What's New in Python

Release 3.14.0rc1

A. M. Kuchling

July 22, 2025

Python Software Foundation
Email: docs@python.org

Contents

Editor
Hugo van Kemenade

This article explains the new features in Python 3.14, compared to 3.13.

For full details, see the changelog.

See also

PEP 745 – Python 3.14 release schedule

Note

Prerelease users should be aware that this document is currently in draft form. It will be updated substantially as Python 3.14 moves towards release, so it's worth checking back even after reading earlier versions.

1 Summary – release highlights

Python 3.14 beta is the pre-release of the next version of the Python programming language, with a mix of changes to the language, the implementation and the standard library.

The biggest changes to the implementation include template strings (**PEP 750**), deferred evaluation of annotations (**PEP 649**), and a new type of interpreter that uses tail calls.

The library changes include the addition of a new `annotationlib` module for introspecting and wrapping annotations (**PEP 749**), a new `compression.zstd` module for Zstandard support (**PEP 784**), plus syntax highlighting in the REPL, as well as the usual deprecations and removals, and improvements in user-friendliness and correctness.

- *PEP 779: Free-threaded Python is officially supported*
- *PEP 649 and 749: deferred evaluation of annotations*
- *PEP 734: Multiple interpreters in the stdlib*
- *PEP 741: Python configuration C API*
- *PEP 750: Template strings*
- *PEP 758: Allow except and except* expressions without parentheses*
- *PEP 761: Discontinuation of PGP signatures*
- *PEP 765: Disallow return/break/continue that exit a finally block*
- *Free-threaded mode improvements*
- *PEP 768: Safe external debugger interface for CPython*
- *PEP 784: Adding Zstandard to the standard library*
- *A new type of interpreter*
- *Syntax highlighting in PyREPL*, and color output in *unittest*, *argparse*, *json* and *calendar* CLIs
- *Binary releases for the experimental just-in-time compiler*

2 Incompatible changes

On platforms other than macOS and Windows, the default start method for `multiprocessing` and `ProcessPoolExecutor` switches from *fork* to *forkserver*.

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.

The interpreter avoids some reference count modifications internally when it's safe to do so. This can lead to different values returned from `sys.getrefcount()` and `Py_REFCNT()` compared to previous versions of Python. See *below* for details.

3 New features

3.1 PEP 779: 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.

We are confident that the project is on the right path, and we 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, we as 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 and its acceptance.

3.2 PEP 734: Multiple interpreters in the stdlib

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 the 3.14 release, with the new `concurrent.interpreters` module.

There are at least two notable reasons why using multiple interpreters is worth considering:

- 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 software, 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 the 3.12 release, 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 stdlib 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 (we will be working next 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 extension modules on PyPI are not compatible with multiple interpreters yet (stdlib 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, we expect to slowly see libraries on PyPI 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*.

See also

PEP 734.

3.3 PEP 750: Template strings

Template string literals (t-strings) are a generalization of f-strings, using a `t` in place of the `f` prefix. Instead of evaluating to `str`, t-strings evaluate to a new `string.templatelib.Template` type:

```
from string.templatelib import Template

name = "World"
template: Template = t"Hello {name}"
```

The template can then be combined with functions that operate on the template's structure to produce a `str` or a string-like result. For example, sanitizing input:

```
evil = "<script>alert('evil')</script>"
template = t"<p>{evil}</p>"
assert html(template) == "<p>&lt;script&gt;alert('evil')&lt;/script&gt;</p>"
```

As another example, generating HTML attributes from data:

```
attributes = {"src": "shrubbery.jpg", "alt": "looks nice"}
template = t"<img {attributes}>"
assert html(template) == '<img src="shrubbery.jpg" alt="looks nice" />'
```

Compared to using an f-string, the `html` function has access to template attributes containing the original information: static strings, interpolations, and values from the original scope. Unlike existing templating approaches, t-strings build from the well-known f-string syntax and rules. Template systems thus benefit from Python tooling as they are much closer to the Python language, syntax, scoping, and more.

