What's New in Python

Release 3.14.0

A. M. Kuchling

October 07, 2025

Python Software Foundation
Email: docs@python.org

Contents

Editors
Adam Turner and Hugo van Kemenade

This article explains the new features in Python 3.14, compared to 3.13. Python 3.14 was released on 7 October 2025. For full details, see the changelog.

See also

PEP 745 – Python 3.14 release schedule

1 Summary – Release highlights

Python 3.14 is the latest stable release of the Python programming language, with a mix of changes to the language, the implementation, and the standard library. The biggest changes include *template string literals*, *deferred evaluation of annotations*, and support for *subinterpreters* in the standard library.

The library changes include significantly improved capabilities for *introspection in asyncio*, *support for Zstandard* via a new `compression.zstd` module, syntax highlighting in the REPL, as well as the usual deprecations and removals, and improvements in user-friendliness and correctness.

This article doesn't attempt to provide a complete specification of all new features, but instead gives a convenient overview. For full details refer to the documentation, such as the Library Reference and Language Reference. To understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented. See *Porting to Python 3.14* for guidance on upgrading from earlier versions of Python.

Interpreter improvements:

- **PEP 649** and **PEP 749**: *Deferred evaluation of annotations*
- **PEP 734**: *Multiple interpreters in the standard library*
- **PEP 750**: *Template strings*
- **PEP 758**: *Allow except and except* expressions without brackets*
- **PEP 765**: *Control flow in finally blocks*
- **PEP 768**: *Safe external debugger interface for CPython*
- *A new type of interpreter*
- *Free-threaded mode improvements*
- *Improved error messages*
- *Incremental garbage collection*

Significant improvements in the standard library:

- **PEP 784**: *Zstandard support in the standard library*
- *Asyncio introspection capabilities*
- *Concurrent safe warnings control*
- *Syntax highlighting in the default interactive shell*, and color output in several standard library CLIs

C API improvements:

- **PEP 741**: *Python configuration C API*

Platform support:

- **PEP 776**: Emscripten is now an *officially supported platform*, at **tier 3**.

Release changes:

- **PEP 779**: *Free-threaded Python is officially supported*
- **PEP 761**: *PGP signatures have been discontinued for official releases*
- *Windows and macOS binary releases now support the experimental just-in-time compiler*
- *Binary releases for Android are now provided*

2 New features

2.1 PEP 649 & PEP 749: Deferred evaluation of annotations

The annotations on functions, classes, and modules are no longer evaluated eagerly. Instead, annotations are stored in special-purpose annotate functions and evaluated only when necessary (except if `from __future__ import annotations` is used).

This change is designed to improve performance and usability of annotations in Python in most circumstances. The runtime cost for defining annotations is minimized, but it remains possible to introspect annotations at runtime. It is no longer necessary to enclose annotations in strings if they contain forward references.

The new `annotationlib` module provides tools for inspecting deferred annotations. Annotations may be evaluated in the `VALUE` format (which evaluates annotations to runtime values, similar to the behavior in earlier Python versions), the `FORWARDREF` format (which replaces undefined names with special markers), and the `STRING` format (which returns annotations as strings).

This example shows how these formats behave:

```
>>> from annotationlib import get_annotations, Format
>>> def func(arg: Undefined):
...     pass
>>> get_annotations(func, format=Format.VALUE)
Traceback (most recent call last):
  ...
NameError: name 'Undefined' is not defined
>>> get_annotations(func, format=Format.FORWARDREF)
{'arg': ForwardRef('Undefined', owner=<function func at 0x...>)}
>>> get_annotations(func, format=Format.STRING)
{'arg': 'Undefined'}
```

The *porting* section contains guidance on changes that may be needed due to these changes, though in the majority of cases, code will continue working as-is.

(Contributed by Jelle Zijlstra in **PEP 749** and gh-119180; **PEP 649** was written by Larry Hastings.)

See also

PEP 649
Deferred Evaluation Of Annotations Using Descriptors

PEP 749
Implementing PEP 649

2.2 PEP 734: Multiple interpreters in the standard library

The CPython runtime supports running multiple copies of Python in the same process simultaneously and has done so for over 20 years. Each of these separate copies is called an 'interpreter'. However, the feature had been available only through the C-API.

That limitation is removed in Python 3.14, with the new `concurrent.interpreters` module.

There are at least two notable reasons why using multiple interpreters has significant benefits:

- they support a new (to Python), human-friendly concurrency model
- true multi-core parallelism

For some use cases, concurrency in software enables efficiency and can simplify design, at a high level. At the same time, implementing and maintaining all but the simplest concurrency is often a struggle for the human brain. That especially applies to plain threads (for example, `threading`), where all memory is shared between all threads.

With multiple isolated interpreters, you can take advantage of a class of concurrency models, like CSP or the actor model, that have found success in other programming languages, like Smalltalk, Erlang, Haskell, and Go. Think of multiple interpreters like threads but with opt-in sharing.

Regarding multi-core parallelism: as of Python 3.12, interpreters are now sufficiently isolated from one another to be used in parallel (see **PEP 684**). This unlocks a variety of CPU-intensive use cases for Python that were limited by the GIL.

Using multiple interpreters is similar in many ways to `multiprocessing`, in that they both provide isolated logical "processes" that can run in parallel, with no sharing by default. However, when using multiple interpreters, an application will use fewer system resources and will operate more efficiently (since it stays within the same process). Think of multiple interpreters as having the isolation of processes with the efficiency of threads.

While the feature has been around for decades, multiple interpreters have not been used widely, due to low awareness and the lack of a standard library module. Consequently, they currently have several notable limitations, which will improve significantly now that the feature is finally going mainstream.

Current limitations:

- starting each interpreter has not been optimized yet
- each interpreter uses more memory than necessary (work continues on extensive internal sharing between interpreters)
- there aren't many options *yet* for truly sharing objects or other data between interpreters (other than `memoryview`)
- many third-party extension modules on PyPI are not yet compatible with multiple interpreters (all standard library extension modules *are* compatible)
- the approach to writing applications that use multiple isolated interpreters is mostly unfamiliar to Python users, for now

The impact of these limitations will depend on future CPython improvements, how interpreters are used, and what the community solves through PyPI packages. Depending on the use case, the limitations may not have much impact, so try it out!

Furthermore, future CPython releases will reduce or eliminate overhead and provide utilities that are less appropriate on PyPI. In the meantime, most of the limitations can also be addressed through extension modules, meaning PyPI packages can fill any gap for 3.14, and even back to 3.12 where interpreters were finally properly isolated and stopped sharing the GIL. Likewise, libraries on PyPI are expected to emerge for high-level abstractions on top of interpreters.

Regarding extension modules, work is in progress to update some PyPI projects, as well as tools like Cython, pybind11, nanobind, and PyO3. The steps for isolating an extension module are found at isolating-extensions-howto. Isolating a module has a lot of overlap with what is required to support *free-threading*, so the ongoing work in the community in that area will help accelerate support for multiple interpreters.

Also added in 3.14: *concurrent.futures.InterpreterPoolExecutor*.

(Contributed by Eric Snow in gh-134939.)

See also

PEP 734

2.3 PEP 750: Template string literals

Template strings are a new mechanism for custom string processing. They share the familiar syntax of f-strings but, unlike f-strings, return an object representing the static and interpolated parts of the string, instead of a simple `str`.

To write a t-string, use a `'t'` prefix instead of an `'f'`:

```
>>> variety = 'Stilton'
>>> template = t'Try some {variety} cheese!'
```

(continues on next page)

(continued from previous page)

```
>>> type(template)
<class 'string.templatelib.Template'>
```

`Template` objects provide access to the static and interpolated (in curly braces) parts of a string *before* they are combined. Iterate over `Template` instances to access their parts in order:

```
>>> list(template)
['Try some ', Interpolation('Stilton', 'variety', None, ''), ' cheese!']
```

It's easy to write (or call) code to process `Template` instances. For example, here's a function that renders static parts lowercase and `Interpolation` instances uppercase:

```
from string.templatelib import Interpolation

def lower_upper(template):
    """Render static parts lowercase and interpolations uppercase."""
    parts = []
    for part in template:
        if isinstance(part, Interpolation):
            parts.append(str(part.value).upper())
        else:
            parts.append(part.lower())
    return ''.join(parts)

name = 'Wenslydale'
template = t'Mister {name}'
assert lower_upper(template) == 'mister WENSLYDALE'
```

Because `Template` instances distinguish between static strings and interpolations at runtime, they can be useful for sanitising user input. Writing a `html()` function that escapes user input in HTML is an exercise left to the reader! Template processing code can provide improved flexibility. For instance, a more advanced `html()` function could accept a `dict` of HTML attributes directly in the template:

```
attributes = {'src': 'limburger.jpg', 'alt': 'lovely cheese'}
template = t'<img {attributes}>'
assert html(template) == '<img src="limburger.jpg" alt="lovely cheese" />'
```

Of course, template processing code does not need to return a string-like result. An even *more* advanced `html()` could return a custom type representing a DOM-like structure.

With t-strings in place, developers can write systems that sanitise SQL, make safe shell operations, improve logging, tackle modern ideas in web development (HTML, CSS, and so on), and implement lightweight custom business DSLs.

(Contributed by Jim Baker, Guido van Rossum, Paul Everitt, Koudai Aono, Lysandros Nikolaou, Dave Peck, Adam Turner, Jelle Zijlstra, Bénédikt Tran, and Pablo Galindo Salgado in gh-132661.)

See also

PEP 750.

2.4 PEP 768: Safe external debugger interface

Python 3.14 introduces a zero-overhead debugging interface that allows debuggers and profilers to safely attach to running Python processes without stopping or restarting them. This is a significant enhancement to Python's debugging capabilities, meaning that unsafe alternatives are no longer required.

The new interface provides safe execution points for attaching debugger code without modifying the interpreter's normal execution path or adding any overhead at runtime. Due to this, tools can now inspect and interact with Python

applications in real-time, which is a crucial capability for high-availability systems and production environments.

For convenience, this interface is implemented in the `sys.remote_exec()` function. For example:

```
import sys
from tempfile import NamedTemporaryFile

with NamedTemporaryFile(mode='w', suffix='.py', delete=False) as f:
    script_path = f.name
    f.write(f'import my_debugger; my_debugger.connect({os.getpid()})')

# Execute in process with PID 1234
print('Behold! An offering:')
sys.remote_exec(1234, script_path)
```

This function allows sending Python code to be executed in a target process at the next safe execution point. However, tool authors can also implement the protocol directly as described in the PEP, which details the underlying mechanisms used to safely attach to running processes.

The debugging interface has been carefully designed with security in mind and includes several mechanisms to control access:

- A `PYTHON_DISABLE_REMOTE_DEBUG` environment variable.
- A `-X disable-remote-debug` command-line option.
- A `--without-remote-debug` configure flag to completely disable the feature at build time.

(Contributed by Pablo Galindo Salgado, Matt Wozniski, and Ivona Stojanovic in gh-131591.)

See also

PEP 768.

2.5 A new type of interpreter

A new type of interpreter has been added to CPython. It uses tail calls between small C functions that implement individual Python opcodes, rather than one large C `case` statement. For certain newer compilers, this interpreter provides significantly better performance. Preliminary benchmarks suggest a geometric mean of 3-5% faster on the standard `pyperformance` benchmark suite, depending on platform and architecture. The baseline is Python 3.14 built with Clang 19, without this new interpreter.

This interpreter currently only works with Clang 19 and newer on x86-64 and AArch64 architectures. However, a future release of GCC is expected will support this as well.

This feature is opt-in for now. Enabling profile-guided optimization is highly recommendeded when using the new interpreter as it is the only configuration that has been tested and validated for improved performance. For further information, see `--with-tail-call-interp`.

Note

This is not to be confused with tail call optimization of Python functions, which is currently not implemented in CPython.

This new interpreter type is an internal implementation detail of the CPython interpreter. It doesn't change the visible behavior of Python programs at all. It can improve their performance, but doesn't change anything else.

(Contributed by Ken Jin in gh-128563, with ideas on how to implement this in CPython by Mark Shannon, Garrett Gu, Haoran Xu, and Josh Haberman.)

2.6 Free-threaded mode improvements

CPython's free-threaded mode (**PEP 703**), initially added in 3.13, has been significantly improved in Python 3.14. The implementation described in PEP 703 has been finished, including C API changes, and temporary workarounds in the interpreter were replaced with more permanent solutions. The specializing adaptive interpreter (**PEP 659**) is now enabled in free-threaded mode, which along with many other optimizations greatly improves its performance. The performance penalty on single-threaded code in free-threaded mode is now roughly 5-10%, depending on the platform and C compiler used.

From Python 3.14, when compiling extension modules for the free-threaded build of CPython on Windows, the preprocessor variable `Py_GIL_DISABLED` now needs to be specified by the build backend, as it will no longer be determined automatically by the C compiler. For a running interpreter, the setting that was used at compile time can be found using `sysconfig.get_config_var()`.

The new `-X context_aware_warnings` flag controls if *concurrent safe warnings control* is enabled. The flag defaults to true for the free-threaded build and false for the GIL-enabled build.

A new `thread_inherit_context` flag has been added, which if enabled means that threads created with `threading.Thread` start with a copy of the `Context()` of the caller of `start()`. Most significantly, this makes the warning filtering context established by `catch_warnings` be "inherited" by threads (or asyncio tasks) started within that context. It also affects other modules that use context variables, such as the `decimal` context manager. This flag defaults to true for the free-threaded build and false for the GIL-enabled build.

(Contributed by Sam Gross, Matt Page, Neil Schemenauer, Thomas Wouters, Donghee Na, Kirill Podoprigora, Ken Jin, Itamar Oren, Brett Simmers, Dino Viehland, Nathan Goldbaum, Ralf Gommers, Lysandros Nikolaou, Kumar Aditya, Edgar Margffoy, and many others. Some of these contributors are employed by Meta, which has continued to provide significant engineering resources to support this project.)

2.7 Improved error messages

- The interpreter now provides helpful suggestions when it detects typos in Python keywords. When a word that closely resembles a Python keyword is encountered, the interpreter will suggest the correct keyword in the error message. This feature helps programmers quickly identify and fix common typing mistakes. For example:

```
>>> whille True:
...     pass
Traceback (most recent call last):
  File "<stdin>", line 1
    whille True:
    ^^^^^^
SyntaxError: invalid syntax. Did you mean 'while'?
```

  While the feature focuses on the most common cases, some variations of misspellings may still result in regular syntax errors. (Contributed by Pablo Galindo in gh-132449.)

