X`* `faulthandler` command line option.

Example of a segmentation fault on Linux:

```
$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700:
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

18.2 ipaddress

The new `ipaddress` module provides tools for creating and manipulating objects representing IPv4 and IPv6 addresses, networks and interfaces (i.e. an IP address associated with a specific IP subnet).

(Contributed by Google and Peter Moody in **PEP 3144**)

18.3 lzma

The newly-added `lzma` module provides data compression and decompression using the LZMA algorithm, including support for the `.xz` and `.lzma` file formats.

(Contributed by Nadeem Vawda and Per Øyvind Karlsen in issue 6715)

19 Improved Modules

19.1 abc

Improved support for abstract base classes containing descriptors composed with abstract methods. The recommended approach to declaring abstract descriptors is now to provide `__isabstractmethod__` as a dynamically updated property. The built-in descriptors have been updated accordingly.

- `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
- `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
- `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.

(Contributed by Darren Dale in issue 11610)

19.2 array

The `array` module supports the `long long` type using `q` and `Q` type codes.

(Contributed by Oren Tirosh and Hirokazu Yamamoto in issue 1172711)

19.3 base64, binascii

ASCII-only Unicode strings are now accepted by the decoding functions of the modern interface. For example, `base64.b64decode('YWJj')` returns `b'abc'`.

19.4 bz2

The `bz2` module has been rewritten from scratch. In the process, several new features have been added:

- New `bz2.open()` function: open a bzip2-compressed file in binary or text mode.
- `bz2.BZ2File` can now read from and write to arbitrary file-like objects, by means of its constructor's *fileobj* argument.

  (Contributed by Nadeem Vawda in issue 5863)
- `bz2.BZ2File` and `bz2.decompress()` can now decompress multi-stream inputs (such as those produced by the **pbzip2** tool). `bz2.BZ2File` can now also be used to create this type of file, using the `'a'` (append) mode.

  (Contributed by Nir Aides in issue 1625)
- `bz2.BZ2File` now implements all of the `io.BufferedIOBase` API, except for the `detach()` and `truncate()` methods.

19.5 codecs

The `mbcs` codec has been rewritten to handle correctly `replace` and `ignore` error handlers on all Windows versions. The `mbcs` codec now supports all error handlers, instead of only `replace` to encode and `ignore` to decode.

A new Windows-only codec has been added: `cp65001` (issue 13216). It is the Windows code page 65001 (Windows UTF-8, `CP_UTF8`). For example, it is used by `sys.stdout` if the console output code page is set to cp65001 (e.g., using `chcp 65001` command).

Multibyte CJK decoders now resynchronize faster. They only ignore the first byte of an invalid byte sequence. For example, `b'\xff\n'.decode('gb2312', 'replace')` now returns a `\n` after the replacement character.

(issue 12016)

Incremental CJK codec encoders are no longer reset at each call to their encode() methods. For example:

```
$ ./python -q
>>> import codecs
>>> encoder = codecs.getincrementalencoder('hz')('strict')
>>> b''.join(encoder.encode(x) for x in '\u52ff\u65bd\u65bc\u4eba\u3002 Bye.')
b'~{NpJ)l6HK!#~} Bye.'
```

This example gives `b'~{Np~}~{J)~}~{l6~}~{HK~}~{!#~} Bye.'` with older Python versions.

(issue 12100)

The `unicode_internal` codec has been deprecated.

19.6 collections

Addition of a new `ChainMap` class to allow treating a number of mappings as a single unit.

(Written by Raymond Hettinger for issue 11089, made public in issue 11297)

The abstract base classes have been moved in a new `collections.abc` module, to better differentiate between the abstract and the concrete collections classes. Aliases for ABCs are still present in the `collections` module to preserve existing imports.

(issue 11085)

19.7 contextlib

`ExitStack` now provides a solid foundation for programmatic manipulation of context managers and similar cleanup functionality. Unlike the previous `contextlib.nested` API (which was deprecated and removed), the new API is designed to work correctly regardless of whether context managers acquire their resources in their `__init__` method (for example, file objects) or in their `__enter__` method (for example, synchronisation objects from the `threading` module).

(issue 13585)

19.8 crypt

Addition of salt and modular crypt format (hashing method) and the `mksalt()` function to the `crypt` module.

(issue 10924)

19.9 curses

- If the `curses` module is linked to the ncursesw library, use Unicode functions when Unicode strings or characters are passed (e.g. `waddwstr()`), and bytes functions otherwise (e.g. `waddstr()`).
- Use the locale encoding instead of `utf-8` to encode Unicode strings.
- `curses.window` has a new `curses.window.encoding` attribute.
- The `curses.window` class has a new `get_wch()` method to get a wide character
- The `curses` module has a new `unget_wch()` function to push a wide character so the next `get_wch()` will return it

(Contributed by Iñigo Serna in issue 6755)

19.10 datetime

- Equality comparisons between naive and aware `datetime` instances don't raise `TypeError`.
- New `datetime.datetime.timestamp()` method: Return POSIX timestamp corresponding to the `datetime` instance.
- The `datetime.datetime.strftime()` method supports formatting years older than 1000.
- XXX The `datetime.datetime.astimezone()` method can now be called without arguments to convert datetime instance to the system timezone.

19.11 decimal

issue 7652 - integrate fast native decimal arithmetic. C-module and libmpdec written by Stefan Krah.

The new C version of the decimal module integrates the high speed libmpdec library for arbitrary precision correctly-rounded decimal floating point arithmetic. libmpdec conforms to IBM's General Decimal Arithmetic Specification.

Performance gains range from 10x for database applications to 100x for numerically intensive applications. These numbers are expected gains for standard precisions used in decimal floating point arithmetic. Since the precision is user configurable, the exact figures may vary. For example, in integer bignum arithmetic the differences can be significantly higher.

The following table is meant as an illustration. Benchmarks are available at http://www.bytereef.org/mpdecimal/quickstart.html.

	decimal.py	_decimal	speedup
pi	42.02s	0.345s	120x
telco	172.19s	5.68s	30x
psycopg	3.57s	0.29s	12x

Features

- The `FloatOperation` signal optionally enables stricter semantics for mixing floats and Decimals.
- If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the variable `HAVE_THREADS` is set to False.

API changes

- The C module has the following context limits, depending on the machine architecture:

	32-bit	64-bit
`MAX_PREC`	`425000000`	`999999999999999999`
`MAX_EMAX`	`425000000`	`999999999999999999`
`MIN_EMIN`	`-425000000`	`-999999999999999999`

- In the context templates (`DefaultContext`, `BasicContext` and `ExtendedContext`) the magnitude of `Emax` and `Emin` has changed to `999999`.
- The `Decimal` constructor in decimal.py does not observe the context limits and converts values with arbitrary exponents or precision exactly. Since the C version has internal limits, the following scheme is used: If possible, values are converted exactly, otherwise `InvalidOperation` is raised and the result is NaN. In the latter case it is always possible to use `create_decimal()` in order to obtain a rounded or inexact value.
- The power function in decimal.py is always correctly-rounded. In the C version, it is defined in terms of the correctly-rounded `exp()` and `ln()` functions, but the final result is only "almost always correctly rounded".

- In the C version, the context dictionary containing the signals is a `MutableMapping`. For speed reasons, `flags` and `traps` always refer to the same `MutableMapping` that the context was initialized with. If a new signal dictionary is assigned, `flags` and `traps` are updated with the new values, but they do not reference the RHS dictionary.
- Pickling a `Context` produces a different output in order to have a common interchange format for the Python and C versions.
- The order of arguments in the `Context` constructor has been changed to match the order displayed by `repr()`.
- The `watchexp` parameter in the `quantize()` method is deprecated.

19.12 ftplib

The `FTP_TLS` class now provides a new `ccc()` function to revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

(Contributed by Giampaolo Rodolà in issue 12139)

19.13 gc

It is now possible to register callbacks invoked by the garbage collector before and after collection using the new `callbacks` list.

19.14 hmac

A new `compare_digest()` function has been added to prevent side channel attacks on digests through timing analysis.

(Contributed by Nick Coghlan and Christian Heimes in issue:*15061*)

19.15 imaplib

The `IMAP4_SSL` constructor now accepts an SSLContext parameter to control parameters of the secure channel.

(Contributed by Sijin Joseph in issue 8808)

19.16 inspect

A new `getclosurevars()` function has been added. This function reports the current binding of all names referenced from the function body and where those names were resolved, making it easier to verify correct internal state when testing code that relies on stateful closures.

(Contributed by Meador Inge and Nick Coghlan in issue 13062)

A new `getgeneratorlocals()` function has been added. This function reports the current binding of local variables in the generator's stack frame, making it easier to verify correct internal state when testing generators.

(Contributed by Meador Inge in issue 15153)

19.17 io

The `open()` function has a new `'x'` mode that can be used to exclusively create a new file, and raise a `FileExistsError` if the file already exists. It is based on the C11 'x' mode to fopen().

(Contributed by David Townshend in issue 12760)

The constructor of the `TextIOWrapper` class has a new *write_through* optional argument. If *write_through* is `True`, calls to `write()` are guaranteed not to be buffered: any data written on the `TextIOWrapper` object is immediately handled to its underlying binary buffer.

19.18 math

The `math` module has a new function:

- `log2()`: return the base-2 logarithm of *x* (Written by Mark Dickinson in issue 11888).

19.19 multiprocessing

The new `multiprocessing.connection.wait()` function allows to poll multiple objects (such as connections, sockets and pipes) with a timeout. (Contributed by Richard Oudkerk in issue 12328.)

`multiprocessing.Connection` objects can now be transferred over multiprocessing connections. (Contributed by Richard Oudkerk in issue 4892.)

19.20 nntplib

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

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...      n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.python.commi
>>>
```