Writing template handlers is straightforward:

```
from string.templatelib import Template, Interpolation

def lower_upper(template: Template) -> str:
    """Render static parts lowercased and interpolations uppercased."""
    parts: list[str] = []
    for item in template:
        if isinstance(item, Interpolation):
            parts.append(str(item.value).upper())
        else:
```

(continues on next page)

(continued from previous page)

```
            parts.append(item.lower())
    return "".join(parts)

name = "world"
assert lower_upper(t"HELLO {name}") == "hello WORLD"
```

With this in place, developers can write template systems to sanitize 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.

3.4 PEP 768: Safe external debugger interface for CPython

PEP 768 introduces a zero-overhead debugging interface that allows debuggers and profilers to safely attach to running Python processes. This is a significant enhancement to Python's debugging capabilities allowing debuggers to forego unsafe alternatives. See *below* for how this feature is leveraged to implement the new `pdb` module's remote attaching capabilities.

The new interface provides safe execution points for attaching debugger code without modifying the interpreter's normal execution path or adding runtime overhead. This enables tools to inspect and interact with Python applications in real-time without stopping or restarting them — a crucial capability for high-availability systems and production environments.

For convenience, CPython implements this interface through the `sys` module with a `sys.remote_exec()` function:

```
sys.remote_exec(pid, 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.

Here's a simple example that inspects object types in a running Python process:

```
import os
import sys
import tempfile

# Create a temporary script
with tempfile.NamedTemporaryFile(mode='w', suffix='.py', delete=False) as␣
↪f:
    script_path = f.name
    f.write(f"import my_debugger; my_debugger.connect({os.getpid()})")
try:
    # Execute in process with PID 1234
    print("Behold! An offering:")
    sys.remote_exec(1234, script_path)
finally:
    os.unlink(script_path)
```

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.

A key implementation detail is that the interface piggybacks on the interpreter's existing evaluation loop and safe points, ensuring zero overhead during normal execution while providing a reliable way for external processes to coordinate debugging operations.

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

See also

PEP 768.

3.5 PEP 784: Adding Zstandard to 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 canonical names for importing these compression modules going forward. 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.

3.6 Remote attaching to a running Python process with 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 `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.)

See also

PEP 768.

3.7 PEP 758 – Allow except and except* expressions without parentheses

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

```
try:
    connect_to_server()
except TimeoutError, ConnectionRefusedError:
    print("Network issue encountered.")

 # The same applies to except* (for exception groups):

try:
    connect_to_server()
except* TimeoutError, ConnectionRefusedError:
    print("Network issue encountered.")
```

Check **PEP 758** for more details.

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

See also

PEP 758.

3.8 PEP 649 and 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 is specified in **PEP 649** and **PEP 749**.

This change is designed to make annotations in Python more performant and more usable 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)
```

(continues on next page)

(continued from previous page)

```
{'arg': ForwardRef('Undefined', owner=<function func at 0x...>)}
>>> get_annotations(func, format=Format.STRING)
{'arg': 'Undefined'}
```

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, use only 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` directive, which turns all annotations into strings. This directive is now considered deprecated and it is expected to be removed in a future version of Python. However, this removal will not happen until after Python 3.13, the last version of Python without deferred evaluation of annotations, reaches its end of life in 2029. In Python 3.14, the behavior of code using `from __future__ import annotations` is unchanged.

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

See also

PEP 649 and **PEP 749**.

3.9 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'?

>>> asynch def fetch_data():
...     pass
Traceback (most recent call last):
  File "<stdin>", line 1
    asynch def fetch_data():
    ^^^^^^
SyntaxError: invalid syntax. Did you mean 'async'?

>>> async def foo():
...     awaid fetch_data()
Traceback (most recent call last):
  File "<stdin>", line 2
    awaid fetch_data()
    ^^^^^
SyntaxError: invalid syntax. Did you mean 'await'?

>>> raisee ValueError("Error")
Traceback (most recent call last):
  File "<stdin>", line 1
    raisee ValueError("Error")
    ^^^^^^
SyntaxError: invalid syntax. Did you mean 'raise'?
```

  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.)

- When unpacking assignment fails due to incorrect number of variables, the error message prints the received number of values in more cases than before. (Contributed by Tushar Sadhwani in gh-122239.)