- `elif` statements that follow an `else` block now have a specific error message. (Contributed by Steele Farnsworth in gh-129902.)

```
>>> if who == "me":
...     print("It's me!")
... else:
...     print("It's not me!")
... elif who is None:
...     print("Who is it?")
File "<stdin>", line 5
  elif who is None:
  ^^^^
SyntaxError: 'elif' block follows an 'else' block
```

- If a statement is passed to the if_expr after `else`, or one of `pass`, `break`, or `continue` is passed before `if`, then the error message highlights where the `expression` is required. (Contributed by Sergey Miryanov in gh-129515.)

```
>>> x = 1 if True else pass
Traceback (most recent call last):
  File "<string>", line 1
    x = 1 if True else pass
                       ^^^^
SyntaxError: expected expression after 'else', but statement is given

>>> x = continue if True else break
Traceback (most recent call last):
  File "<string>", line 1
    x = continue if True else break
        ^^^^^^^^
SyntaxError: expected expression before 'if', but statement is given
```

- When incorrectly closed strings are detected, the error message suggests that the string may be intended to be part of the string. (Contributed by Pablo Galindo in gh-88535.)

```
>>> "The interesting object "The important object" is very important"
Traceback (most recent call last):
SyntaxError: invalid syntax. Is this intended to be part of the string?
```

- When strings have incompatible prefixes, the error now shows which prefixes are incompatible. (Contributed by Nikita Sobolev in gh-133197.)

```
>>> ub'abc'
  File "<python-input-0>", line 1
    ub'abc'
    ^^
SyntaxError: 'u' and 'b' prefixes are incompatible
```

- Improved error messages when using `as` with incompatible targets in:
  - Imports: `import ... as ...`
  - From imports: `from ... import ... as ...`
  - Except handlers: `except ... as ...`
  - Pattern-match cases: `case ... as ...`

  (Contributed by Nikita Sobolev in gh-123539, gh-123562, and gh-123440.)

- Improved error message when trying to add an instance of an unhashable type to a `dict` or `set`. (Contributed by CF Bolz-Tereick and Victor Stinner in gh-132828.)

```
>>> s = set()
>>> s.add({'pages': 12, 'grade': 'A'})
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    s.add({'pages': 12, 'grade': 'A'})
    ~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: cannot use 'dict' as a set element (unhashable type: 'dict')
>>> d = {}
>>> l = [1, 2, 3]
>>> d[l] = 12
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    d[l] = 12
    ~^^^
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
```

- Improved error message when an object supporting the synchronous context manager protocol is entered using `async with` instead of `with`, and vice versa for the asynchronous context manager protocol. (Contributed by Bénédikt Tran in gh-128398.)

2.8 PEP 784: Zstandard support in the standard library

The new `compression` package contains modules `compression.lzma`, `compression.bz2`, `compression.gzip` and `compression.zlib` which re-export the `lzma`, `bz2`, `gzip` and `zlib` modules respectively. The new import names under `compression` are the preferred names for importing these compression modules from Python 3.14. However, the existing modules names have not been deprecated. Any deprecation or removal of the existing compression modules will occur no sooner than five years after the release of 3.14.

The new `compression.zstd` module provides compression and decompression APIs for the Zstandard format via bindings to Meta's zstd library. Zstandard is a widely adopted, highly efficient, and fast compression format. In addition to the APIs introduced in `compression.zstd`, support for reading and writing Zstandard compressed archives has been added to the `tarfile`, `zipfile`, and `shutil` modules.

Here's an example of using the new module to compress some data:

```
from compression import zstd
import math

data = str(math.pi).encode() * 20
compressed = zstd.compress(data)
ratio = len(compressed) / len(data)
print(f"Achieved compression ratio of {ratio}")
```

As can be seen, the API is similar to the APIs of the `lzma` and `bz2` modules.

(Contributed by Emma Harper Smith, Adam Turner, Gregory P. Smith, Tomas Roun, Victor Stinner, and Rogdham in gh-132983.)

See also

PEP 784.

2.9 Asyncio introspection capabilities

Added a new command-line interface to inspect running Python processes using asynchronous tasks, available via `python -m asyncio ps PID` or `python -m asyncio pstree PID`.

The `ps` subcommand inspects the given process ID (PID) and displays information about currently running asyncio tasks. It outputs a task table: a flat listing of all tasks, their names, their coroutine stacks, and which tasks are awaiting them.

The `pstree` subcommand fetches the same information, but instead renders a visual async call tree, showing coroutine relationships in a hierarchical format. This command is particularly useful for debugging long-running or stuck asynchronous programs. It can help developers quickly identify where a program is blocked, what tasks are pending, and how coroutines are chained together.

For example given this code:

```
import asyncio

async def play_track(track):
    await asyncio.sleep(5)
    print(f'? Finished: {track}')

async def play_album(name, tracks):
    async with asyncio.TaskGroup() as tg:
```

(continues on next page)

(continued from previous page)

```
        for track in tracks:
            tg.create_task(play_track(track), name=track)

async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(
          play_album('Sundowning', ['TNDNBTG', 'Levitate']),
          name='Sundowning')
        tg.create_task(
          play_album('TMBTE', ['DYWTYLM', 'Aqua Regia']),
          name='TMBTE')

if __name__ == '__main__':
    asyncio.run(main())
```

Executing the new tool on the running process will yield a table like this:

```
python -m asyncio ps 12345

tid        task id              task name            coroutine stack              ␣
↪                     awaiter chain                                      awaiter␣
↪name    awaiter id
--------------------------------------------------------------------------------
↪--------------------------------------------------------------------------------
↪----------------
1935500    0x7fc930c18050       Task-1               TaskGroup._aexit -> TaskGroup.
↪__aexit__ -> main                                                              ␣
↪       0x0
1935500    0x7fc930c18230       Sundowning           TaskGroup._aexit -> TaskGroup.
↪__aexit__ -> album   TaskGroup._aexit -> TaskGroup.__aexit__ -> main    Task-1  ␣
↪       0x7fc930c18050
1935500    0x7fc93173fa50       TMBTE                TaskGroup._aexit -> TaskGroup.
↪__aexit__ -> album   TaskGroup._aexit -> TaskGroup.__aexit__ -> main    Task-1  ␣
↪       0x7fc930c18050
1935500    0x7fc93173fdf0       TNDNBTG              sleep -> play                ␣
↪                     TaskGroup._aexit -> TaskGroup.__aexit__ -> album  ␣
↪Sundowning      0x7fc930c18230
1935500    0x7fc930d32510       Levitate             sleep -> play                ␣
↪                     TaskGroup._aexit -> TaskGroup.__aexit__ -> album  ␣
↪Sundowning      0x7fc930c18230
1935500    0x7fc930d32890       DYWTYLM              sleep -> play                ␣
↪                     TaskGroup._aexit -> TaskGroup.__aexit__ -> album   TMBTE   ␣
↪       0x7fc93173fa50
1935500    0x7fc93161ec30       Aqua Regia           sleep -> play                ␣
↪                     TaskGroup._aexit -> TaskGroup.__aexit__ -> album   TMBTE   ␣
↪       0x7fc93173fa50
```

or a tree like this:

```
python -m asyncio pstree 12345

└── (T) Task-1
    └── main example.py:13
        └── TaskGroup.__aexit__ Lib/asyncio/taskgroups.py:72
            └── TaskGroup._aexit Lib/asyncio/taskgroups.py:121
                ├── (T) Sundowning
                │   └── album example.py:8
```

(continues on next page)

(continued from previous page)

```
                    │       └── TaskGroup.__aexit__ Lib/asyncio/taskgroups.py:72
                    │           └── TaskGroup._aexit Lib/asyncio/taskgroups.py:121
                    │               ├── (T) TNDNBTG
                    │               │   └── play example.py:4
                    │               │       └── sleep Lib/asyncio/tasks.py:702
                    │               └── (T) Levitate
                    │                   └── play example.py:4
                    │                       └── sleep Lib/asyncio/tasks.py:702
                    └── (T) TMBTE
                        └── album example.py:8
                            └── TaskGroup.__aexit__ Lib/asyncio/taskgroups.py:72
                                └── TaskGroup._aexit Lib/asyncio/taskgroups.py:121
                                    ├── (T) DYWTYLM
                                    │   └── play example.py:4
                                    │       └── sleep Lib/asyncio/tasks.py:702
                                    └── (T) Aqua Regia
                                        └── play example.py:4
                                            └── sleep Lib/asyncio/tasks.py:702
```

If a cycle is detected in the async await graph (which could indicate a programming issue), the tool raises an error and lists the cycle paths that prevent tree construction:

```
python -m asyncio pstree 12345

ERROR: await-graph contains cycles - cannot print a tree!

cycle: Task-2 → Task-3 → Task-2
```

(Contributed by Pablo Galindo, Łukasz Langa, Yury Selivanov, and Marta Gomez Macias in gh-91048.)

2.10 Concurrent safe warnings control

The `warnings.catch_warnings` context manager will now optionally use a context variable for warning filters. This is enabled by setting the `context_aware_warnings` flag, either with the `-X` command-line option or an environment variable. This gives predictable warnings control when using `catch_warnings` combined with multiple threads or asynchronous tasks. The flag defaults to true for the free-threaded build and false for the GIL-enabled build.

(Contributed by Neil Schemenauer and Kumar Aditya in gh-130010.)

3 Other language changes

- All Windows code pages are now supported as 'cpXXX' codecs on Windows. (Contributed by Serhiy Storchaka in gh-123803.)
- Implement mixed-mode arithmetic rules combining real and complex numbers as specified by the C standard since C99. (Contributed by Sergey B Kirpichev in gh-69639.)
- More syntax errors are now detected regardless of optimisation and the `-O` command-line option. This includes writes to `__debug__`, incorrect use of `await`, and asynchronous comprehensions outside asynchronous functions. For example, `python -O -c 'assert (__debug__ := 1)'` or `python -O -c 'assert await 1'` now produce `SyntaxError`s. (Contributed by Irit Katriel and Jelle Zijlstra in gh-122245 & gh-121637.)
- When subclassing a pure C type, the C slots for the new type are no longer replaced with a wrapped version on class creation if they are not explicitly overridden in the subclass. (Contributed by Tomasz Pytel in gh-132284.)

3.1 Built-ins

- The `bytes.fromhex()` and `bytearray.fromhex()` methods now accept ASCII `bytes` and bytes-like objects. (Contributed by Daniel Pope in gh-129349.)
- Add class methods `float.from_number()` and `complex.from_number()` to convert a number to `float` or `complex` type correspondingly. They raise a `TypeError` if the argument is not a real number. (Contributed by Serhiy Storchaka in gh-84978.)
- Support underscore and comma as thousands separators in the fractional part for floating-point presentation types of the new-style string formatting (with `format()` or f-strings). (Contributed by Sergey B Kirpichev in gh-87790.)
- The `int()` function no longer delegates to `__trunc__()`. Classes that want to support conversion to `int()` must implement either `__int__()` or `__index__()`. (Contributed by Mark Dickinson in gh-119743.)
- The `map()` function now has an optional keyword-only *strict* flag like `zip()` to check that all the iterables are of equal length. (Contributed by Wannes Boeykens in gh-119793.)
- The `memoryview` type now supports subscription, making it a generic type. (Contributed by Brian Schubert in gh-126012.)
- Using `NotImplemented` in a boolean context will now raise a `TypeError`. This has raised a `DeprecationWarning` since Python 3.9. (Contributed by Jelle Zijlstra in gh-118767.)
- Three-argument `pow()` now tries calling `__rpow__()` if necessary. Previously it was only called in two-argument `pow()` and the binary power operator. (Contributed by Serhiy Storchaka in gh-130104.)
- `super` objects are now `copyable` and `pickleable`. (Contributed by Serhiy Storchaka in gh-125767.)

3.2 Command line and environment

- The import time flag can now track modules that are already loaded ('cached'), via the new `-X importtime=2`. When such a module is imported, the `self` and `cumulative` times are replaced by the string `cached`.

  Values above `2` for `-X importtime` are now reserved for future use.

  (Contributed by Noah Kim and Adam Turner in gh-118655.)
- The command-line option `-c` now automatically dedents its code argument before execution. The auto-dedentation behavior mirrors `textwrap.dedent()`. (Contributed by Jon Crall and Steven Sun in gh-103998.)
- `-J` is no longer a reserved flag for Jython, and now has no special meaning. (Contributed by Adam Turner in gh-133336.)

3.3 PEP 758: Allow `except` and `except*` expressions without brackets

The `except` and `except*` expressions now allow brackets to be omitted when there are multiple exception types and the `as` clause is not used. For example:

```
try:
    connect_to_server()
except TimeoutError, ConnectionRefusedError:
    print('The network has ceased to be!')
```

(Contributed by Pablo Galindo and Brett Cannon in **PEP 758** and gh-131831.)

3.4 PEP 765: Control flow in `finally` blocks

The compiler now emits a `SyntaxWarning` when a `return`, `break`, or `continue` statement have the effect of leaving a `finally` block. This change is specified in **PEP 765**.

In situations where this change is inconvenient (such as those where the warnings are redundant due to code linting), the warning filter can be used to turn off all syntax warnings by adding `ignore::SyntaxWarning`

as a filter. This can be specified in combination with a filter that converts other warnings to errors (for example, passing `-Werror -Wignore::SyntaxWarning` as CLI options, or setting `PYTHONWARNINGS=error,ignore::SyntaxWarning`).

Note that applying such a filter at runtime using the `warnings` module will only suppress the warning in code that is compiled *after* the filter is adjusted. Code that is compiled prior to the filter adjustment (for example, when a module is imported) will still emit the syntax warning.

(Contributed by Irit Katriel in gh-130080.)

3.5 Incremental garbage collection

The cycle garbage collector is now incremental. This means that maximum pause times are reduced by an order of magnitude or more for larger heaps.

There are now only two generations: young and old. When `gc.collect()` is not called directly, the GC is invoked a little less frequently. When invoked, it collects the young generation and an increment of the old generation, instead of collecting one or more generations.

The behavior of `gc.collect()` changes slightly:

- `gc.collect(1)`: Performs an increment of garbage collection, rather than collecting generation 1.
- Other calls to `gc.collect()` are unchanged.

(Contributed by Mark Shannon in gh-108362.)

3.6 Default interactive shell

- The default interactive shell now highlights Python syntax. The feature is enabled by default, save if `PYTHON_BASIC_REPL` or any other environment variable that disables colour is set. See using-on-controlling-color for details.

  The default color theme for syntax highlighting strives for good contrast and exclusively uses the 4-bit VGA standard ANSI color codes for maximum compatibility. The theme can be customized using an experimental API `_colorize.set_theme()`. This can be called interactively or in the `PYTHONSTARTUP` script. Note that this function has no stability guarantees, and may change or be removed.

  (Contributed by Łukasz Langa in gh-131507.)

- The default interactive shell now supports import auto-completion. This means that typing `import co` and pressing `<Tab>` will suggest modules starting with `co`. Similarly, typing `from concurrent import i` will suggest submodules of `concurrent` starting with `i`. Note that autocompletion of module attributes is not currently supported. (Contributed by Tomas Roun in gh-69605.)

4 New modules

- `annotationlib`: For introspecting annotations. See *PEP 749* for more details. (Contributed by Jelle Zijlstra in gh-119180.)
- `compression` (including `compression.zstd`): A package for compression-related modules, including a new module to support the Zstandard compression format. See *PEP 784* for more details. (Contributed by Emma Harper Smith, Adam Turner, Gregory P. Smith, Tomas Roun, Victor Stinner, and Rogdham in gh-132983.)
- `concurrent.interpreters`: Support for multiple interpreters in the standard library. See *PEP 734* for more details. (Contributed by Eric Snow in gh-134939.)
- `string.templatelib`: Support for template string literals (t-strings). See *PEP 750* for more details. (Contributed by Jim Baker, Guido van Rossum, Paul Everitt, Koudai Aono, Lysandros Nikolaou, Dave Peck, Adam Turner, Jelle Zijlstra, Bénédikt Tran, and Pablo Galindo Salgado in gh-132661.)