(Contributed by Giampaolo Rodolà in issue 9795)

19.21 os

- The `os` module has a new `pipe2()` function that makes it possible to create a pipe with `O_CLOEXEC` or `O_NONBLOCK` flags set atomically. This is especially useful to avoid race conditions in multi-threaded programs.
- The `os` module has a new `sendfile()` function which provides an efficent "zero-copy" way for copying data from one file (or socket) descriptor to another. The phrase "zero-copy" refers to the fact that all of the copying of data between the two descriptors is done entirely by the kernel, with no copying of data into userspace buffers. `sendfile()` can be used to efficiently copy data from a file on disk to a network socket, e.g. for downloading a file.

  (Patch submitted by Ross Lagerwall and Giampaolo Rodolà in issue 10882.)
- To avoid race conditions like symlink attacks and issues with temporary files and directories, it is more reliable (and also faster) to manipulate file descriptors instead of file names. Python 3.3 enhances existing functions and introduces new functions to work on file descriptors (issue 4761, issue 10755 and issue 14626).
  - The `os` module has a new `fwalk()` function similar to `walk()` except that it also yields file descriptors referring to the directories visited. This is especially useful to avoid symlink races.
  - The following functions get new optional *dir_fd* (*paths relative to directory descriptors*) and/or *follow_symlinks* (*not following symlinks*): `access()`, `chflags()`, `chmod()`, `chown()`, `link()`, `lstat()`, `mkdir()`, `mkfifo()`, `mknod()`, `open()`, `readlink()`, `remove()`, `rename()`, `replace()`, `rmdir()`, `stat()`, `symlink()`, `unlink()`, `utime()`.

  - The following functions now support a file descriptor for their path argument: `chdir()`, `chmod()`, `chown()`, `execve()`, `listdir()`, `pathconf()`, `exists()`, `stat()`, `statvfs()`, `utime()`.

- The `os` module has two new functions: `getpriority()` and `setpriority()`. They can be used to get or set process niceness/priority in a fashion similar to `os.nice()` but extended to all processes instead of just the current one.

  (Patch submitted by Giampaolo Rodolà in issue 10784.)

- The new `os.replace()` function allows cross-platform renaming of a file with overwriting the destination. With `os.rename()`, an existing destination file is overwritten under POSIX, but raises an error under Windows. (Contributed by Antoine Pitrou in issue 8828.)

- The stat family of functions (`stat()`, `fstat()`, and `lstat()`) now support reading a file's timestamps with nanosecond precision. Symmetrically, `utime()` can now write file timestamps with nanosecond precision. (Contributed by Larry Hastings in issue 14127.)

- The new `os.get_terminal_size()` function queries the size of the terminal attached to a file descriptor. See also `shutil.get_terminal_size()`. (Contributed by Zbigniew Jędrzejewski-Szmek in issue 13609.)