```
>>> x, y, z = 1, 2, 3, 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    x, y, z = 1, 2, 3, 4
    ^^^^^^^
ValueError: too many values to unpack (expected 3, got 4)
```

- `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?")
```

(continues on next page)

(continued from previous page)

```
File "<stdin>", line 5
  elif who is None:
  ^^^^
SyntaxError: 'elif' block follows an 'else' block
```

- If a statement (`pass`, `del`, `return`, `yield`, `raise`, `break`, `continue`, `assert`, `import`, `from`) 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.)

```
>>> import ast as arr[0]
  File "<python-input-1>", line 1
    import ast as arr[0]
                  ^^^^^^
SyntaxError: cannot use subscript as import target
```

- 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')
```

3.10 PEP 741: 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; 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 the Python initialization. This PEP unifies also the configuration of the Python preinitialization and the Python 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 simplify the API further.

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.

3.11 Asyncio introspection capabilities

Added a new command-line interface to inspect running Python processes using asynchronous tasks, available via:

```
python -m asyncio ps PID
```

This tool 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.

```
python -m asyncio pstree PID
```

This tool fetches the same information, but 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):
    await asyncio.sleep(5)
    print(f"🎵 Finished: {track}")

async def album(name, tracks):
    async with asyncio.TaskGroup() as tg:
        for track in tracks:
            tg.create_task(play(track), name=track)

async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(
          album("Sundowning", ["TNDNBTG", "Levitate"]), name="Sundowning")
        tg.create_task(
          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
```

(continues on next page)