5 Improved modules

5.1 argparse

- The default value of the program name for `argparse.ArgumentParser` now reflects the way the Python interpreter was instructed to find the `__main__` module code. (Contributed by Serhiy Storchaka and Alyssa Coghlan in gh-66436.)
- Introduced the optional *suggest_on_error* parameter to `argparse.ArgumentParser`, enabling suggestions for argument choices and subparser names if mistyped by the user. (Contributed by Savannah Ostrowski in gh-124456.)
- Enable color for help text, which can be disabled with the optional *color* parameter to `argparse.ArgumentParser`. This can also be controlled by environment variables. (Contributed by Hugo van Kemenade in gh-130645.)

5.2 ast

- Add `compare()`, a function for comparing two ASTs. (Contributed by Batuhan Taskaya and Jeremy Hylton in gh-60191.)
- Add support for `copy.replace()` for AST nodes. (Contributed by Bénédikt Tran in gh-121141.)
- Docstrings are now removed from an optimized AST in optimization level 2. (Contributed by Irit Katriel in gh-123958.)
- The `repr()` output for AST nodes now includes more information. (Contributed by Tomas Roun in gh-116022.)
- When called with an AST as input, the `parse()` function now always verifies that the root node type is appropriate. (Contributed by Irit Katriel in gh-130139.)
- Add new options to the command-line interface: `--feature-version`, `--optimize`, and `--show-empty`. (Contributed by Semyon Moroz in gh-133367.)

5.3 asyncio

- The function and methods named `create_task()` now take an arbitrary list of keyword arguments. All keyword arguments are passed to the `Task` constructor or the custom task factory. (See `set_task_factory()` for details.) The `name` and `context` keyword arguments are no longer special; the name should now be set using the `name` keyword argument of the factory, and `context` may be `None`.

  This affects the following function and methods: `asyncio.create_task()`, `asyncio.loop.create_task()`, `asyncio.TaskGroup.create_task()`.

  (Contributed by Thomas Grainger in gh-128307.)
- There are two new utility functions for introspecting and printing a program's call graph: `capture_call_graph()` and `print_call_graph()`. See *Asyncio introspection capabilities* for more details. (Contributed by Yury Selivanov, Pablo Galindo Salgado, and Łukasz Langa in gh-91048.)

5.4 calendar

- By default, today's date is highlighted in color in `calendar`'s command-line text output. This can be controlled by environment variables. (Contributed by Hugo van Kemenade in gh-128317.)

5.5 concurrent.futures

- Add a new executor class, `InterpreterPoolExecutor`, which exposes multiple Python interpreters in the same process ('subinterpreters') to Python code. This uses a pool of independent Python interpreters to execute calls asynchronously.

  This is separate from the new `interpreters` module introduced by *PEP 734*. (Contributed by Eric Snow in gh-124548.)

- On Unix platforms other than macOS, 'forkserver' is now the the default start method for `ProcessPoolExecutor` (replacing 'fork'). This change does not affect Windows or macOS, where 'spawn' remains the default start method.

  If the threading incompatible *fork* method is required, you must explicitly request it by supplying a multiprocessing context *mp_context* to `ProcessPoolExecutor`.

  See forkserver restrictions for information and differences with the *fork* method and how this change may affect existing code with mutable global shared variables and/or shared objects that can not be automatically `pickled`.

  (Contributed by Gregory P. Smith in gh-84559.)

- Add two new methods to `ProcessPoolExecutor`, `terminate_workers()` and `kill_workers()`, as ways to terminate or kill all living worker processes in the given pool. (Contributed by Charles Machalow in gh-130849.)

- Add the optional *buffersize* parameter to `Executor.map` to limit the number of submitted tasks whose results have not yet been yielded. If the buffer is full, iteration over the *iterables* pauses until a result is yielded from the buffer. (Contributed by Enzo Bonnal and Josh Rosenberg in gh-74028.)

5.6 configparser

- `configparser` will no longer write config files it cannot read, to improve security. Attempting to `write()` keys containing delimiters or beginning with the section header pattern will raise an `InvalidWriteError`. (Contributed by Jacob Lincoln in gh-129270.)

5.7 contextvars

- Support the context manager protocol for `Token` objects. (Contributed by Andrew Svetlov in gh-129889.)

5.8 ctypes

- The layout of bit fields in `Structure` and `Union` objects is now a closer match to platform defaults (GCC/Clang or MSVC). In particular, fields no longer overlap. (Contributed by Matthias Görgens in gh-97702.)

- The `Structure._layout_` class attribute can now be set to help match a non-default ABI. (Contributed by Petr Viktorin in gh-97702.)

- The class of `Structure`/`Union` field descriptors is now available as `CField`, and has new attributes to aid debugging and introspection. (Contributed by Petr Viktorin in gh-128715.)

- On Windows, the `COMError` exception is now public. (Contributed by Jun Komoda in gh-126686.)

- On Windows, the `CopyComPointer()` function is now public. (Contributed by Jun Komoda in gh-127275.)

- Add `memoryview_at()`, a function to create a `memoryview` object that refers to the supplied pointer and length. This works like `ctypes.string_at()` except it avoids a buffer copy, and is typically useful when implementing pure Python callback functions that are passed dynamically-sized buffers. (Contributed by Rian Hunter in gh-112018.)

- Complex types, `c_float_complex`, `c_double_complex`, and `c_longdouble_complex`, are now available if both the compiler and the `libffi` library support complex C types. (Contributed by Sergey B Kirpichev in gh-61103.)

- Add `ctypes.util.dllist()` for listing the shared libraries loaded by the current process. (Contributed by Brian Ward in gh-119349.)

- Move `ctypes.POINTER()` types cache from a global internal cache (`_pointer_type_cache`) to the `_CData.__pointer_type__` attribute of the corresponding `ctypes` types. This will stop the cache from growing without limits in some situations. (Contributed by Sergey Miryanov in gh-100926.)

- The `py_object` type now supports subscription, making it a generic type. (Contributed by Brian Schubert in gh-132168.)

- `ctypes` now supports free-threading builds. (Contributed by Kumar Aditya and Peter Bierma in gh-127945.)

5.9 curses

- Add the `assume_default_colors()` function, a refinement of the `use_default_colors()` function which allows changing the color pair `0`. (Contributed by Serhiy Storchaka in gh-133139.)

5.10 datetime

- Add the `strptime()` method to the `datetime.date` and `datetime.time` classes. (Contributed by Wannes Boeykens in gh-41431.)

5.11 decimal

- Add `Decimal.from_number()` as an alternative constructor for `Decimal`. (Contributed by Serhiy Storchaka in gh-121798.)
- Expose `IEEEContext()` to support creation of contexts corresponding to the IEEE 754 (2008) decimal interchange formats. (Contributed by Sergey B Kirpichev in gh-53032.)

5.12 difflib

- Comparison pages with highlighted changes generated by the `HtmlDiff` class now support 'dark mode'. (Contributed by Jiahao Li in gh-129939.)

5.13 dis

- Add support for rendering full source location information of `instructions`, rather than only the line number. This feature is added to the following interfaces via the *show_positions* keyword argument:
  - `dis.Bytecode`
  - `dis.dis()`
  - `dis.distb()`
  - `dis.disassemble()`

  This feature is also exposed via `dis --show-positions`. (Contributed by Bénédikt Tran in gh-123165.)
- Add the `dis --specialized` command-line option to show specialized bytecode. (Contributed by Bénédikt Tran in gh-127413.)

5.14 errno

- Add the `EHWPOISON` error code constant. (Contributed by James Roy in gh-126585.)

5.15 faulthandler

- Add support for printing the C stack trace on systems that support it via the new `dump_c_stack()` function or via the *c_stack* argument in `faulthandler.enable()`. (Contributed by Peter Bierma in gh-127604.)

5.16 fnmatch

- Add `filterfalse()`, a function to reject names matching a given pattern. (Contributed by Bénédikt Tran in gh-74598.)

5.17 fractions

- A `Fraction` object may now be constructed from any object with the `as_integer_ratio()` method. (Contributed by Serhiy Storchaka in gh-82017.)
- Add `Fraction.from_number()` as an alternative constructor for `Fraction`. (Contributed by Serhiy Storchaka in gh-121797.)

5.18 functools

- Add the `Placeholder` sentinel. This may be used with the `partial()` or `partialmethod()` functions to reserve a place for positional arguments in the returned partial object. (Contributed by Dominykas Grigonis in gh-119127.)
- Allow the *initial* parameter of `reduce()` to be passed as a keyword argument. (Contributed by Sayandip Dutta in gh-125916.)

5.19 getopt

- Add support for options with optional arguments. (Contributed by Serhiy Storchaka in gh-126374.)
- Add support for returning intermixed options and non-option arguments in order. (Contributed by Serhiy Storchaka in gh-126390.)

5.20 getpass

- Support keyboard feedback in the `getpass()` function via the keyword-only optional argument *echo_char*. Placeholder characters are rendered whenever a character is entered, and removed when a character is deleted. (Contributed by Semyon Moroz in gh-77065.)

5.21 graphlib

- Allow `TopologicalSorter.prepare()` to be called more than once as long as sorting has not started. (Contributed by Daniel Pope in gh-130914.)

5.22 heapq

- The `heapq` module has improved support for working with max-heaps, via the following new functions:
  - `heapify_max()`
  - `heappush_max()`
  - `heappop_max()`
  - `heapreplace_max()`
  - `heappushpop_max()`

5.23 hmac

- Add a built-in implementation for HMAC (**RFC 2104**) using formally verified code from the HACL* project. This implementation is used as a fallback when the OpenSSL implementation of HMAC is not available. (Contributed by Bénédikt Tran in gh-99108.)

5.24 http

- Directory lists and error pages generated by the `http.server` module allow the browser to apply its default dark mode. (Contributed by Yorik Hansen in gh-123430.)
- The `http.server` module now supports serving over HTTPS using the `http.server.HTTPSServer` class. This functionality is exposed by the command-line interface (`python -m http.server`) through the following options:

- `--tls-cert <path>`: Path to the TLS certificate file.
- `--tls-key <path>`: Optional path to the private key file.
- `--tls-password-file <path>`: Optional path to the password file for the private key.

(Contributed by Semyon Moroz in gh-85162.)

5.25 imaplib

- Add `IMAP4.idle()`, implementing the IMAP4 `IDLE` command as defined in **RFC 2177**. (Contributed by Forest in gh-55454.)

5.26 inspect

- `signature()` takes a new argument *annotation_format* to control the `annotationlib.Format` used for representing annotations. (Contributed by Jelle Zijlstra in gh-101552.)
- `Signature.format()` takes a new argument *unquote_annotations*. If true, string annotations are displayed without surrounding quotes. (Contributed by Jelle Zijlstra in gh-101552.)
- Add function `ispackage()` to determine whether an object is a package or not. (Contributed by Zhikang Yan in gh-125634.)

5.27 io

- Reading text from a non-blocking stream with `read` may now raise a `BlockingIOError` if the operation cannot immediately return bytes. (Contributed by Giovanni Siragusa in gh-109523.)
- Add the `Reader` and `Writer` protocols as simpler alternatives to the pseudo-protocols `typing.IO`, `typing.TextIO`, and `typing.BinaryIO`. (Contributed by Sebastian Rittau in gh-127648.)

5.28 json

- Add exception notes for JSON serialization errors that allow identifying the source of the error. (Contributed by Serhiy Storchaka in gh-122163.)
- Allow using the `json` module as a script using the `-m` switch: **`python -m json`**. This is now preferred to **`python -m json.tool`**, which is soft deprecated. See the JSON command-line interface documentation. (Contributed by Trey Hunner in gh-122873.)
- By default, the output of the JSON command-line interface is highlighted in color. This can be controlled by environment variables. (Contributed by Tomas Roun in gh-131952.)

5.29 linecache

- `getline()` can now retrieve source code for frozen modules. (Contributed by Tian Gao in gh-131638.)

5.30 logging.handlers

- `QueueListener` objects now support the context manager protocol. (Contributed by Charles Machalow in gh-132106.)
- `QueueListener.start` now raises a `RuntimeError` if the listener is already started. (Contributed by Charles Machalow in gh-132106.)

5.31 math

- Added more detailed error messages for domain errors in the module. (Contributed by Charlie Zhao and Sergey B Kirpichev in gh-101410.)

5.32 mimetypes

- Add a public command-line for the module, invoked via **`python -m mimetypes`**. (Contributed by Oleg Iarygin and Hugo van Kemenade in gh-93096.)
- Add several new MIME types based on RFCs and common usage:

Microsoft and RFC 8081 MIME types for fonts

- Embedded OpenType: `application/vnd.ms-fontobject`
- OpenType Layout (OTF) `font/otf`
- TrueType: `font/ttf`
- WOFF 1.0 `font/woff`
- WOFF 2.0 `font/woff2`

RFC 9559 MIME types for Matroska audiovisual data container structures

- audio with no video: `audio/matroska` (`.mka`)
- video: `video/matroska` (`.mkv`)
- stereoscopic video: `video/matroska-3d` (`.mk3d`)

Images with RFCs

- **RFC 1494**: CCITT Group 3 (`.g3`)
- **RFC 3362**: Real-time Facsimile, T.38 (`.t38`)
- **RFC 3745**: JPEG 2000 (`.jp2`), extension (`.jpx`) and compound (`.jpm`)
- **RFC 3950**: Tag Image File Format Fax eXtended, TIFF-FX (`.tfx`)
- **RFC 4047**: Flexible Image Transport System (`.fits`)
- **RFC 7903**: Enhanced Metafile (`.emf`) and Windows Metafile (`.wmf`)

Other MIME type additions and changes

- **RFC 2361**: Change type for `.avi` to `video/vnd.avi` and for `.wav` to `audio/vnd.wave`
- **RFC 4337**: Add MPEG-4 `audio/mp4` (`.m4a`)
- **RFC 5334**: Add Ogg media (`.oga`, `.ogg` and `.ogx`)
- **RFC 6713**: Add gzip `application/gzip` (`.gz`)
- **RFC 9639**: Add FLAC `audio/flac` (`.flac`)
- **RFC 9512** `application/yaml` MIME type for YAML files (`.yaml` and `.yml`)
- Add 7z `application/x-7z-compressed` (`.7z`)
- Add Android Package `application/vnd.android.package-archive` (`.apk`) when not strict
- Add deb `application/x-debian-package` (`.deb`)
- Add glTF binary `model/gltf-binary` (`.glb`)
- Add glTF JSON/ASCII `model/gltf+json` (`.gltf`)
- Add M4V `video/x-m4v` (`.m4v`)
- Add PHP `application/x-httpd-php` (`.php`)
- Add RAR `application/vnd.rar` (`.rar`)
- Add RPM `application/x-rpm` (`.rpm`)

- Add STL `model/stl` (`.stl`)
- Add Windows Media Video `video/x-ms-wmv` (`.wmv`)
- De facto: Add WebM `audio/webm` (`.weba`)
- ECMA-376: Add `.docx`, `.pptx` and `.xlsx` types
- OASIS: Add OpenDocument `.odg`, `.odp`, `.ods` and `.odt` types
- W3C: Add EPUB `application/epub+zip` (`.epub`)

(Contributed by Sahil Prajapati and Hugo van Kemenade in gh-84852, by Sasha "Nelie" Chernykh and Hugo van Kemenade in gh-132056, and by Hugo van Kemenade in gh-89416, gh-85957, and gh-129965.)