- New functions to support Linux extended attributes (issue 12720): `getxattr()`, `listxattr()`, `removexattr()`, `setxattr()`.

- New interface to the scheduler. These functions control how a process is allocated CPU time by the operating system. New functions: `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getaffinity()`, `sched_getparam()`, `sched_getscheduler()`, `sched_rr_get_interval()`, `sched_setaffinity()`, `sched_setparam()`, `sched_setscheduler()`, `sched_yield()`,

- New functions to control the file system:

  - `posix_fadvise()`: Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations.

  - `posix_fallocate()`: Ensures that enough disk space is allocated for a file.

  - `sync()`: Force write of everything to disk.

- Add some extra posix functions to the os module:

  - `lockf()`: Apply, test or remove a POSIX lock on an open file descriptor.

  - `pread()`: Read from a file descriptor at an offset, the file offset remains unchanged.

  - `pwrite()`: Write to a file descriptor from an offset, leaving the file offset unchanged.

  - `readv()`: Read from a file descriptor into a number of writable buffers.

  - `truncate()`: Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.

  - `waitid()`: Wait for the completion of one or more child processes.

  - `writev()`: Write the contents of *buffers* to a file descriptor, where *buffers* is an arbitrary sequence of buffers.

  - `getgrouplist()` (issue 9344): Return list of group ids that specified user belongs to.

- `times()` and `uname()`: Return type changed from a tuple to a tuple-like object with named attributes.

19.22 pdb

- Tab-completion is now available not only for command names, but also their arguments. For example, for the `break` command, function and file names are completed. (Contributed by Georg Brandl in issue 14210)

19.23 pickle

`pickle.Pickler` objects now have an optional `dispatch_table` attribute allowing to set per-pickler reduction functions. (Contributed by Richard Oudkerk in issue 14166.)

19.24 pydoc

The Tk GUI and the `serve()` function have been removed from the `pydoc` module: `pydoc -g` and `serve()` have been deprecated in Python 3.2.

19.25 re

`str` regular expressions now support `\u` and `\U` escapes.

(Contributed by Serhiy Storchaka in issue 3665.)

19.26 sched

- `run()` now accepts a *blocking* parameter which when set to False makes the method execute the scheduled events due to expire soonest (if any) and then return immediately. This is useful in case you want to use the `scheduler` in non-blocking applications. (Contributed by Giampaolo Rodolà in issue 13449)
- `scheduler` class can now be safely used in multi-threaded environments. (Contributed by Josiah Carlson and Giampaolo Rodolà in issue 8684)
- *timefunc* and *delayfunct* parameters of `scheduler` class constructor are now optional and defaults to `time.time()` and `time.sleep()` respectively. (Contributed by Chris Clark in issue 13245)
- `enter()` and `enterabs()` *argument* parameter is now optional. (Contributed by Chris Clark in issue 13245)
- `enter()` and `enterabs()` now accept a *kwargs* parameter. (Contributed by Chris Clark in issue 13245)

19.27 shutil

- The `shutil` module has these new fuctions:
  - `disk_usage()`: provides total, used and free disk space statistics. (Contributed by Giampaolo Rodolà in issue 12442)
  - `chown()`: allows one to change user and/or group of the given path also specifying the user/group names and not only their numeric ids. (Contributed by Sandro Tosi in issue 12191)
- `copy2()` and `copystat()` now preserve file timestamps with nanosecond precision on platforms that support it. They also preserve file "extended attributes" on Linux. (Contributed by Larry Hastings in issue 14127 and issue 15238.)
- The new `shutil.get_terminal_size()` function returns the size of the terminal window the interpreter is attached to. (Contributed by Zbigniew Jędrzejewski-Szmek in issue 13609.)
- Several functions now take an optional `symlinks` argument: when that parameter is true, symlinks aren't dereferenced and the operation instead acts on the symlink itself (or creates one, if relevant). (Contributed by Hynek Schlawack in issue 12715.)
- `rmtree()` is now resistant to symlink attacks on platforms which support the new `dir_fd` parameter in `os.open()` and `os.unlink()`. (Contributed by Martin von Löwis and Hynek Schlawack in issue 4489.)