(continued from previous page)

```
└── (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
                │       └── 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.)

3.12 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 numbers on our machines suggest anywhere up to 30% faster Python code, and a geometric mean of 3-5% faster on `pyperformance` 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, we expect that a future release of GCC will support this as well.

This feature is opt-in for now. We highly recommend enabling profile-guided optimization with the new interpreter as it is the only configuration we have tested and can validate its improved performance. For further information on how to build Python, 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.

Attention

This section previously reported a 9-15% geometric mean speedup. This number has since been cautiously revised down to 3-5%. While we expect performance results to be better than what we report, our estimates are more conservative due to a compiler bug found in Clang/LLVM 19, which causes the normal interpreter to be slower. We were unaware of this bug, resulting in inaccurate results. We sincerely apologize for communicating results that were only accurate for LLVM v19.1.x and v20.1.0. In the meantime, the bug has been fixed in LLVM v20.1.1 and for the upcoming v21.1, but it will remain unfixed for LLVM v19.1.x and v20.1.0. Thus any benchmarks with those versions of LLVM may produce inaccurate numbers. (Thanks to Nelson Elhage for bringing this to light.)

(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.)

3.13 Free-threaded mode

Free-threaded mode (**PEP 703**), initially added in 3.13, has been significantly improved. The implementation described in PEP 703 was 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 platform and C compiler used.

This work was done by many contributors: 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.

From 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()`.

A new flag has been added, `context_aware_warnings`. This flag defaults to true for the free-threaded build and false for the GIL-enabled build. If the flag is true then the `warnings.catch_warnings` context manager uses a context variable for warning filters. This makes the context manager behave predicably when used with multiple threads or asynchronous tasks.

A new flag has been added, `thread_inherit_context`. This flag defaults to true for the free-threaded build and false for the GIL-enabled build. If the flag is true then 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.

3.14 Syntax highlighting in PyREPL

The default interactive shell now highlights Python syntax as you type. The feature is enabled by default unless the `PYTHON_BASIC_REPL` environment is set or any color-disabling environment variables are used. See using-on-controlling-color for details.

The default color theme for syntax highlighting strives for good contrast and uses exclusively 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, as well as in the `PYTHONSTARTUP` script.

(Contributed by Łukasz Langa in gh-131507.)

3.15 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

3.16 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 predicable 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.)

3.17 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.)

4 Other language changes

- The default interactive shell now supports import autocompletion. This means that typing `import foo` and pressing `<tab>` will suggest modules starting with `foo`. Similarly, typing `from foo import b` will suggest submodules of `foo` starting with `b`. Note that autocompletion of module attributes is not currently supported. (Contributed by Tomas Roun in gh-69605.)
- The `map()` built-in 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.)
- Incorrect usage of `await` and asynchronous comprehensions is now detected even if the code is optimized away by the `-O` command-line option. For example, `python -O -c 'assert await 1'` now produces a `SyntaxError`. (Contributed by Jelle Zijlstra in gh-121637.)

- Writes to `__debug__` are now detected even if the code is optimized away by the `-O` command-line option. For example, `python -O -c 'assert (__debug__ := 1)'` now produces a `SyntaxError`. (Contributed by Irit Katriel in gh-122245.)
- Add class methods `float.from_number()` and `complex.from_number()` to convert a number to `float` or `complex` type correspondingly. They raise an error if the argument is a string. (Contributed by Serhiy Storchaka in gh-84978.)
- Implement mixed-mode arithmetic rules combining real and complex numbers as specified by C standards since C99. (Contributed by Sergey B Kirpichev in gh-69639.)
- All Windows code pages are now supported as "cpXXX" codecs on Windows. (Contributed by Serhiy Storchaka in gh-123803.)
- `super` objects are now `pickleable` and `copyable`. (Contributed by Serhiy Storchaka in gh-125767.)
- The `memoryview` type now supports subscription, making it a generic type. (Contributed by Brian Schubert in gh-126012.)
- 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 `bytes.fromhex()` and `bytearray.fromhex()` methods now accept ASCII `bytes` and bytes-like objects. (Contributed by Daniel Pope in gh-129349.)
- 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 empty input string. Now it is always the opposite of `\b`. (Contributed by Serhiy Storchaka in gh-124130.)
- 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.)
- 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.)
- 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.)
- 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.)
- When subclassing from 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-132329.)
- 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.)