5.33 multiprocessing

- On Unix platforms other than macOS, 'forkserver' is now the the default start method (replacing 'fork'). This change does not affect Windows or macOS, where 'spawn' remains the default start method.

  If the threading incompatible *fork* method is required, you must explicitly request it via a context from `get_context()` (preferred) or change the default via `set_start_method()`.

  See forkserver restrictions for information and differences with the *fork* method and how this change may affect existing code with mutable global shared variables and/or shared objects that can not be automatically `pickled`.

  (Contributed by Gregory P. Smith in gh-84559.)

- `multiprocessing`'s `'forkserver'` start method now authenticates its control socket to avoid solely relying on filesystem permissions to restrict what other processes could cause the forkserver to spawn workers and run code. (Contributed by Gregory P. Smith for gh-97514.)

- The multiprocessing proxy objects for *list* and *dict* types gain previously overlooked missing methods:
  - `clear()` and `copy()` for proxies of `list`
  - `fromkeys()`, `reversed(d)`, `d | {}`, `{} | d`, `d |= {'b': 2}` for proxies of `dict`

  (Contributed by Roy Hyunjin Han for gh-103134.)

- Add support for shared `set` objects via `SyncManager.set()`. The `set()` in `Manager()` method is now available. (Contributed by Mingyu Park in gh-129949.)

- Add the `interrupt()` to `multiprocessing.Process` objects, which terminates the child process by sending `SIGINT`. This enables `finally` clauses to print a stack trace for the terminated process. (Contributed by Artem Pulkin in gh-131913.)

5.34 operator

- Add `is_none()` and `is_not_none()` as a pair of functions, such that `operator.is_none(obj)` is equivalent to `obj is None` and `operator.is_not_none(obj)` is equivalent to `obj is not None`. (Contributed by Raymond Hettinger and Nico Mexis in gh-115808.)

5.35 os

- Add the `reload_environ()` function to update `os.environ` and `os.environb` with changes to the environment made by `os.putenv()`, by `os.unsetenv()`, or made outside Python in the same process. (Contributed by Victor Stinner in gh-120057.)

- Add the `SCHED_DEADLINE` and `SCHED_NORMAL` constants to the `os` module. (Contributed by James Roy in gh-127688.)

- Add the `readinto()` function to read into a buffer object from a file descriptor. (Contributed by Cody Maloney in gh-129205.)

5.36 os.path

- The *strict* parameter to `realpath()` accepts a new value, `ALLOW_MISSING`. If used, errors other than `FileNotFoundError` will be re-raised; the resulting path can be missing but it will be free of symlinks. (Contributed by Petr Viktorin for **CVE 2025-4517**.)

5.37 pathlib

- Add methods to `pathlib.Path` to recursively copy or move files and directories:
  - `copy()` copies a file or directory tree to a destination.
  - `copy_into()` copies *into* a destination directory.
  - `move()` moves a file or directory tree to a destination.
  - `move_into()` moves *into* a destination directory.

  (Contributed by Barney Gale in gh-73991.)
- Add the `info` attribute, which stores an object implementing the new `pathlib.types.PathInfo` protocol. The object supports querying the file type and internally caching `stat()` results. Path objects generated by `iterdir()` are initialized with file type information gleaned from scanning the parent directory. (Contributed by Barney Gale in gh-125413.)

5.38 pdb

- The `pdb` module now supports remote attaching to a running Python process using a new `-p PID` command-line option:

```
python -m pdb -p 1234
```

  This will connect to the Python process with the given PID and allow you to debug it interactively. Notice that due to how the Python interpreter works attaching to a remote process that is blocked in a system call or waiting for I/O will only work once the next bytecode instruction is executed or when the process receives a signal.

  This feature uses *PEP 768* and the new `sys.remote_exec()` function to attach to the remote process and send the PDB commands to it.

  (Contributed by Matt Wozniski and Pablo Galindo in gh-131591.)
- Hardcoded breakpoints (`breakpoint()` and `set_trace()`) now reuse the most recent `Pdb` instance that calls `set_trace()`, instead of creating a new one each time. As a result, all the instance specific data like `display` and `commands` are preserved across hardcoded breakpoints. (Contributed by Tian Gao in gh-121450.)
- Add a new argument *mode* to `pdb.Pdb`. Disable the `restart` command when `pdb` is in `inline` mode. (Contributed by Tian Gao in gh-123757.)
- A confirmation prompt will be shown when the user tries to quit `pdb` in `inline` mode. `y`, `Y`, `<Enter>` or `EOF` will confirm the quit and call `sys.exit()`, instead of raising `bdb.BdbQuit`. (Contributed by Tian Gao in gh-124704.)
- Inline breakpoints like `breakpoint()` or `pdb.set_trace()` will always stop the program at calling frame, ignoring the `skip` pattern (if any). (Contributed by Tian Gao in gh-130493.)
- `<tab>` at the beginning of the line in `pdb` multi-line input will fill in a 4-space indentation now, instead of inserting a `\t` character. (Contributed by Tian Gao in gh-130471.)
- Auto-indent is introduced in `pdb` multi-line input. It will either keep the indentation of the last line or insert a 4-space indentation when it detects a new code block. (Contributed by Tian Gao in gh-133350.)
- `$_asynctask` is added to access the current asyncio task if applicable. (Contributed by Tian Gao in gh-124367.)

- `pdb.set_trace_async()` is added to support debugging asyncio coroutines. `await` statements are supported with this function. (Contributed by Tian Gao in gh-132576.)
- Source code displayed in `pdb` will be syntax-highlighted. This feature can be controlled using the same methods as the default interactive shell, in addition to the newly added `colorize` argument of `pdb.Pdb`. (Contributed by Tian Gao and Łukasz Langa in gh-133355.)

5.39 pickle

- Set the default protocol version on the `pickle` module to 5. For more details, see pickle protocols.
- Add exception notes for pickle serialization errors that allow identifying the source of the error. (Contributed by Serhiy Storchaka in gh-122213.)

5.40 platform

- Add `invalidate_caches()`, a function to invalidate cached results in the `platform` module. (Contributed by Bénédikt Tran in gh-122549.)

5.41 pydoc

- Annotations in help output are now usually displayed in a format closer to that in the original source. (Contributed by Jelle Zijlstra in gh-101552.)

5.42 re

- Support `\z` as a synonym for `\Z` in `regular expressions`. It is interpreted unambiguously in many other regular expression engines, unlike `\Z`, which has subtly different behavior. (Contributed by Serhiy Storchaka in gh-133306.)
- `\B` in `regular expression` now matches the empty input string, meaning that it is now always the opposite of `\b`. (Contributed by Serhiy Storchaka in gh-124130.)

5.43 socket

- Improve and fix support for Bluetooth sockets.
  - Fix support of Bluetooth sockets on NetBSD and DragonFly BSD. (Contributed by Serhiy Storchaka in gh-132429.)
  - Fix support for `BTPROTO_HCI` on FreeBSD. (Contributed by Victor Stinner in gh-111178.)
  - Add support for `BTPROTO_SCO` on FreeBSD. (Contributed by Serhiy Storchaka in gh-85302.)
  - Add support for *cid* and *bdaddr_type* in the address for `BTPROTO_L2CAP` on FreeBSD. (Contributed by Serhiy Storchaka in gh-132429.)
  - Add support for *channel* in the address for `BTPROTO_HCI` on Linux. (Contributed by Serhiy Storchaka in gh-70145.)
  - Accept an integer as the address for `BTPROTO_HCI` on Linux. (Contributed by Serhiy Storchaka in gh-132099.)
  - Return *cid* in `getsockname()` for `BTPROTO_L2CAP`. (Contributed by Serhiy Storchaka in gh-132429.)
  - Add many new constants. (Contributed by Serhiy Storchaka in gh-132734.)

5.44 ssl

- Indicate through the `HAS_PHA` Boolean whether the `ssl` module supports TLSv1.3 post-handshake client authentication (PHA). (Contributed by Will Childs-Klein in gh-128036.)

5.45 struct

- Support the `float complex` and `double complex` C types in the `struct` module (formatting characters `'F'` and `'D'` respectively). (Contributed by Sergey B Kirpichev in gh-121249.)

5.46 symtable

- Expose the following `Symbol` methods:
  - `is_comp_cell()`
  - `is_comp_iter()`
  - `is_free_class()`

  (Contributed by Bénédikt Tran in gh-120029.)

5.47 sys

- The previously undocumented special function `sys.getobjects()`, which only exists in specialized builds of Python, may now return objects from other interpreters than the one it's called in. (Contributed by Eric Snow in gh-125286.)
- Add `sys._is_immortal()` for determining if an object is immortal. (Contributed by Peter Bierma in gh-128509.)
- On FreeBSD, `sys.platform` no longer contains the major version number. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`. (Contributed by Michael Osipov in gh-129393.)
- Raise `DeprecationWarning` for `sys._clear_type_cache()`. This function was deprecated in Python 3.13 but it didn't raise a runtime warning.
- Add `sys.remote_exec()` to implement the new external debugger interface. See *PEP 768* for details. (Contributed by Pablo Galindo Salgado, Matt Wozniski, and Ivona Stojanovic in gh-131591.)
- Add the `sys._jit` namespace, containing utilities for introspecting just-in-time compilation. (Contributed by Brandt Bucher in gh-133231.)

5.48 sys.monitoring

- Add two new monitoring events, `BRANCH_LEFT` and `BRANCH_RIGHT`. These replace and deprecate the `BRANCH` event. (Contributed by Mark Shannon in gh-122548.)

5.49 sysconfig

- Add `ABIFLAGS` key to `get_config_vars()` on Windows. (Contributed by Xuehai Pan in gh-131799.)

5.50 tarfile

- `data_filter()` now normalizes symbolic link targets in order to avoid path traversal attacks. (Contributed by Petr Viktorin in gh-127987 and **CVE 2025-4138**.)
- `extractall()` now skips fixing up directory attributes when a directory was removed or replaced by another kind of file. (Contributed by Petr Viktorin in gh-127987 and **CVE 2024-12718**.)
- `extract()` and `extractall()` now (re-)apply the extraction filter when substituting a link (hard or symbolic) with a copy of another archive member, and when fixing up directory attributes. The former raises a new exception, `LinkFallbackError`. (Contributed by Petr Viktorin for **CVE 2025-4330** and **CVE 2024-12718**.)
- `extract()` and `extractall()` no longer extract rejected members when `errorlevel()` is zero. (Contributed by Matt Prodani and Petr Viktorin in gh-112887 and **CVE 2025-4435**.)

5.51 threading

- `threading.Thread.start()` now sets the operating system thread name to `threading.Thread.name`. (Contributed by Victor Stinner in gh-59705.)

5.52 tkinter

- Make `tkinter` widget methods `after()` and `after_idle()` accept keyword arguments. (Contributed by Zhikang Yan in gh-126899.)
- Add ability to specify a name for `tkinter.OptionMenu` and `tkinter.ttk.OptionMenu`. (Contributed by Zhikang Yan in gh-130482.)

5.53 turtle

- Add context managers for `turtle.fill()`, `turtle.poly()`, and `turtle.no_animation()`. (Contributed by Marie Roald and Yngve Mardal Moe in gh-126350.)

5.54 types

- `types.UnionType` is now an alias for `typing.Union`. See *below* for more details. (Contributed by Jelle Zijlstra in gh-105499.)

5.55 typing

- The `types.UnionType` and `typing.Union` types are now aliases for each other, meaning that both old-style unions (created with `Union[int, str]`) and new-style unions (`int | str`) now create instances of the same runtime type. This unifies the behavior between the two syntaxes, but leads to some differences in behavior that may affect users who introspect types at runtime:
  - Both syntaxes for creating a union now produce the same string representation in `repr()`. For example, `repr(Union[int, str])` is now `"int | str"` instead of `"typing.Union[int, str]"`.
  - Unions created using the old syntax are no longer cached. Previously, running `Union[int, str]` multiple times would return the same object (`Union[int, str] is Union[int, str]` would be `True`), but now it will return two different objects. Use `==` to compare unions for equality, not `is`. New-style unions have never been cached this way. This change could increase memory usage for some programs that use a large number of unions created by subscripting `typing.Union`. However, several factors offset this cost: unions used in annotations are no longer evaluated by default in Python 3.14 because of **PEP 649**; an instance of `types.UnionType` is itself much smaller than the object returned by `Union[]` was on prior Python versions; and removing the cache also saves some space. It is therefore unlikely that this change will cause a significant increase in memory usage for most users.
  - Previously, old-style unions were implemented using the private class `typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like `get_origin()` and `typing.get_args()` instead of relying on private implementation details.
  - It is now possible to use `typing.Union` itself in `isinstance()` checks. For example, `isinstance(int | str, typing.Union)` will return `True`; previously this raised `TypeError`.
  - The `__args__` attribute of `typing.Union` objects is no longer writable.
  - It is no longer possible to set any attributes on `Union` objects. This only ever worked for dunder attributes on previous versions, was never documented to work, and was subtly broken in many cases.

  (Contributed by Jelle Zijlstra in gh-105499.)
- `TypeAliasType` now supports star unpacking.

5.56 unicodedata

- The Unicode database has been updated to Unicode 16.0.0.

5.57 unittest

- `unittest` output is now colored by default. This can be controlled by environment variables. (Contributed by Hugo van Kemenade in gh-127221.)
- unittest discovery supports namespace package as start directory again. It was removed in Python 3.11. (Contributed by Jacob Walls in gh-80958.)
- A number of new methods were added in the `TestCase` class that provide more specialized tests.
  - `assertHasAttr()` and `assertNotHasAttr()` check whether the object has a particular attribute.
  - `assertIsSubclass()` and `assertNotIsSubclass()` check whether the object is a subclass of a particular class, or of one of a tuple of classes.
  - `assertStartsWith()`, `assertNotStartsWith()`, `assertEndsWith()` and `assertNotEndsWith()` check whether the Unicode or byte string starts or ends with particular strings.

  (Contributed by Serhiy Storchaka in gh-71339.)

5.58 urllib

- Upgrade HTTP digest authentication algorithm for `urllib.request` by supporting SHA-256 digest authentication as specified in **RFC 7616**. (Contributed by Calvin Bui in gh-128193.)
- Improve ergonomics and standards compliance when parsing and emitting `file:` URLs.

  In `url2pathname()`:
  - Accept a complete URL when the new *require_scheme* argument is set to true.
  - Discard URL authority if it matches the local hostname.
  - Discard URL authority if it resolves to a local IP address when the new *resolve_host* argument is set to true.
  - Discard URL query and fragment components.
  - Raise `URLError` if a URL authority isn't local, except on Windows where we return a UNC path as before.

  In `pathname2url()`:
  - Return a complete URL when the new *add_scheme* argument is set to true.
  - Include an empty URL authority when a path begins with a slash. For example, the path `/etc/hosts` is converted to the URL `///etc/hosts`.