19.28 signal

- The `signal` module has new functions:
  - `pthread_sigmask()`: fetch and/or change the signal mask of the calling thread (Contributed by Jean-Paul Calderone in issue 8407) ;
  - `pthread_kill()`: send a signal to a thread ;
  - `sigpending()`: examine pending functions ;
  - `sigwait()`: wait a signal.
  - `sigwaitinfo()`: wait for a signal, returning detailed information about it.
  - `sigtimedwait()`: like `sigwaitinfo()` but with a timeout.
- The signal handler writes the signal number as a single byte instead of a nul byte into the wakeup file descriptor. So it is possible to wait more than one signal and know which signals were raised.
- `signal.signal()` and `signal.siginterrupt()` raise an OSError, instead of a RuntimeError: OSError has an errno attribute.

19.29 smtplib

The `SMTP_SSL` constructor and the `starttls()` method now accept an SSLContext parameter to control parameters of the secure channel.

(Contributed by Kasun Herath in issue 8809)

19.30 socket

- The `socket` class now exposes additional methods to process ancillary data when supported by the underlying platform:
  - `sendmsg()`
  - `recvmsg()`
  - `recvmsg_into()`

  (Contributed by David Watson in issue 6560, based on an earlier patch by Heiko Wundram)
- The `socket` class now supports the PF_CAN protocol family (http://en.wikipedia.org/wiki/Socketcan), on Linux (http://lwn.net/Articles/253425).

  (Contributed by Matthias Fuchs, updated by Tiago Gonçalves in issue 10141)
- The `socket` class now supports the PF_RDS protocol family (http://en.wikipedia.org/wiki/Reliable_Datagram_Sockets and http://oss.oracle.com/projects/rds/).

19.31 ssl

- The `ssl` module has two new random generation functions:
  - `RAND_bytes()`: generate cryptographically strong pseudo-random bytes.
  - `RAND_pseudo_bytes()`: generate pseudo-random bytes.

  (Contributed by Victor Stinner in issue 12049)
- The `ssl` module now exposes a finer-grained exception hierarchy in order to make it easier to inspect the various kinds of errors.

  (Contributed by Antoine Pitrou in issue 11183)

- `load_cert_chain()` now accepts a *password* argument to be used if the private key is encrypted.

  (Contributed by Adam Simpkins in issue 12803)

- Diffie-Hellman key exchange, both regular and Elliptic Curve-based, is now supported through the `load_dh_params()` and `set_ecdh_curve()` methods.

  (Contributed by Antoine Pitrou in issue 13626 and issue 13627)

- SSL sockets have a new `get_channel_binding()` method allowing the implementation of certain authentication mechanisms such as SCRAM-SHA-1-PLUS.

  (Contributed by Jacek Konieczny in issue 12551)

- You can query the SSL compression algorithm used by an SSL socket, thanks to its new `compression()` method.

  (Contributed by Antoine Pitrou in issue 13634)

- Support has been added for the Next Procotol Negotiation extension using the `ssl.SSLContext.set_npn_protocols()` method.

  (Contributed by Colin Marc in issue 14204)

- SSL errors can now be introspected more easily thanks to `library` and `reason` attributes.

  (Contributed by Antoine Pitrou in issue 14837)

19.32 stat

- The undocumented tarfile.filemode function has been moved to `stat.filemode()`. It can be used to convert a file's mode to a string of the form '-rwxrwxrwx'.

  (Contributed by Giampaolo Rodolà in issue 14807)

19.33 sys

- The `sys` module has a new `thread_info` *struct sequence* holding informations about the thread implementation.

  (issue 11223)

19.34 textwrap

- The `textwrap` module has a new `indent()` that makes it straightforward to add a common prefix to selected lines in a block of text.

  (issue 13857)

19.35 time

The **PEP 418** added new functions to the `time` module:

- `get_clock_info()`: Get information on a clock.
- `monotonic()`: Monotonic clock (cannot go backward), not affected by system clock updates.
- `perf_counter()`: Performance counter with the highest available resolution to measure a short duration.
- `process_time()`: Sum of the system and user CPU time of the current process.