- Improve error message when an object supporting the synchronous context manager protocol is entered using `async with` instead of `with`. And vice versa with the asynchronous context manager protocol. (Contributed by Bénédikt Tran in gh-128398.)
- `-J` is no longer a reserved flag for Jython, and now has no special meaning. (Contributed by Adam Turner in gh-133336.)

4.1 PEP 765: Disallow `return`/`break`/`continue` that exit a `finally` block

The compiler emits a `SyntaxWarning` when a `return`, `break` or `continue` statements appears where it exits a `finally` block. This change is specified in **PEP 765**.

5 New modules

- `annotationlib`: For introspecting annotations. See **PEP 749** for more details. (Contributed by Jelle Zijlstra in gh-119180.)

6 Improved modules

6.1 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.)

6.2 ast

- Add `ast.compare()` 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.)
- `ast.parse()`, when called with an AST as input, now always verifies that the root node type is appropriate. (Contributed by Irit Katriel in gh-130139.)
- Add new `--feature-version`, `--optimize`, `--show-empty` options to command-line interface. (Contributed by Semyon Moroz in gh-133367.)

6.3 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.)

6.4 bdb

- The `bdb` module now supports the `sys.monitoring` backend. (Contributed by Tian Gao in gh-124533.)

6.5 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.)

6.6 concurrent.futures

- Add `InterpreterPoolExecutor`, which exposes "subinterpreters" (multiple Python interpreters in the same process) to Python code. This is separate from the proposed API in **PEP 734**. (Contributed by Eric Snow in gh-124548.)
- The default `ProcessPoolExecutor` start method changed from fork to forkserver on platforms other than macOS and Windows where it was already spawn.

  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 `concurrent.futures.ProcessPoolExecutor.terminate_workers()` and `concurrent.futures.ProcessPoolExecutor.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 `concurrent.futures.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.)

6.7 configparser

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

6.8 contextvars

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

6.9 ctypes

- The layout of bit fields in `Structure` and `Union` now matches platform defaults (GCC/Clang or MSVC) more closely. 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.)
- `ctypes.memoryview_at()` now exists 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 `ctypes._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 `ctypes.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.)

6.10 curses

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

6.11 datetime

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

6.12 decimal

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

6.13 difflib

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

6.14 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.)

6.15 errno

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

6.16 faulthandler

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

6.17 fnmatch

- Added `fnmatch.filterfalse()` for excluding names matching a pattern. (Contributed by Bénédikt Tran in gh-74598.)

6.18 fractions

- Add support for converting any objects that have the `as_integer_ratio()` method to a `Fraction`. (Contributed by Serhiy Storchaka in gh-82017.)
- Add alternative `Fraction` constructor `Fraction.from_number()`. (Contributed by Serhiy Storchaka in gh-121797.)

6.19 functools

- Add support to `functools.partial()` and `functools.partialmethod()` for `functools.Placeholder` sentinels to reserve a place for positional arguments. (Contributed by Dominykas Grigonis in gh-119127.)
- Allow the *initial* parameter of `functools.reduce()` to be passed as a keyword argument. (Contributed by Sayandip Dutta in gh-125916.)

6.20 gc

The cyclic garbage collector is now incremental, which changes the meaning of the results of `get_threshold()` and `set_threshold()` as well as `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 meaningless and is always zero.
- `set_threshold()` 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.

6.21 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.)

6.22 getpass

- Support keyboard feedback by `getpass.getpass()` 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.)

6.23 graphlib

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

6.24 heapq

- Add functions for working with max-heaps:
  - `heapq.heapify_max()`,
  - `heapq.heappush_max()`,
  - `heapq.heappop_max()`,
  - `heapq.heapreplace_max()`
  - `heapq.heappushpop_max()`

6.25 hmac

- Add a built-in implementation for HMAC (**RFC 2104**) using formally verified code from the HACL* project. (Contributed by Bénédikt Tran in gh-99108.)

6.26 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.)

6.27 imaplib

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

6.28 inspect

- `inspect.signature()` takes a new argument *annotation_format* to control the `annotationlib.Format` used for representing annotations. (Contributed by Jelle Zijlstra in gh-101552.)
- `inspect.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 `inspect.ispackage()` to determine whether an object is a package or not. (Contributed by Zhikang Yan in gh-125634.)

6.29 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 protocols `io.Reader` and `io.Writer` as a simpler alternatives to the pseudo-protocols `typing.IO`, `typing.TextIO`, and `typing.BinaryIO`. (Contributed by Sebastian Rittau in gh-127648.)

6.30 json

- Add notes for JSON serialization errors that allow to identify the source of the error. (Contributed by Serhiy Storchaka in gh-122163.)

- Enable the `json` module to work as a script using the `-m` switch: **`python -m json`**. 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.)

6.31 linecache