  On Windows, drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an `OSError` exception to be raised.

  (Contributed by Barney Gale in gh-125866.)

5.59 uuid

- Add support for UUID versions 6, 7, and 8 via `uuid6()`, `uuid7()`, and `uuid8()` respectively, as specified in **RFC 9562**. (Contributed by Bénédikt Tran in gh-89083.)
- `NIL` and `MAX` are now available to represent the Nil and Max UUID formats as defined by **RFC 9562**. (Contributed by Nick Pope in gh-128427.)
- Allow generating multiple UUIDs simultaneously on the command-line via `python -m uuid --count`. (Contributed by Simon Legner in gh-131236.)

5.60 webbrowser

- Names in the `BROWSER` environment variable can now refer to already registered browsers for the `webbrowser` module, instead of always generating a new browser command.

  This makes it possible to set `BROWSER` to the value of one of the supported browsers on macOS.

5.61 zipfile

- Added `ZipInfo._for_archive`, a method to resolve suitable defaults for a `ZipInfo` object as used by `ZipFile.writestr`. (Contributed by Bénédikt Tran in gh-123424.)
- `ZipFile.writestr()` now respects the `SOURCE_DATE_EPOCH` environment variable in order to better support reproducible builds. (Contributed by Jiahao Li in gh-91279.)

6 Optimizations

- The import time for several standard library modules has been improved, including `annotationlib`, `ast`, `asyncio`, `base64`, `cmd`, `csv`, `gettext`, `importlib.util`, `locale`, `mimetypes`, `optparse`, `pickle`, `pprint`, `pstats`, `shlex`, `socket`, `string`, `subprocess`, `threading`, `tomllib`, `types`, and `zipfile`.

  (Contributed by Adam Turner, Bénédikt Tran, Chris Markiewicz, Eli Schwartz, Hugo van Kemenade, Jelle Zijlstra, and others in gh-118761.)
- The interpreter now avoids some reference count modifications internally when it's safe to do so. This can lead to different values being returned from `sys.getrefcount()` and `Py_REFCNT()` compared to previous versions of Python. See *below* for details.

6.1 asyncio

- Standard benchmark results have improved by 10-20% following the implementation of a new per-thread doubly linked list for `native tasks`, also reducing memory usage. This enables external introspection tools such as *python -m asyncio pstree* to introspect the call graph of asyncio tasks running in all threads. (Contributed by Kumar Aditya in gh-107803.)
- The module now has first class support for free-threading builds. This enables parallel execution of multiple event loops across different threads, scaling linearly with the number of threads. (Contributed by Kumar Aditya in gh-128002.)

6.2 base64

- `b16decode()` is now up to six times faster. (Contributed by Bénédikt Tran, Chris Markiewicz, and Adam Turner in gh-118761.)

6.3 bdb

- The basic debugger now has a `sys.monitoring`-based backend, which can be selected via the passing `'monitoring'` to the `Bdb` class's new *backend* parameter. (Contributed by Tian Gao in gh-124533.)

6.4 difflib

- The `IS_LINE_JUNK()` function is now up to twice as fast. (Contributed by Adam Turner and Semyon Moroz in gh-130167.)

6.5 gc

- The new *incremental garbage collector* means that maximum pause times are reduced by an order of magnitude or more for larger heaps.

  Because of this optimization, the meaning of the results of `get_threshold()` and `set_threshold()` have changed, along with `get_count()` and `get_stats()`.

- For backwards compatibility, `get_threshold()` continues to return a three-item tuple. The first value is the threshold for young collections, as before; the second value determines the rate at which the old collection is scanned (the default is 10, and higher values mean that the old collection is scanned more slowly). The third value is now meaningless and is always zero.
- `set_threshold()` now ignores any items after the second.
- `get_count()` and `get_stats()` continue to return the same format of results. The only difference is that instead of the results referring to the young, aging and old generations, the results refer to the young generation and the aging and collecting spaces of the old generation.

In summary, code that attempted to manipulate the behavior of the cycle GC may not work exactly as intended, but it is very unlikely to be harmful. All other code will work just fine.

(Contributed by Mark Shannon in gh-108362.)

6.6 io

- Opening and reading files now executes fewer system calls. Reading a small operating system cached file in full is up to 15% faster. (Contributed by Cody Maloney and Victor Stinner in gh-120754 and gh-90102.)

6.7 pathlib

- `Path.read_bytes` now uses unbuffered mode to open files, which is between 9% and 17% faster to read in full. (Contributed by Cody Maloney in gh-120754.)

6.8 pdb

- `pdb` now supports two backends, based on either `sys.settrace()` or `sys.monitoring`. Using the pdb CLI or `breakpoint()` will always use the `sys.monitoring` backend. Explicitly instantiating `pdb.Pdb` and its derived classes will use the `sys.settrace()` backend by default, which is configurable. (Contributed by Tian Gao in gh-124533.)

6.9 uuid

- `uuid3()` and `uuid5()` are now both roughly 40% faster for 16-byte names and 20% faster for 1024-byte names. Performance for longer names remains unchanged. (Contributed by Bénédikt Tran in gh-128150.)
- `uuid4()` is now c. 30% faster. (Contributed by Bénédikt Tran in gh-128150.)

6.10 zlib

- On Windows, zlib-ng is now used as the implementation of the `zlib` module in the default binaries. There are no known incompatibilities between `zlib-ng` and the previously-used `zlib` implementation. This should result in better performance at all compression levels.

  It is worth noting that `zlib.Z_BEST_SPEED` (`1`) may result in significantly less compression than the previous implementation, whilst also significantly reducing the time taken to compress.

  (Contributed by Steve Dower in gh-91349.)

7 Removed

7.1 argparse

- Remove the *type*, *choices*, and *metavar* parameters of `BooleanOptionalAction`. These have been deprecated since Python 3.12. (Contributed by Nikita Sobolev in gh-118805.)
- Calling `add_argument_group()` on an argument group now raises a `ValueError`. Similarly, `add_argument_group()` or `add_mutually_exclusive_group()` on a mutually exclusive group now

both raise `ValueError`s. This 'nesting' was never supported, often failed to work correctly, and was unintentionally exposed through inheritance. This functionality has been deprecated since Python 3.11. (Contributed by Savannah Ostrowski in gh-127186.)

7.2 ast

- Remove the following classes, which have been deprecated aliases of `Constant` since Python 3.8 and have emitted deprecation warnings since Python 3.12:
  - `Bytes`
  - `Ellipsis`
  - `NameConstant`
  - `Num`
  - `Str`

  As a consequence of these removals, user-defined `visit_Num`, `visit_Str`, `visit_Bytes`, `visit_NameConstant` and `visit_Ellipsis` methods on custom `NodeVisitor` subclasses will no longer be called when the `NodeVisitor` subclass is visiting an AST. Define a `visit_Constant` method instead.

  (Contributed by Alex Waygood in gh-119562.)
- Remove the following deprecated properties on `ast.Constant`, which were present for compatibility with the now-removed AST classes:
  - `Constant.n`
  - `Constant.s`

  Use `Constant.value` instead. (Contributed by Alex Waygood in gh-119562.)

7.3 asyncio

- Remove the following classes, methods, and functions, which have been deprecated since Python 3.12:
  - `AbstractChildWatcher`
  - `FastChildWatcher`
  - `MultiLoopChildWatcher`
  - `PidfdChildWatcher`
  - `SafeChildWatcher`
  - `ThreadedChildWatcher`
  - `AbstractEventLoopPolicy.get_child_watcher()`
  - `AbstractEventLoopPolicy.set_child_watcher()`
  - `get_child_watcher()`
  - `set_child_watcher()`

  (Contributed by Kumar Aditya in gh-120804.)
- `asyncio.get_event_loop()` now raises a `RuntimeError` if there is no current event loop, and no longer implicitly creates an event loop.

  (Contributed by Kumar Aditya in gh-126353.)

  There's a few patterns that use `asyncio.get_event_loop()`, most of them can be replaced with `asyncio.run()`.

  If you're running an async function, simply use `asyncio.run()`.

  Before:

```
async def main():
    ...


loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

After:

```
async def main():
    ...

asyncio.run(main())
```

If you need to start something, for example, a server listening on a socket and then run forever, use `asyncio.run()` and an `asyncio.Event`.

Before:

```
def start_server(loop): ...

loop = asyncio.get_event_loop()
try:
    start_server(loop)
    loop.run_forever()
finally:
    loop.close()
```

After:

```
def start_server(loop): ...

async def main():
    start_server(asyncio.get_running_loop())
    await asyncio.Event().wait()

asyncio.run(main())
```

If you need to run something in an event loop, then run some blocking code around it, use `asyncio.Runner`.

Before:

```
async def operation_one(): ...
def blocking_code(): ...
async def operation_two(): ...

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(operation_one())
    blocking_code()
    loop.run_until_complete(operation_two())
finally:
    loop.close()
```

After:

```
async def operation_one(): ...
def blocking_code(): ...
async def operation_two(): ...