Other new functions:

- `clock_getres()`, `clock_gettime()` and `clock_settime()` functions with `CLOCK_xxx` constants. (Contributed by Victor Stinner in issue 10278)

19.36 types

Add a new `types.MappingProxyType` class: Read-only proxy of a mapping. (issue 14386)

The new functions *types.new_class* and *types.prepare_class* provide support for PEP 3115 compliant dynamic type creation. (issue 14588)

19.37 urllib

The `Request` class, now accepts a *method* argument used by `get_method()` to determine what HTTP method should be used. For example, this will send a `'HEAD'` request:

```
>>> urlopen(Request('http://www.python.org', method='HEAD'))
```

(issue 1673007)

19.38 webbrowser

The `webbrowser` module supports more browsers: Google Chrome (named **chrome**, **chromium**, **chrome-browser** or **chromium-browser** depending on the version and operating system) as well as the the generic launchers **xdg-open** from the FreeDesktop.org project and **gvfs-open** which is the default URI handler for GNOME 3.

(issue 13620 and issue 14493)

19.39 xml.etree.ElementTree

The `xml.etree.ElementTree` module now imports its C accelerator by default; there is no longer a need to explicitly import `xml.etree.cElementTree` (this module stays for backwards compatibility, but is now deprecated). In addition, the `iter` family of methods of `Element` has been optimized (rewritten in C). The module's documentation has also been greatly improved with added examples and a more detailed reference.

20 Optimizations

Major performance enhancements have been added:

- Thanks to **PEP 393**, some operations on Unicode strings have been optimized:
  - the memory footprint is divided by 2 to 4 depending on the text
  - encode an ASCII string to UTF-8 doesn't need to encode characters anymore, the UTF-8 representation is shared with the ASCII representation
  - the UTF-8 encoder has been optimized
  - repeating a single ASCII letter and getting a substring of a ASCII strings is 4 times faster
- UTF-8 is now 2x to 4x faster. UTF-16 encoding is now up to 10x faster.

  (contributed by Serhiy Storchaka, issue 14624, issue 14738 and issue 15026.)

21 Build and C API Changes

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

- New **PEP 3118** related function:
  - `PyMemoryView_FromMemory()`
- **PEP 393** added new Unicode types, macros and functions:

- High-level API:
  - `PyUnicode_CopyCharacters()`
  - `PyUnicode_FindChar()`
  - `PyUnicode_GetLength()`, `PyUnicode_GET_LENGTH`
  - `PyUnicode_New()`
  - `PyUnicode_Substring()`
  - `PyUnicode_ReadChar()`, `PyUnicode_WriteChar()`
- Low-level API:
  - `Py_UCS1`, `Py_UCS2`, `Py_UCS4` types
  - `PyASCIIObject` and `PyCompactUnicodeObject` structures
  - `PyUnicode_READY`
  - `PyUnicode_FromKindAndData()`
  - `PyUnicode_AsUCS4()`, `PyUnicode_AsUCS4Copy()`
  - `PyUnicode_DATA`, `PyUnicode_1BYTE_DATA`, `PyUnicode_2BYTE_DATA`, `PyUnicode_4BYTE_DATA`
  - `PyUnicode_KIND` with `PyUnicode_Kind` enum: `PyUnicode_WCHAR_KIND`, `PyUnicode_1BYTE_KIND`, `PyUnicode_2BYTE_KIND`, `PyUnicode_4BYTE_KIND`
  - `PyUnicode_READ`, `PyUnicode_READ_CHAR`, `PyUnicode_WRITE`
  - `PyUnicode_MAX_CHAR_VALUE`

22 Deprecated

22.1 Unsupported Operating Systems

OS/2 and VMS are no longer supported due to the lack of a maintainer.

Windows 2000 and Windows platforms which set `COMSPEC` to `command.com` are no longer supported due to maintenance burden.

22.2 Deprecated Python modules, functions and methods

- The `unicode_internal` codec has been deprecated because of the

  PEP 393, use UTF-8, UTF-16 (`utf-16-le` or `utf-16-be`), or UTF-32 (`utf-32-le` or `utf-32-be`)
- `ftplib.FTP.nlst()` and `ftplib.FTP.dir()`: use `ftplib.FTP.mlsd()`
- `platform.popen()`: use the `subprocess` module. Check especially the *subprocess-replacements* section.
- issue 13374: The Windows bytes API has been deprecated in the `os` module. Use Unicode filenames, instead of bytes filenames, to not depend on the ANSI code page anymore and to support any filename.
- issue 13988: The `xml.etree.cElementTree` module is deprecated. The accelerator is used automatically whenever available.
- The behaviour of `time.clock()` depends on the platform: use the new `time.perf_counter()` or `time.process_time()` function instead, depending on your requirements, to have a well defined behaviour.
- The `os.stat_float_times()` function is deprecated.