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

6.32 logging.handlers

- `logging.handlers.QueueListener` now implements the context manager protocol, allowing it to be used in a `with` statement. (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.)

6.33 math

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

6.34 mimetypes

- Document the command-line for `mimetypes`. It now exits with `1` on failure instead of `0` and `2` on incorrect command-line parameters instead of `1`. Also, errors are printed to stderr instead of stdout and their text is made tighter. (Contributed by Oleg Iarygin and Hugo van Kemenade in gh-93096.)
- Add MS 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`

  (Contributed by Sahil Prajapati and Hugo van Kemenade in gh-84852.)
- Add **RFC 9559** MIME types for Matroska audiovisual data container structures, containing:
  - audio with no video: `audio/matroska` (`.mka`)
  - video: `video/matroska` (`.mkv`)
  - stereoscopic video: `video/matroska-3d` (`.mk3d`)

  (Contributed by Hugo van Kemenade in gh-89416.)
- Add MIME types for 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`)

  (Contributed by Hugo van Kemenade in gh-85957.)

- More MIME type 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`)
  - 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 Hugo van Kemenade in gh-129965.)
- Add **RFC 9512** `application/yaml` MIME type for YAML files (`.yaml` and `.yml`). (Contributed by Sasha "Nelie" Chernykh and Hugo van Kemenade in gh-132056.)

6.35 multiprocessing

- The default start method changed from fork to forkserver on platforms other than macOS and Windows where it was already spawn.

  If the threading incompatible *fork* method is required, you must explicitly request it via a context from `multiprocessing.get_context()` (preferred) or change the default via `multiprocessing.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 `multiprocessing.Manager()` method is now available. (Contributed by Mingyu Park in gh-129949.)
- Add `multiprocessing.Process.interrupt()` 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.)

6.36 operator

- Two new functions `operator.is_none()` and `operator.is_not_none()` have been added, 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.)

6.37 os

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

6.38 os.path

- The *strict* parameter to `os.path.realpath()` accepts a new value, `os.path.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**.)

6.39 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 `pathlib.Path.info` attribute, which stores an object implementing the `pathlib.types.PathInfo` protocol (also new). 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.)

6.40 pdb

- Hardcoded breakpoints (`breakpoint()` and `pdb.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` now supports two backends: `sys.settrace()` and `sys.monitoring`. Using `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.)
- `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 PyREPL, in addition to the newly added `colorize` argument of `pdb.Pdb`. (Contributed by Tian Gao and Łukasz Langa in gh-133355.)

6.41 pickle

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

6.42 platform

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

6.43 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.)

6.44 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.)

6.45 ssl

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

6.46 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.)

6.47 symtable

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

  (Contributed by Bénédikt Tran in gh-120029.)

6.48 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.
- Add `sys._is_immortal()` for determining if an object is immortal. (Contributed by Peter Bierma in gh-128509.)
- On FreeBSD, `sys.platform` doesn't contain the major version anymore. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`.
- Raise `DeprecationWarning` for `sys._clear_type_cache()`. This function was deprecated in Python 3.13 but it didn't raise a runtime warning.

6.49 sys.monitoring

- Two new events are added: `BRANCH_LEFT` and `BRANCH_RIGHT`. The `BRANCH` event is deprecated.

6.50 sysconfig

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

6.51 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**.)

6.52 threading

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

6.53 tkinter

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

6.54 turtle

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

6.55 types

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

6.56 typing

- `types.UnionType` and `typing.Union` 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. Users should 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 `typing.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 `typing.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.)

6.57 unicodedata

- The Unicode database has been updated to Unicode 16.0.0.

6.58 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 string(s).

  (Contributed by Serhiy Storchaka in gh-71339.)

6.59 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 `urllib.request.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 `urllib.request.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.)

6.60 uuid

- Add support for UUID versions 6, 7, and 8 via `uuid.uuid6()`, `uuid.uuid7()`, and `uuid.uuid8()` respectively, as specified in **RFC 9562**. (Contributed by Bénédikt Tran in gh-89083.)
- `uuid.NIL` and `uuid.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 to generate multiple UUIDs at once via `python -m uuid --count`. (Contributed by Simon Legner in gh-131236.)

6.61 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.

6.62 zipinfo

- Added `ZipInfo._for_archive` to resolve suitable defaults for a `ZipInfo` object as used by `ZipFile.writestr`. (Contributed by Bénédikt Tran in gh-123424.)