with asyncio.Runner() as runner:
    runner.run(operation_one())
    blocking_code()
    runner.run(operation_two())
```

7.4 email

- Remove `email.utils.localtime()`'s *isdst* parameter, which was deprecated in and has been ignored since Python 3.12. (Contributed by Hugo van Kemenade in gh-118798.)

7.5 importlib.abc

- Remove deprecated `importlib.abc` classes:
  - `ResourceReader` (use `TraversableResources`)
  - `Traversable` (use `Traversable`)
  - `TraversableResources` (use `TraversableResources`)

  (Contributed by Jason R. Coombs and Hugo van Kemenade in gh-93963.)

7.6 itertools

- Remove support for copy, deepcopy, and pickle operations from `itertools` iterators. These have emitted a `DeprecationWarning` since Python 3.12. (Contributed by Raymond Hettinger in gh-101588.)

7.7 pathlib

- Remove support for passing additional keyword arguments to `Path`. In previous versions, any such arguments are ignored. (Contributed by Barney Gale in gh-74033.)
- Remove support for passing additional positional arguments to `PurePath.relative_to()` and `is_relative_to()`. In previous versions, any such arguments are joined onto *other*. (Contributed by Barney Gale in gh-78707.)

7.8 pkgutil

- Remove the `get_loader()` and `find_loader()` functions, which have been deprecated since Python 3.12. (Contributed by Bénédikt Tran in gh-97850.)

7.9 pty

- Remove the `master_open()` and `slave_open()` functions, which have been deprecated since Python 3.12. Use `pty.openpty()` instead. (Contributed by Nikita Sobolev in gh-118824.)

7.10 sqlite3

- Remove `version` and `version_info` from the `sqlite3` module; use `sqlite_version` and `sqlite_version_info` for the actual version number of the runtime SQLite library. (Contributed by Hugo van Kemenade in gh-118924.)
- Using a sequence of parameters with named placeholders now raises a `ProgrammingError`, having been deprecated since Python 3.12. (Contributed by Erlend E. Aasland in gh-118928 and gh-101693.)

7.11 urllib

- Remove the `Quoter` class from `urllib.parse`, which has been deprecated since Python 3.11. (Contributed by Nikita Sobolev in gh-118827.)
- Remove the `URLopener` and `FancyURLopener` classes from `urllib.request`, which have been deprecated since Python 3.3.

  `myopener.open()` can be replaced with `urlopen()`. `myopener.retrieve()` can be replaced with `urlretrieve()`. Customisations to the opener classes can be replaced by passing customized handlers to `build_opener()`. (Contributed by Barney Gale in gh-84850.)

8 Deprecated

8.1 New deprecations

- Passing a complex number as the *real* or *imag* argument in the `complex()` constructor is now deprecated; complex numbers should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218.)
- `argparse`:
  - Passing the undocumented keyword argument *prefix_chars* to the `add_argument_group()` method is now deprecated. (Contributed by Savannah Ostrowski in gh-125563.)
  - Deprecated the `argparse.FileType` type converter. Anything relating to resource management should be handled downstream, after the arguments have been parsed. (Contributed by Serhiy Storchaka in gh-58032.)
- `asyncio`:
  - The `asyncio.iscoroutinefunction()` is now deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875.)
  - The `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
    * `asyncio.AbstractEventLoopPolicy`
    * `asyncio.DefaultEventLoopPolicy`
    * `asyncio.WindowsSelectorEventLoopPolicy`
    * `asyncio.WindowsProactorEventLoopPolicy`
    * `asyncio.get_event_loop_policy()`
    * `asyncio.set_event_loop_policy()`

    Users should use `asyncio.run()` or `asyncio.Runner` with the *loop_factory* argument to use the desired event loop implementation.

    For example, to use `asyncio.SelectorEventLoop` on Windows:

    ```
    import asyncio

    async def main():
        ...

    asyncio.run(main(), loop_factory=asyncio.SelectorEventLoop)
    ```

    (Contributed by Kumar Aditya in gh-127949.)
- `codecs`: The `codecs.open()` function is now deprecated, and will be removed in a future version of Python. Use `open()` instead. (Contributed by Inada Naoki in gh-133036.)

- ctypes:
  - On non-Windows platforms, setting `Structure._pack_` to use a MSVC-compatible default memory layout is now deprecated in favor of setting `Structure._layout_` to `'ms'`, and will be removed in Python 3.19. (Contributed by Petr Viktorin in gh-131747.)
  - Calling `ctypes.POINTER()` on a string is now deprecated. Use incomplete types for self-referential structures. Also, the internal `ctypes._pointer_type_cache` is deprecated. See `ctypes.POINTER()` for updated implementation details. (Contributed by Sergey Myrianov in gh-100926.)
- `functools`: Calling the Python implementation of `functools.reduce()` with *function* or *sequence* as keyword arguments is now deprecated; the parameters will be made positional-only in Python 3.16. (Contributed by Kirill Podoprigora in gh-121676.)
- `logging`: Support for custom logging handlers with the *strm* argument is now deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032.)
- `mimetypes`: Valid extensions are either empty or must start with '.' for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223.)
- `nturl2path`: This module is now deprecated. Call `urllib.request.url2pathname()` and `pathname2url()` instead. (Contributed by Barney Gale in gh-125866.)
- `os`: The `os.popen()` and `os.spawn*` functions are now soft deprecated. They should no longer be used to write new code. The `subprocess` module is recommended instead. (Contributed by Victor Stinner in gh-120743.)
- `pathlib`: `pathlib.PurePath.as_uri()` is now deprecated and scheduled for removal in Python 3.19. Use `pathlib.Path.as_uri()` instead. (Contributed by Barney Gale in gh-123599.)
- `pdb`: The undocumented `pdb.Pdb.curframe_locals` attribute is now a deprecated read-only property, which will be removed in a future version of Python. The low overhead dynamic frame locals access added in Python 3.13 by **PEP 667** means the frame locals cache reference previously stored in this attribute is no longer needed. Derived debuggers should access `pdb.Pdb.curframe.f_locals` directly in Python 3.13 and later versions. (Contributed by Tian Gao in gh-124369 and gh-125951.)
- `symtable`: Deprecate `symtable.Class.get_methods()` due to the lack of interest, scheduled for removal in Python 3.16. (Contributed by Bénédikt Tran in gh-119698.)
- `tkinter`: The `tkinter.Variable` methods `trace_variable()`, `trace_vdelete()` and `trace_vinfo()` are now deprecated. Use `trace_add()`, `trace_remove()` and `trace_info()` instead. (Contributed by Serhiy Storchaka in gh-120220.)
- `urllib.parse`: Accepting objects with false values (like `0` and `[]`) except empty strings, bytes-like objects and `None` in `parse_qsl()` and `parse_qs()` is now deprecated. (Contributed by Serhiy Storchaka in gh-116897.)

8.2 Pending removal in Python 3.15

- The import system:
  - Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or take into consideration by the import system or standard library. (gh-97879)
  - Setting `__package__` on a module while failing to set `__spec__.parent` is deprecated. In Python 3.15, `__package__` will cease to be set or take into consideration by the import system or standard library. (gh-97879)
- `ctypes`:
  - The undocumented `ctypes.SetPointerType()` function has been deprecated since Python 3.13.
- `http.server`:

  - The obsolete and rarely used `CGIHTTPRequestHandler` has been deprecated since Python 3.13. No direct replacement exists. *Anything* is better than CGI to interface a web server with a request handler.
  - The `--cgi` flag to the **`python -m http.server`** command-line interface has been deprecated since Python 3.13.

- `importlib`:
  - `load_module()` method: use `exec_module()` instead.

- `locale`:
  - The `getdefaultlocale()` function has been deprecated since Python 3.11. Its removal was originally planned for Python 3.13 (gh-90817), but has been postponed to Python 3.15. Use `getlocale()`, `setlocale()`, and `getencoding()` instead. (Contributed by Hugo van Kemenade in gh-111187.)

- `pathlib`:
  - `PurePath.is_reserved()` has been deprecated since Python 3.13. Use `os.path.isreserved()` to detect reserved paths on Windows.

- `platform`:
  - `java_ver()` has been deprecated since Python 3.13. This function is only useful for Jython support, has a confusing API, and is largely untested.

- `sysconfig`:
  - The *check_home* argument of `sysconfig.is_python_build()` has been deprecated since Python 3.12.

- `threading`:
  - `RLock()` will take no arguments in Python 3.15. Passing any arguments has been deprecated since Python 3.14, as the Python version does not permit any arguments, but the C version allows any number of positional or keyword arguments, ignoring every argument.

- `types`:
  - `types.CodeType`: Accessing `co_lnotab` was deprecated in **PEP 626** since 3.10 and was planned to be removed in 3.12, but it only got a proper `DeprecationWarning` in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866.)

- `typing`:
  - The undocumented keyword argument syntax for creating `NamedTuple` classes (for example, `Point = NamedTuple("Point", x=int, y=int)`) has been deprecated since Python 3.13. Use the class-based syntax or the functional syntax instead.
  - When using the functional syntax of `TypedDict`s, failing to pass a value to the *fields* parameter (`TD = TypedDict("TD")`) or passing `None` (`TD = TypedDict("TD", None)`) has been deprecated since Python 3.13. Use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})` to create a TypedDict with zero field.
  - The `typing.no_type_check_decorator()` decorator function has been deprecated since Python 3.13. After eight years in the `typing` module, it has yet to be supported by any major type checker.

- `wave`:
  - The `getmark()`, `setmark()`, and `getmarkers()` methods of the `Wave_read` and `Wave_write` classes have been deprecated since Python 3.13.

- `zipimport`:
  - `load_module()` has been deprecated since Python 3.10. Use `exec_module()` instead. (Contributed by Jiahao Li in gh-125746.)

8.3 Pending removal in Python 3.16

- The import system:
  - Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.
- `array`:
  - The `'u'` format code (`wchar_t`) has been deprecated in documentation since Python 3.3 and at runtime since Python 3.13. Use the `'w'` format code (`Py_UCS4`) for Unicode characters instead.
- `asyncio`:
  - `asyncio.iscoroutinefunction()` is deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875.)
  - `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
    * `asyncio.AbstractEventLoopPolicy`
    * `asyncio.DefaultEventLoopPolicy`
    * `asyncio.WindowsSelectorEventLoopPolicy`
    * `asyncio.WindowsProactorEventLoopPolicy`
    * `asyncio.get_event_loop_policy()`
    * `asyncio.set_event_loop_policy()`

    Users should use `asyncio.run()` or `asyncio.Runner` with *loop_factory* to use the desired event loop implementation.

    For example, to use `asyncio.SelectorEventLoop` on Windows:

    ```
    import asyncio

    async def main():
        ...

    asyncio.run(main(), loop_factory=asyncio.SelectorEventLoop)
    ```

    (Contributed by Kumar Aditya in gh-127949.)
- `builtins`:
  - Bitwise inversion on boolean types, `~True` or `~False` has been deprecated since Python 3.12, as it produces surprising and unintuitive results (`-2` and `-1`). Use `not x` instead for the logical negation of a Boolean. In the rare case that you need the bitwise inversion of the underlying integer, convert to `int` explicitly (`~int(x)`).
- `functools`:
  - Calling the Python implementation of `functools.reduce()` with *function* or *sequence* as keyword arguments has been deprecated since Python 3.14.
- `logging`:

  Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032.)
- `mimetypes`:
  - Valid extensions start with a '.' or are empty for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223.)

- shutil:
  - The ExecError exception has been deprecated since Python 3.14. It has not been used by any function in shutil since Python 3.4, and is now an alias of RuntimeError.
- symtable:
  - The Class.get_methods method has been deprecated since Python 3.14.
- sys:
  - The _enablelegacywindowsfsencoding() function has been deprecated since Python 3.13. Use the PYTHONLEGACYWINDOWSFSENCODING environment variable instead.
- sysconfig:
  - The sysconfig.expand_makefile_vars() function has been deprecated since Python 3.14. Use the vars argument of sysconfig.get_paths() instead.
- tarfile:
  - The undocumented and unused TarFile.tarfile attribute has been deprecated since Python 3.13.

8.4 Pending removal in Python 3.17

- collections.abc:
  - collections.abc.ByteString is scheduled for removal in Python 3.17.

    Use isinstance(obj, collections.abc.Buffer) to test if obj implements the buffer protocol at runtime. For use in type annotations, either use Buffer or a union that explicitly specifies the types your code supports (e.g., bytes | bytearray | memoryview).

    ByteString was originally intended to be an abstract class that would serve as a supertype of both bytes and bytearray. However, since the ABC never had any methods, knowing that an object was an instance of ByteString never actually told you anything useful about the object. Other common buffer types such as memoryview were also never understood as subtypes of ByteString (either at runtime or by static type checkers).

    See **PEP 688** for more details. (Contributed by Shantanu Jain in gh-91896.)
- typing:
  - Before Python 3.14, old-style unions were implemented using the private class typing._UnionGenericAlias. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like typing.get_origin() and typing.get_args() instead of relying on private implementation details.
  - typing.ByteString, deprecated since Python 3.9, is scheduled for removal in Python 3.17.

    Use isinstance(obj, collections.abc.Buffer) to test if obj implements the buffer protocol at runtime. For use in type annotations, either use Buffer or a union that explicitly specifies the types your code supports (e.g., bytes | bytearray | memoryview).

    ByteString was originally intended to be an abstract class that would serve as a supertype of both bytes and bytearray. However, since the ABC never had any methods, knowing that an object was an instance of ByteString never actually told you anything useful about the object. Other common buffer types such as memoryview were also never understood as subtypes of ByteString (either at runtime or by static type checkers).

    See **PEP 688** for more details. (Contributed by Shantanu Jain in gh-91896.)

8.5 Pending removal in Python 3.19

- ctypes:
  - Implicitly switching to the MSVC-compatible struct layout by setting _pack_ but not _layout_ on non-Windows platforms.

8.6 Pending removal in future versions

The following APIs will be removed in the future, although there is currently no date scheduled for their removal.

- argparse:
  - Nesting argument groups and nesting mutually exclusive groups are deprecated.
  - Passing the undocumented keyword argument *prefix_chars* to add_argument_group() is now deprecated.
  - The argparse.FileType type converter is deprecated.
- builtins:
  - Generators: throw(type, exc, tb) and athrow(type, exc, tb) signature is deprecated: use throw(exc) and athrow(exc) instead, the single argument signature.
  - Currently Python accepts numeric literals immediately followed by keywords, for example 0in x, 1or x, 0if 1else 2. It allows confusing and ambiguous expressions like [0x1for x in y] (which can be interpreted as [0x1 for x in y] or [0x1f or x in y]). A syntax warning is raised if the numeric literal is immediately followed by one of keywords and, else, for, if, in, is and or. In a future release it will be changed to a syntax error. (gh-87999)
  - Support for __index__() and __int__() method returning non-int type: these methods will be required to return an instance of a strict subclass of int.
  - Support for __float__() method returning a strict subclass of float: these methods will be required to return an instance of float.
  - Support for __complex__() method returning a strict subclass of complex: these methods will be required to return an instance of complex.
  - Delegation of int() to __trunc__() method.
  - Passing a complex number as the *real* or *imag* argument in the complex() constructor is now deprecated; it should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218.)
- calendar: calendar.January and calendar.February constants are deprecated and replaced by calendar.JANUARY and calendar.FEBRUARY. (Contributed by Prince Roshan in gh-103636.)
- codecs: use open() instead of codecs.open(). (gh-133038)
- codeobject.co_lnotab: use the codeobject.co_lines() method instead.
- datetime:
  - utcnow(): use datetime.datetime.now(tz=datetime.UTC).
  - utcfromtimestamp(): use datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC).
- gettext: Plural value must be an integer.
- importlib:
  - cache_from_source() *debug_override* parameter is deprecated: use the *optimization* parameter instead.
- importlib.metadata:
  - EntryPoints tuple interface.
  - Implicit None on return values.