- `abc` module:
  - `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
  - `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
  - `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.

22.3 Deprecated functions and types of the C API

The `Py_UNICODE` has been deprecated by **PEP 393** and will be removed in Python 4. All functions using this type are deprecated:

Unicode functions and methods using `Py_UNICODE` and `Py_UNICODE*` types:

- `PyUnicode_FromUnicode`: use `PyUnicode_FromWideChar()` or `PyUnicode_FromKindAndData()`
- `PyUnicode_AS_UNICODE`, `PyUnicode_AsUnicode()`, `PyUnicode_AsUnicodeAndSize()`: use `PyUnicode_AsWideCharString()`
- `PyUnicode_AS_DATA`: use `PyUnicode_DATA` with `PyUnicode_READ` and `PyUnicode_WRITE`
- `PyUnicode_GET_SIZE`, `PyUnicode_GetSize()`: use `PyUnicode_GET_LENGTH` or `PyUnicode_GetLength()`
- `PyUnicode_GET_DATA_SIZE`: use `PyUnicode_GET_LENGTH(str) * PyUnicode_KIND(str)` (only work on ready strings)
- `PyUnicode_AsUnicodeCopy()`: use `PyUnicode_AsUCS4Copy()` or `PyUnicode_AsWideCharString()`
- `PyUnicode_GetMax()`

Functions and macros manipulating Py_UNICODE* strings:

- `Py_UNICODE_strlen`: use `PyUnicode_GetLength()` or `PyUnicode_GET_LENGTH`
- `Py_UNICODE_strcat`: use `PyUnicode_CopyCharacters()` or `PyUnicode_FromFormat()`
- `Py_UNICODE_strcpy`, `Py_UNICODE_strncpy`, `Py_UNICODE_COPY`: use `PyUnicode_CopyCharacters()` or `PyUnicode_Substring()`
- `Py_UNICODE_strcmp`: use `PyUnicode_Compare()`
- `Py_UNICODE_strncmp`: use `PyUnicode_Tailmatch()`
- `Py_UNICODE_strchr`, `Py_UNICODE_strrchr`: use `PyUnicode_FindChar()`
- `Py_UNICODE_FILL`: use `PyUnicode_Fill()`
- `Py_UNICODE_MATCH`

Encoders:

- `PyUnicode_Encode()`: use `PyUnicode_AsEncodedObject()`
- `PyUnicode_EncodeUTF7()`
- `PyUnicode_EncodeUTF8()`: use `PyUnicode_AsUTF8()` or `PyUnicode_AsUTF8String()`
- `PyUnicode_EncodeUTF32()`
- `PyUnicode_EncodeUTF16()`
- `PyUnicode_EncodeUnicodeEscape:()` use `PyUnicode_AsUnicodeEscapeString()`
- `PyUnicode_EncodeRawUnicodeEscape:()` use `PyUnicode_AsRawUnicodeEscapeString()`

- `PyUnicode_EncodeLatin1()`: use `PyUnicode_AsLatin1String()`
- `PyUnicode_EncodeASCII()`: use `PyUnicode_AsASCIIString()`
- `PyUnicode_EncodeCharmap()`
- `PyUnicode_TranslateCharmap()`
- `PyUnicode_EncodeMBCS()`: use `PyUnicode_AsMBCSString()` or `PyUnicode_EncodeCodePage()` (with `CP_ACP` code_page)
- `PyUnicode_EncodeDecimal()`, `PyUnicode_TransformDecimalToASCII()`

22.4 Deprecated features

The `array` module's `'u'` format code is now deprecated and will be removed in Python 4 together with the rest of the (`Py_UNICODE`) API.