- `logging`: the `warn()` method has been deprecated since Python 3.3, use `warning()` instead.
- `mailbox`: Use of StringIO input and text mode is deprecated, use BytesIO and binary mode instead.
- `os`: Calling `os.register_at_fork()` in multi-threaded process.
- `pydoc.ErrorDuringImport`: A tuple value for *exc_info* parameter is deprecated, use an exception instance.
- `re`: More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760.)
- `sre_compile`, `sre_constants` and `sre_parse` modules.
- `shutil`: `rmtree()`'s *onerror* parameter is deprecated in Python 3.12; use the *onexc* parameter instead.
- `ssl` options and protocols:
  - `ssl.SSLContext` without protocol argument is deprecated.
  - `ssl.SSLContext`: `set_npn_protocols()` and `selected_npn_protocol()` are deprecated: use ALPN instead.
  - `ssl.OP_NO_SSL*` options
  - `ssl.OP_NO_TLS*` options
  - `ssl.PROTOCOL_SSLv3`
  - `ssl.PROTOCOL_TLS`
  - `ssl.PROTOCOL_TLSv1`
  - `ssl.PROTOCOL_TLSv1_1`
  - `ssl.PROTOCOL_TLSv1_2`
  - `ssl.TLSVersion.SSLv3`
  - `ssl.TLSVersion.TLSv1`
  - `ssl.TLSVersion.TLSv1_1`
- `threading` methods:
  - `threading.Condition.notifyAll()`: use `notify_all()`.
  - `threading.Event.isSet()`: use `is_set()`.
  - `threading.Thread.isDaemon()`, `threading.Thread.setDaemon()`: use `threading.Thread.daemon` attribute.
  - `threading.Thread.getName()`, `threading.Thread.setName()`: use `threading.Thread.name` attribute.
  - `threading.currentThread()`: use `threading.current_thread()`.
  - `threading.activeCount()`: use `threading.active_count()`.
- `typing.Text` (gh-92332).
- The internal class `typing._UnionGenericAlias` is no longer used to implement `typing.Union`. To preserve compatibility with users using this private class, a compatibility shim will be provided until at least Python 3.17. (Contributed by Jelle Zijlstra in gh-105499.)
- `unittest.IsolatedAsyncioTestCase`: it is deprecated to return a value that is not `None` from a test case.
- `urllib.parse` deprecated functions: `urlparse()` instead
  - `splitattr()`
  - `splithost()`

- splitnport()
- splitpasswd()
- splitport()
- splitquery()
- splittag()
- splittype()
- splituser()
- splitvalue()
- to_bytes()

- wsgiref: SimpleHandler.stdout.write() should not do partial writes.
- xml.etree.ElementTree: Testing the truth value of an Element is deprecated. In a future release it will always return True. Prefer explicit len(elem) or elem is not None tests instead.
- sys._clear_type_cache() is deprecated: use sys._clear_internal_caches() instead.

9 CPython bytecode changes

- Replaced the opcode BINARY_SUBSCR by the BINARY_OP opcode with the NB_SUBSCR oparg. (Contributed by Irit Katriel in gh-100239.)
- Add the BUILD_INTERPOLATION and BUILD_TEMPLATE opcodes to construct new Interpolation and Template instances, respectively. (Contributed by Lysandros Nikolaou and others in gh-132661; see also *PEP 750: Template strings*).
- Remove the BUILD_CONST_KEY_MAP opcode. Use BUILD_MAP instead. (Contributed by Mark Shannon in gh-122160.)
- Replace the LOAD_ASSERTION_ERROR opcode with LOAD_COMMON_CONSTANT and add support for loading NotImplementedError.
- Add the LOAD_FAST_BORROW and LOAD_FAST_BORROW_LOAD_FAST_BORROW opcodes to reduce reference counting overhead when the interpreter can prove that the reference in the frame outlives the reference loaded onto the stack. (Contributed by Matt Page in gh-130704.)
- Add the LOAD_SMALL_INT opcode, which pushes a small integer equal to the oparg to the stack. The RETURN_CONST opcode is removed as it is no longer used. (Contributed by Mark Shannon in gh-125837.)
- Add the new LOAD_SPECIAL instruction. Generate code for with and async with statements using the new instruction. Removed the BEFORE_WITH and BEFORE_ASYNC_WITH instructions. (Contributed by Mark Shannon in gh-120507.)
- Add the POP_ITER opcode to support 'virtual' iterators. (Contributed by Mark Shannon in gh-132554.)

9.1 Pseudo-instructions

- Add the ANNOTATIONS_PLACEHOLDER pseudo instruction to support partially executed module-level annotations with *deferred evaluation of annotations*. (Contributed by Jelle Zijlstra in gh-130907.)
- Add the BINARY_OP_EXTEND pseudo instruction, which executes a pair of functions (guard and specialization functions) accessed from the inline cache. (Contributed by Irit Katriel in gh-100239.)
- Add three specializations for CALL_KW; CALL_KW_PY for calls to Python functions, CALL_KW_BOUND_METHOD for calls to bound methods, and CALL_KW_NON_PY for all other calls. (Contributed by Mark Shannon in gh-118093.)
- Add the JUMP_IF_TRUE and JUMP_IF_FALSE pseudo instructions, conditional jumps which do not impact the stack. Replaced by the sequence COPY 1, TO_BOOL, POP_JUMP_IF_TRUE/FALSE. (Contributed by Irit Katriel in gh-124285.)

- Add the `LOAD_CONST_MORTAL` pseudo instruction. (Contributed by Mark Shannon in gh-128685.)
- Add the `LOAD_CONST_IMMORTAL` pseudo instruction, which does the same as `LOAD_CONST`, but is more efficient for immortal objects. (Contributed by Mark Shannon in gh-125837.)
- Add the `NOT_TAKEN` pseudo instruction, used by `sys.monitoring` to record branch events (such as `BRANCH_LEFT`). (Contributed by Mark Shannon in gh-122548.)

10 C API changes

10.1 Python configuration C API

Add a PyInitConfig C API to configure the Python initialization without relying on C structures and the ability to make ABI-compatible changes in the future.

Complete the **PEP 587** PyConfig C API by adding `PyInitConfig_AddModule()` which can be used to add a built-in extension module; a feature previously referred to as the "inittab".

Add `PyConfig_Get()` and `PyConfig_Set()` functions to get and set the current runtime configuration.

PEP 587 'Python Initialization Configuration' unified all the ways to configure Python's initialization. This PEP also unifies the configuration of Python's preinitialization and initialization in a single API. Moreover, this PEP only provides a single choice to embed Python, instead of having two 'Python' and 'Isolated' choices (PEP 587), to further simplify the API.

The lower level PEP 587 PyConfig API remains available for use cases with an intentionally higher level of coupling to CPython implementation details (such as emulating the full functionality of CPython's CLI, including its configuration mechanisms).

(Contributed by Victor Stinner in gh-107954.)

See also

PEP 741 and **PEP 587**

10.2 New features in the C API

- Add `Py_PACK_VERSION()` and `Py_PACK_FULL_VERSION()`, two new macros for bit-packing Python version numbers. This is useful for comparisons with `Py_Version` or `PY_VERSION_HEX`. (Contributed by Petr Viktorin in gh-128629.)
- Add `PyBytes_Join(sep, iterable)` function, similar to `sep.join(iterable)` in Python. (Contributed by Victor Stinner in gh-121645.)
- Add functions to manipulate the configuration of the current runtime Python interpreter (*PEP 741: Python configuration C API*):
  - `PyConfig_Get()`
  - `PyConfig_GetInt()`
  - `PyConfig_Set()`
  - `PyConfig_Names()`

  (Contributed by Victor Stinner in gh-107954.)
- Add functions to configure Python initialization (*PEP 741: Python configuration C API*):
  - `Py_InitializeFromInitConfig()`
  - `PyInitConfig_AddModule()`
  - `PyInitConfig_Create()`
  - `PyInitConfig_Free()`

  - `PyInitConfig_FreeStrList()`
  - `PyInitConfig_GetError()`
  - `PyInitConfig_GetExitCode()`
  - `PyInitConfig_GetInt()`
  - `PyInitConfig_GetStr()`
  - `PyInitConfig_GetStrList()`
  - `PyInitConfig_HasOption()`
  - `PyInitConfig_SetInt()`
  - `PyInitConfig_SetStr()`
  - `PyInitConfig_SetStrList()`

  (Contributed by Victor Stinner in gh-107954.)

- Add `Py_fopen()` function to open a file. This works similarly to the standard C `fopen()` function, instead accepting a Python object for the *path* parameter and setting an exception on error. The corresponding new `Py_fclose()` function should be used to close a file. (Contributed by Victor Stinner in gh-127350.)
- Add `Py_HashBuffer()` to compute and return the hash value of a buffer. (Contributed by Antoine Pitrou and Victor Stinner in gh-122854.)
- Add `PyImport_ImportModuleAttr()` and `PyImport_ImportModuleAttrString()` helper functions to import a module and get an attribute of the module. (Contributed by Victor Stinner in gh-128911.)
- Add `PyIter_NextItem()` to replace `PyIter_Next()`, which has an ambiguous return value. (Contributed by Irit Katriel and Erlend Aasland in gh-105201.)
- Add `PyLong_GetSign()` function to get the sign of `int` objects. (Contributed by Sergey B Kirpichev in gh-116560.)
- Add `PyLong_IsPositive()`, `PyLong_IsNegative()` and `PyLong_IsZero()` for checking if `PyLongObject` is positive, negative, or zero, respectively. (Contributed by James Roy and Sergey B Kirpichev in gh-126061.)
- Add new functions to convert C `<stdint.h>` numbers to/from Python `int` objects:
  - `PyLong_AsInt32()`
  - `PyLong_AsInt64()`
  - `PyLong_AsUInt32()`
  - `PyLong_AsUInt64()`
  - `PyLong_FromInt32()`
  - `PyLong_FromInt64()`
  - `PyLong_FromUInt32()`
  - `PyLong_FromUInt64()`

  (Contributed by Victor Stinner in gh-120389.)

- Add a new import and export API for Python `int` objects (**PEP 757**):
  - `PyLong_GetNativeLayout()`
  - `PyLong_Export()`
  - `PyLong_FreeExport()`
  - `PyLongWriter_Create()`
  - `PyLongWriter_Finish()`
  - `PyLongWriter_Discard()`

(Contributed by Sergey B Kirpichev and Victor Stinner in gh-102471.)

- Add `PyMonitoring_FireBranchLeftEvent()` and `PyMonitoring_FireBranchRightEvent()` for generating `BRANCH_LEFT` and `BRANCH_RIGHT` events, respectively. (Contributed by Mark Shannon in gh-122548.)
- Add `PyType_Freeze()` function to make a type immutable. (Contributed by Victor Stinner in gh-121654.)
- Add `PyType_GetBaseByToken()` and `Py_tp_token` slot for easier superclass identification, which attempts to resolve the type checking issue mentioned in **PEP 630**. (Contributed in gh-124153.)
- Add a new `PyUnicode_Equal()` function to test if two strings are equal. The function is also added to the Limited C API. (Contributed by Victor Stinner in gh-124502.)
- Add a new `PyUnicodeWriter` API to create a Python `str` object, with the following functions:
  - `PyUnicodeWriter_Create()`
  - `PyUnicodeWriter_DecodeUTF8Stateful()`
  - `PyUnicodeWriter_Discard()`
  - `PyUnicodeWriter_Finish()`
  - `PyUnicodeWriter_Format()`
  - `PyUnicodeWriter_WriteASCII()`
  - `PyUnicodeWriter_WriteChar()`
  - `PyUnicodeWriter_WriteRepr()`
  - `PyUnicodeWriter_WriteStr()`
  - `PyUnicodeWriter_WriteSubstring()`
  - `PyUnicodeWriter_WriteUCS4()`
  - `PyUnicodeWriter_WriteUTF8()`
  - `PyUnicodeWriter_WriteWideChar()`

  (Contributed by Victor Stinner in gh-119182.)
- The `k` and `K` formats in `PyArg_ParseTuple()` and similar functions now use `__index__()` if available, like all other integer formats. (Contributed by Serhiy Storchaka in gh-112068.)
- Add support for a new `p` format unit in `Py_BuildValue()` that produces a Python `bool` object from a C integer. (Contributed by Pablo Galindo in bpo-45325.)
- Add `PyUnstable_IsImmortal()` for determining if an object is immortal, for debugging purposes. (Contributed by Peter Bierma in gh-128509.)
- Add `PyUnstable_Object_EnableDeferredRefcount()` for enabling deferred reference counting, as outlined in **PEP 703**.
- Add `PyUnstable_Object_IsUniquelyReferenced()` as a replacement for `Py_REFCNT(op) == 1` on free threaded builds. (Contributed by Peter Bierma in gh-133140.)
- Add `PyUnstable_Object_IsUniqueReferencedTemporary()` to determine if an object is a unique temporary object on the interpreter's operand stack. This can be used in some cases as a replacement for checking if `Py_REFCNT()` is `1` for Python objects passed as arguments to C API functions. (Contributed by Sam Gross in gh-133164.)

10.3 Limited C API changes

- In the limited C API version 3.14 and newer, `Py_TYPE()` and `Py_REFCNT()` are now implemented as an opaque function call to hide implementation details. (Contributed by Victor Stinner in gh-120600 and gh-124127.)

- Remove the `PySequence_Fast_GET_SIZE`, `PySequence_Fast_GET_ITEM`, and `PySequence_Fast_ITEMS` macros from the limited C API, since they have always been broken in the limited C API. (Contributed by Victor Stinner in gh-91417.)

10.4 Removed C APIs

- Creating `immutable types` with mutable bases was deprecated in Python 3.12, and now raises a `TypeError`. (Contributed by Nikita Sobolev in gh-119775.)
- Remove `PyDictObject.ma_version_tag` member, which was deprecated in Python 3.12. Use the `PyDict_AddWatcher()` API instead. (Contributed by Sam Gross in gh-124296.)
- Remove the private `_Py_InitializeMain()` function. It was a provisional API added to Python 3.8 by **PEP 587**. (Contributed by Victor Stinner in gh-129033.)
- Remove the undocumented APIs `Py_C_RECURSION_LIMIT` and `PyThreadState.c_recursion_remaining`. These were added in 3.13 and have been removed without deprecation. Use `Py_EnterRecursiveCall()` to guard against runaway recursion in C code. (Removed by Petr Viktorin in gh-133079, see also gh-130396.)

10.5 Deprecated C APIs

- The `Py_HUGE_VAL` macro is now soft deprecated. Use `Py_INFINITY` instead. (Contributed by Sergey B Kirpichev in gh-120026.)
- The `Py_IS_NAN`, `Py_IS_INFINITY`, and `Py_IS_FINITE` macros are now soft deprecated. Use `isnan`, `isinf` and `isfinite` instead, available from `math.h` since C99. (Contributed by Sergey B Kirpichev in gh-119613.)
- Non-tuple sequences are now deprecated as argument for the `(items)` format unit in `PyArg_ParseTuple()` and other argument parsing functions if *items* contains format units which store a borrowed buffer or a borrowed reference. (Contributed by Serhiy Storchaka in gh-50333.)
- The `_PyMonitoring_FireBranchEvent` function is now deprecated and should be replaced with calls to `PyMonitoring_FireBranchLeftEvent()` and `PyMonitoring_FireBranchRightEvent()`.
- The previously undocumented function `PySequence_In()` is now soft deprecated. Use `PySequence_Contains()` instead. (Contributed by Yuki Kobayashi in gh-127896.)

Pending removal in Python 3.15

- The `PyImport_ImportModuleNoBlock()`: Use `PyImport_ImportModule()` instead.
- `PyWeakref_GetObject()` and `PyWeakref_GET_OBJECT()`: Use `PyWeakref_GetRef()` instead. The pythoncapi-compat project can be used to get `PyWeakref_GetRef()` on Python 3.12 and older.
- `Py_UNICODE` type and the `Py_UNICODE_WIDE` macro: Use `wchar_t` instead.
- `PyUnicode_AsDecodedObject()`: Use `PyCodec_Decode()` instead.
- `PyUnicode_AsDecodedUnicode()`: Use `PyCodec_Decode()` instead; Note that some codecs (for example, "base64") may return a type other than `str`, such as `bytes`.
- `PyUnicode_AsEncodedObject()`: Use `PyCodec_Encode()` instead.
- `PyUnicode_AsEncodedUnicode()`: Use `PyCodec_Encode()` instead; Note that some codecs (for example, "base64") may return a type other than `bytes`, such as `str`.
- Python initialization functions, deprecated in Python 3.13:
  - `Py_GetPath()`: Use `PyConfig_Get("module_search_paths")` (`sys.path`) instead.
  - `Py_GetPrefix()`: Use `PyConfig_Get("base_prefix")` (`sys.base_prefix`) instead. Use `PyConfig_Get("prefix")` (`sys.prefix`) if virtual environments need to be handled.

- `Py_GetExecPrefix()`: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`) if virtual environments need to be handled.
- `Py_GetProgramFullPath()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
- `Py_GetProgramName()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
- `Py_GetPythonHome()`: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

The pythoncapi-compat project can be used to get `PyConfig_Get()` on Python 3.13 and older.

- Functions to configure Python's initialization, deprecated in Python 3.11:
  - `PySys_SetArgvEx()`: Set `PyConfig.argv` instead.
  - `PySys_SetArgv()`: Set `PyConfig.argv` instead.
  - `Py_SetProgramName()`: Set `PyConfig.program_name` instead.
  - `Py_SetPythonHome()`: Set `PyConfig.home` instead.
  - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and `warnings.filters` instead.

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead.
- Global configuration variables:
  - `Py_DebugFlag`: Use `PyConfig.parser_debug` or `PyConfig_Get("parser_debug")` instead.
  - `Py_VerboseFlag`: Use `PyConfig.verbose` or `PyConfig_Get("verbose")` instead.
  - `Py_QuietFlag`: Use `PyConfig.quiet` or `PyConfig_Get("quiet")` instead.
  - `Py_InteractiveFlag`: Use `PyConfig.interactive` or `PyConfig_Get("interactive")` instead.
  - `Py_InspectFlag`: Use `PyConfig.inspect` or `PyConfig_Get("inspect")` instead.
  - `Py_OptimizeFlag`: Use `PyConfig.optimization_level` or `PyConfig_Get("optimization_level")` instead.
  - `Py_NoSiteFlag`: Use `PyConfig.site_import` or `PyConfig_Get("site_import")` instead.
  - `Py_BytesWarningFlag`: Use `PyConfig.bytes_warning` or `PyConfig_Get("bytes_warning")` instead.
  - `Py_FrozenFlag`: Use `PyConfig.pathconfig_warnings` or `PyConfig_Get("pathconfig_warnings")` instead.
  - `Py_IgnoreEnvironmentFlag`: Use `PyConfig.use_environment` or `PyConfig_Get("use_environment")` instead.
  - `Py_DontWriteBytecodeFlag`: Use `PyConfig.write_bytecode` or `PyConfig_Get("write_bytecode")` instead.
  - `Py_NoUserSiteDirectory`: Use `PyConfig.user_site_directory` or `PyConfig_Get("user_site_directory")` instead.
  - `Py_UnbufferedStdioFlag`: Use `PyConfig.buffered_stdio` or `PyConfig_Get("buffered_stdio")` instead.
  - `Py_HashRandomizationFlag`: Use `PyConfig.use_hash_seed` and `PyConfig.hash_seed` or `PyConfig_Get("hash_seed")` instead.
  - `Py_IsolatedFlag`: Use `PyConfig.isolated` or `PyConfig_Get("isolated")` instead.
  - `Py_LegacyWindowsFSEncodingFlag`: Use `PyPreConfig.legacy_windows_fs_encoding` or `PyConfig_Get("legacy_windows_fs_encoding")` instead.
  - `Py_LegacyWindowsStdioFlag`: Use `PyConfig.legacy_windows_stdio` or `PyConfig_Get("legacy_windows_stdio")` instead.

- `Py_FileSystemDefaultEncoding`, `Py_HasFileSystemDefaultEncoding`: Use `PyConfig.filesystem_encoding` or `PyConfig_Get("filesystem_encoding")` instead.
- `Py_FileSystemDefaultEncodeErrors`: Use `PyConfig.filesystem_errors` or `PyConfig_Get("filesystem_errors")` instead.
- `Py_UTF8Mode`: Use `PyPreConfig.utf8_mode` or `PyConfig_Get("utf8_mode")` instead. (see `Py_PreInitialize()`)

The `Py_InitializeFromConfig()` API should be used with `PyConfig` to set these options. Or `PyConfig_Get()` can be used to get these options at runtime.

Pending removal in Python 3.16

- The bundled copy of `libmpdec`.

Pending removal in Python 3.18

- The following private functions are deprecated and planned for removal in Python 3.18:
  - `_PyBytes_Join()`: use `PyBytes_Join()`.
  - `_PyDict_GetItemStringWithError()`: use `PyDict_GetItemStringRef()`.
  - `_PyDict_Pop()`: use `PyDict_Pop()`.
  - `_PyLong_Sign()`: use `PyLong_GetSign()`.
  - `_PyLong_FromDigits()` and `_PyLong_New()`: use `PyLongWriter_Create()`.
  - `_PyThreadState_UncheckedGet()`: use `PyThreadState_GetUnchecked()`.
  - `_PyUnicode_AsString()`: use `PyUnicode_AsUTF8()`.
  - `_PyUnicodeWriter_Init()`: replace `_PyUnicodeWriter_Init(&writer)` with `writer = PyUnicodeWriter_Create(0)`.
  - `_PyUnicodeWriter_Finish()`: replace `_PyUnicodeWriter_Finish(&writer)` with `PyUnicodeWriter_Finish(writer)`.
  - `_PyUnicodeWriter_Dealloc()`: replace `_PyUnicodeWriter_Dealloc(&writer)` with `PyUnicodeWriter_Discard(writer)`.
  - `_PyUnicodeWriter_WriteChar()`: replace `_PyUnicodeWriter_WriteChar(&writer, ch)` with `PyUnicodeWriter_WriteChar(writer, ch)`.
  - `_PyUnicodeWriter_WriteStr()`: replace `_PyUnicodeWriter_WriteStr(&writer, str)` with `PyUnicodeWriter_WriteStr(writer, str)`.
  - `_PyUnicodeWriter_WriteSubstring()`: replace `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` with `PyUnicodeWriter_WriteSubstring(writer, str, start, end)`.
  - `_PyUnicodeWriter_WriteASCIIString()`: replace `_PyUnicodeWriter_WriteASCIIString(&writer, str)` with `PyUnicodeWriter_WriteASCII(writer, str)`.
  - `_PyUnicodeWriter_WriteLatin1String()`: replace `_PyUnicodeWriter_WriteLatin1String(&writer, str)` with `PyUnicodeWriter_WriteUTF8(writer, str)`.
  - `_PyUnicodeWriter_Prepare()`: (no replacement).
  - `_PyUnicodeWriter_PrepareKind()`: (no replacement).
  - `_Py_HashPointer()`: use `Py_HashPointer()`.
  - `_Py_fopen_obj()`: use `Py_fopen()`.

The pythoncapi-compat project can be used to get these new public functions on Python 3.13 and older. (Contributed by Victor Stinner in gh-128863.)

Pending removal in future versions

The following APIs are deprecated and will be removed, although there is currently no date scheduled for their removal.

- `Py_TPFLAGS_HAVE_FINALIZE`: Unneeded since Python 3.8.
- `PyErr_Fetch()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_NormalizeException()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_Restore()`: Use `PyErr_SetRaisedException()` instead.
- `PyModule_GetFilename()`: Use `PyModule_GetFilenameObject()` instead.
- `PyOS_AfterFork()`: Use `PyOS_AfterFork_Child()` instead.
- `PySlice_GetIndicesEx()`: Use `PySlice_Unpack()` and `PySlice_AdjustIndices()` instead.
- `PyUnicode_READY()`: Unneeded since Python 3.12
- `PyErr_Display()`: Use `PyErr_DisplayException()` instead.
- `_PyErr_ChainExceptions()`: Use `_PyErr_ChainExceptions1()` instead.
- `PyBytesObject.ob_shash` member: call `PyObject_Hash()` instead.
- Thread Local Storage (TLS) API:
  - `PyThread_create_key()`: Use `PyThread_tss_alloc()` instead.
  - `PyThread_delete_key()`: Use `PyThread_tss_free()` instead.
  - `PyThread_set_key_value()`: Use `PyThread_tss_set()` instead.
  - `PyThread_get_key_value()`: Use `PyThread_tss_get()` instead.
  - `PyThread_delete_key_value()`: Use `PyThread_tss_delete()` instead.
  - `PyThread_ReInitTLS()`: Unneeded since Python 3.7.

11 Build changes

- **PEP 776**: Emscripten is now an officially supported platform at **tier 3**. As a part of this effort, more than 25 bugs in Emscripten libc were fixed. Emscripten now includes support for `ctypes`, `termios`, and `fcntl`, as well as experimental support for the new default interactive shell. (Contributed by R. Hood Chatham in gh-127146, gh-127683, and gh-136931.)
- Official Android binary releases are now provided on python.org.
- GNU Autoconf 2.72 is now required to generate `configure`. (Contributed by Erlend Aasland in gh-115765.)
- `wasm32-unknown-emscripten` is now a **PEP 11** tier 3 platform. (Contributed by R. Hood Chatham in gh-127146, gh-127683, and gh-136931.)
- `#pragma`-based linking with `python3*.lib` can now be switched off with `Py_NO_LINK_LIB`. (Contributed by Jean-Christophe Fillion-Robin in gh-82909.)
- CPython now enables a set of recommended compiler options by default for improved security. Use the `--disable-safety` `configure` option to disable them, or the `--enable-slower-safety` option for a larger set of compiler options, albeit with a performance cost.
- The `WITH_FREELISTS` macro and `--without-freelists` `configure` option have been removed.
- The new `configure` option `--with-tail-call-interp` may be used to enable the experimental tail call interpreter. See *A new type of interpreter* for further details.
- To disable the new remote debugging support, use the `--without-remote-debug` `configure` option. This may be useful for security reasons.

- iOS and macOS apps can now be configured to redirect `stdout` and `stderr` content to the system log. (Contributed by Russell Keith-Magee in gh-127592.)
- The iOS testbed is now able to stream test output while the test is running. The testbed can also be used to run the test suite of projects other than CPython itself. (Contributed by Russell Keith-Magee in gh-127592.)

11.1 `build-details.json`

Installations of Python now contain a new file, `build-details.json`. This is a static JSON document containing build details for CPython, to allow for introspection without needing to run code. This is helpful for use-cases such as Python launchers, cross-compilation, and so on.

`build-details.json` must be installed in the platform-independent standard library directory. This corresponds to the 'stdlib' `sysconfig` installation path, which can be found by running `sysconfig.get_path('stdlib')`.

See also

PEP 739 – `build-details.json` 1.0 – a static description file for Python build details

11.2 Discontinuation of PGP signatures

PGP (Pretty Good Privacy) signatures will not be provided for releases of Python 3.14 or future versions. To verify CPython artifacts, users must use Sigstore verification materials. Releases have been signed using Sigstore since Python 3.11.

This change in release process was specified in **PEP 761**.

11.3 Free-threaded Python is officially supported

The free-threaded build of Python is now supported and no longer experimental. This is the start of phase II where free-threaded Python is officially supported but still optional.

The free-threading team are confident that the project is on the right path, and appreciate the continued dedication from everyone working to make free-threading ready for broader adoption across the Python community.

With these recommendations and the acceptance of this PEP, the Python developer community should broadly advertise that free-threading is a supported Python build option now and into the future, and that it will not be removed without a proper deprecation schedule.

Any decision to transition to phase III, with free-threading as the default or sole build of Python is still undecided, and dependent on many factors both within CPython itself and the community. This decision is for the future.

See also

PEP 779

PEP 779's acceptance

11.4 Binary releases for the experimental just-in-time compiler

The official macOS and Windows release binaries now include an *experimental* just-in-time (JIT) compiler. Although it is **not** recommended for production use, it can be tested by setting `PYTHON_JIT=1` as an environment variable. Downstream source builds and redistributors can use the `--enable-experimental-jit=yes-off` configuration option for similar behavior.

The JIT is at an early stage and still in active development. As such, the typical performance impact of enabling it can range from 10% slower to 20% faster, depending on workload. To aid in testing and evaluation, a set of introspection functions has been provided in the `sys._jit` namespace. `sys._jit.is_available()` can be used to determine if the current executable supports JIT compilation, while `sys._jit.is_enabled()` can be used to tell if JIT compilation has been enabled for the current process.

Currently, the most significant missing functionality is that native debuggers and profilers like `gdb` and `perf` are unable to unwind through JIT frames (Python debuggers and profilers, like `pdb` or `profile`, continue to work without modification). Free-threaded builds do not support JIT compilation.

Please report any bugs or major performance regressions that you encounter!

See also

PEP 744

12 Porting to Python 3.14

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

12.1 Changes in the Python API

- On Unix platforms other than macOS, *forkserver* is now the default start method for `multiprocessing` and `ProcessPoolExecutor`, instead of *fork*.

  See *(1)* and *(2)* for details.

  If you encounter `NameError`s or pickling errors coming out of `multiprocessing` or `concurrent.futures`, see the forkserver restrictions.

  This change does not affect Windows or macOS, where 'spawn' remains the default start method.

- `functools.partial` is now a method descriptor. Wrap it in `staticmethod()` if you want to preserve the old behavior. (Contributed by Serhiy Storchaka and Dominykas Grigonis in gh-121027.)

- The *garbage collector is now incremental*, which means that the behavior of `gc.collect()` changes slightly:

  - `gc.collect(1)`: Performs an increment of garbage collection, rather than collecting generation 1.
  - Other calls to `gc.collect()` are unchanged.

- The `locale.nl_langinfo()` function now temporarily sets the `LC_CTYPE` locale in some cases. This temporary change affects other threads. (Contributed by Serhiy Storchaka in gh-69998.)

- `types.UnionType` is now an alias for `typing.Union`, causing changes in some behaviors. See *above* for more details. (Contributed by Jelle Zijlstra in gh-105499.)

- The runtime behavior of annotations has changed in various ways; see *above* for details. While most code that interacts with annotations should continue to work, some undocumented details may behave differently.

- As part of making the `mimetypes` CLI public, it now exits with `1` on failure instead of `0` and `2` on incorrect command-line parameters instead of `1`. Error messages are now printed to stderr.

- The `\B` pattern in regular expression now matches the empty string when given as the entire pattern, which may cause behavioural changes.

- On FreeBSD, `sys.platform` no longer contains the major version number.

12.2 Changes in annotations (PEP 649 and PEP 749)

This section contains guidance on changes that may be needed to annotations or Python code that interacts with or introspects annotations, due to the changes related to *deferred evaluation of annotations*.

In the majority of cases, working code from older versions of Python will not require any changes.

Implications for annotated code

If you define annotations in your code (for example, for use with a static type checker), then this change probably does not affect you: you can keep writing annotations the same way you did with previous versions of Python.

You will likely be able to remove quoted strings in annotations, which are frequently used for forward references. Similarly, if you use `from __future__ import annotations` to avoid having to write strings in annotations, you may well be able to remove that import once you support only Python 3.14 and newer. However, if you rely on third-party libraries that read annotations, those libraries may need changes to support unquoted annotations before they work as expected.

Implications for readers of `__annotations__`

If your code reads the `__annotations__` attribute on objects, you may want to make changes in order to support code that relies on deferred evaluation of annotations. For example, you may want to use `annotationlib.get_annotations()` with the `FORWARDREF` format, as the `dataclasses` module now does.

The external typing_extensions package provides partial backports of some of the functionality of the `annotationlib` module, such as the `Format` enum and the `get_annotations()` function. These can be used to write cross-version code that takes advantage of the new behavior in Python 3.14.

Related changes

The changes in Python 3.14 are designed to rework how `__annotations__` works at runtime while minimizing breakage to code that contains annotations in source code and to code that reads `__annotations__`. However, if you rely on undocumented details of the annotation behavior or on private functions in the standard library, there are many ways in which your code may not work in Python 3.14. To safeguard your code against future changes, only use the documented functionality of the `annotationlib` module.

In particular, do not read annotations directly from the namespace dictionary attribute of type objects. Use `annotationlib.get_annotate_from_class_namespace()` during class construction and `annotationlib.get_annotations()` afterwards.

In previous releases, it was sometimes possible to access class annotations from an instance of an annotated class. This behavior was undocumented and accidental, and will no longer work in Python 3.14.

`from __future__ import annotations`

In Python 3.7, **PEP 563** introduced the `from __future__ import annotations` future statement, which turns all annotations into strings.

However, this statement is now deprecated and it is expected to be removed in a future version of Python. This removal will not happen until after Python 3.13 reaches its end of life in 2029, being the last version of Python without support for deferred evaluation of annotations.

In Python 3.14, the behavior of code using `from __future__ import annotations` is unchanged.

12.3 Changes in the C API

- `Py_Finalize()` now deletes all interned strings. This is backwards incompatible to any C extension that holds onto an interned string after a call to `Py_Finalize()` and is then reused after a call to `Py_Initialize()`. Any issues arising from this behavior will normally result in crashes during the execution of the subsequent call to `Py_Initialize()` from accessing uninitialized memory. To fix, use an address sanitizer to identify any use-after-free coming from an interned string and deallocate it during module shutdown. (Contributed by Eddie Elizondo in gh-113601.)
- The Unicode Exception Objects C API now raises a `TypeError` if its exception argument is not a `UnicodeError` object. (Contributed by Bénédikt Tran in gh-127691.)
- The interpreter internally avoids some reference count modifications when loading objects onto the operands stack by borrowing references when possible. This can lead to smaller reference count values compared to previous Python versions. C API extensions that checked `Py_REFCNT()` of `1` to de-

termine if an function argument is not referenced by any other code should instead use `PyUnstable_Object_IsUniqueReferencedTemporary()` as a safer replacement.

- Private functions promoted to public C APIs:
  - `_PyBytes_Join()`: `PyBytes_Join()`
  - `_PyLong_IsNegative()`: `PyLong_IsNegative()`
  - `_PyLong_IsPositive()`: `PyLong_IsPositive()`
  - `_PyLong_IsZero()`: `PyLong_IsZero()`
  - `_PyLong_Sign()`: `PyLong_GetSign()`
  - `_PyUnicodeWriter_Dealloc()`: `PyUnicodeWriter_Discard()`
  - `_PyUnicodeWriter_Finish()`: `PyUnicodeWriter_Finish()`
  - `_PyUnicodeWriter_Init()`: use `PyUnicodeWriter_Create()`
  - `_PyUnicodeWriter_Prepare()`: (no replacement)
  - `_PyUnicodeWriter_PrepareKind()`: (no replacement)
  - `_PyUnicodeWriter_WriteChar()`: `PyUnicodeWriter_WriteChar()`
  - `_PyUnicodeWriter_WriteStr()`: `PyUnicodeWriter_WriteStr()`
  - `_PyUnicodeWriter_WriteSubstring()`: `PyUnicodeWriter_WriteSubstring()`
  - `_PyUnicode_EQ()`: `PyUnicode_Equal()`
  - `_PyUnicode_Equal()`: `PyUnicode_Equal()`
  - `_Py_GetConfig()`: `PyConfig_Get()` and `PyConfig_GetInt()`
  - `_Py_HashBytes()`: `Py_HashBuffer()`
  - `_Py_fopen_obj()`: `Py_fopen()`
  - `PyMutex_IsLocked()` : `PyMutex_IsLocked()`

  The pythoncapi-compat project can be used to get most of these new functions on Python 3.13 and older.

Index