23 Porting to Python 3.3

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

23.1 Porting Python code

- Hash randomization is enabled by default. Set the `PYTHONHASHSEED` environment variable to `0` to disable hash randomization. See also the `object.__hash__()` method.
- issue 12326: On Linux, sys.platform doesn't contain the major version anymore. It is now always 'linux', instead of 'linux2' or 'linux3' depending on the Linux version used to build Python. Replace sys.platform == 'linux2' with sys.platform.startswith('linux'), or directly sys.platform == 'linux' if you don't need to support older Python versions.
- issue 13847, issue 14180: `time` and `datetime`: `OverflowError` is now raised instead of `ValueError` if a timestamp is out of range. `OSError` is now raised if C functions `gmtime()` or `localtime()` failed.
- The default finders used by import now utilize a cache of what is contained within a specific directory. If you create a Python source file or sourceless bytecode file, make sure to call `importlib.invalidate_caches()` to clear out the cache for the finders to notice the new file.
- `ImportError` now uses the full name of the module that was attemped to be imported. Doctests that check ImportErrors' message will need to be updated to use the full name of the module instead of just the tail of the name.
- The **index** argument to `__import__()` now defaults to 0 instead of -1 and no longer support negative values. It was an oversight when **PEP 328** was implemented that the default value remained -1. If you need to continue to perform a relative import followed by an absolute import, then perform the relative import using an index of 1, followed by another import using an index of 0. It is preferred, though, that you use `importlib.import_module()` rather than call `__import__()` directly.
- `__import__()` no longer allows one to use an index value other than 0 for top-level modules. E.g. `__import__('sys', level=1)` is now an error.
- Because `sys.meta_path` and `sys.path_hooks` now have finders on them by default, you will most likely want to use `list.insert()` instead of `list.append()` to add to those lists.
- Because `None` is now inserted into `sys.path_importer_cache`, if you are clearing out entries in the dictionary of paths that do not have a finder, you will need to remove keys paired with values of `None` **and** `imp.NullImporter` to be backwards-compatible. This will lead to extra overhead on older versions of Python that re-insert `None` into `sys.path_importer_cache` where it repesents the use of implicit finders, but semantically it should not change anything.

- `importlib.abc.SourceLoader.path_mtime()` is now deprecated in favour of `importlib.abc.SourceLoader.path_stats()` as bytecode files now store both the modification time and size of the source file the bytecode file was compiled from.
- `importlib.abc.Finder` no longer specifies a *find_module()* abstract method that must be implemented. If you were relying on subclasses to implement that method, make sure to check for the method's existence first. You will probably want to check for *find_loader()* first, though, in the case of working with *path entry finders*.
- `pkgutil` has been converted to use `importlib` internally. This eliminates many edge cases where the old behaviour of the PEP 302 import emulation failed to match the behaviour of the real import system. The import emulation itself is still present, but is now deprecated. The `pkgutil.iter_importers()` and `pkgutil.walk_packages()` functions special case the standard import hooks so they are still supported even though they do not provide the non-standard `iter_modules()` method.

23.2 Porting C code

- In the course of changes to the buffer API the undocumented `smalltable` member of the `Py_buffer` structure has been removed and the layout of the `PyMemoryViewObject` has changed.

  All extensions relying on the relevant parts in `memoryobject.h` or `object.h` must be rebuilt.
- Due to *PEP 393*, the `Py_UNICODE` type and all functions using this type are deprecated (but will stay available for at least five years). If you were using low-level Unicode APIs to construct and access unicode objects and you want to benefit of the memory footprint reduction provided by PEP 393, you have to convert your code to the new `Unicode API`.

  However, if you only have been using high-level functions such as `PyUnicode_Concat()`, `PyUnicode_Join()` or `PyUnicode_FromFormat()`, your code will automatically take advantage of the new unicode representations.
- `PyImport_GetMagicNumber()` now returns -1 upon failure.
- As a negative value for the **level** argument to `__import__()` is no longer valid, the same now holds for `PyImport_ImportModuleLevel()`. This also means that the value of **level** used by `PyImport_ImportModuleEx()` is now 0 instead of -1.

23.3 Building C extensions

- The range of possible file names for C extensions has been narrowed. Very rarely used spellings have been suppressed: under POSIX, files named `xxxmodule.so`, `xxxmodule.abi3.so` and `xxxmodule.cpython-*.so` are no longer recognized as implementing the `xxx` module. If you had been generating such files, you have to switch to the other spellings (i.e., remove the `module` string from the file names).

  (implemented in issue 14040.)

23.4 Other issues

Index

E

P