- `zipfile.ZipFile.writestr()` now respect `SOURCE_DATE_EPOCH` that distributions can set centrally and have build tools consume this in order to produce reproducible output. (Contributed by Jiahao Li in gh-91279.)

7 Optimizations

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

  (Contributed by Adam Turner, Bénédikt Tran, Chris Markiewicz, Eli Schwartz, Hugo van Kemenade, Jelle Zijlstra, and others in gh-118761.)

7.1 asyncio

- `asyncio` has a new per-thread double linked list implementation internally for `native tasks` which speeds up execution by 10-20% on standard pyperformance benchmarks and reduces 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.)
- `asyncio` has first class support for free-threading builds. This enables parallel execution of multiple event loops across different threads and scales linearly with the number of threads. (Contributed by Kumar Aditya in gh-128002.)
- `asyncio` has new utility functions for introspecting and printing the program's call graph: `asyncio.capture_call_graph()` and `asyncio.print_call_graph()`. (Contributed by Yury Selivanov, Pablo Galindo Salgado, and Łukasz Langa in gh-91048.)

7.2 base64

- Improve the performance of `base64.b16decode()` by up to ten times, and reduce the import time of `base64` by up to six times. (Contributed by Bénédikt Tran, Chris Markiewicz, and Adam Turner in gh-118761.)

7.3 gc

- The new *incremental garbage collector* means that maximum pause times are reduced by an order of magnitude or more for larger heaps. (Contributed by Mark Shannon in gh-108362.)

7.4 io

- `io` which provides the built-in `open()` makes less system calls when opening regular files as well as reading whole files. Reading a small operating system cached file in full is up to 15% faster. `pathlib.Path.read_bytes()` has the most optimizations for reading a file's bytes in full. (Contributed by Cody Maloney and Victor Stinner in gh-120754 and gh-90102.)

7.5 uuid

- Improve generation of `UUID` objects via their dedicated functions:
  - `uuid3()` and `uuid5()` are both roughly 40% faster for 16-byte names and 20% faster for 1024-byte names. Performance for longer names remains unchanged.
  - `uuid4()` is 30% faster.

  (Contributed by Bénédikt Tran in gh-128150.)

7.6 zlib

- On Windows, `zlib-ng` is now used as the implementation of the `zlib` module. This should produce compatible and comparable results with better performance, though it is worth noting that `zlib.Z_BEST_SPEED` (1) may result in significantly less compression than the previous implementation (while also significantly reducing the time taken to compress). (Contributed by Steve Dower in gh-91349.)

8 Deprecated

- `argparse`:
  - Passing the undocumented keyword argument *prefix_chars* to `add_argument_group()` is now deprecated. (Contributed by Savannah Ostrowski in gh-125563.)
  - Deprecated the `argparse.FileType` type converter. Anything with resource management should be done downstream after the arguments are parsed. (Contributed by Serhiy Storchaka in gh-58032.)
- `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`: 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.)

- codecs: `codecs.open()` is now deprecated. 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 deprecated in favor of setting `Structure._layout_` to `'ms'`. (Contributed by Petr Viktorin in gh-131747.)
  - Calling `ctypes.POINTER()` on a string is deprecated. Use ctypes-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. (Contributed by Kirill Podoprigora in gh-121676.)
- 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.)
- nturl2path: This module is now deprecated. Call `urllib.request.url2pathname()` and `pathname2url()` instead. (Contributed by Barney Gale in gh-125866.)
- os: Soft deprecate `os.popen()` and `os.spawn*` functions. 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 deprecated and will be removed 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. 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. (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, byte-like objects and `None` in `urllib.parse` functions `parse_qsl()` and `parse_qs()` is now deprecated. (Contributed by Serhiy Storchaka in gh-116897.)

8.1 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.)

8.2 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.

8.3 Pending removal in Python 3.17

- 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.

8.4 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.

8.5 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:
  - bool(NotImplemented).
  - 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 Removed

9.1 argparse

- Remove the *type*, *choices*, and *metavar* parameters of `argparse.BooleanOptionalAction`. They were deprecated since 3.12.

- Calling `add_argument_group()` on an argument group, and calling `add_argument_group()` or `add_mutually_exclusive_group()` on a mutually exclusive group now raise exceptions. 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.)

9.2 ast

- Remove the following classes. They were all deprecated since Python 3.8, and have emitted deprecation warnings since Python 3.12:

  - `ast.Bytes`
  - `ast.Ellipsis`
  - `ast.NameConstant`
  - `ast.Num`
  - `ast.Str`

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

  Also, remove the following deprecated properties on `ast.Constant`, which were present for compatibility with the now-removed AST classes:

  - `ast.Constant.n`
  - `ast.Constant.s`

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

9.3 asyncio

- Remove the following classes and functions. They were all deprecated and emitted deprecation warnings since Python 3.12:

  - `asyncio.get_child_watcher()`
  - `asyncio.set_child_watcher()`
  - `asyncio.AbstractEventLoopPolicy.get_child_watcher()`
  - `asyncio.AbstractEventLoopPolicy.set_child_watcher()`
  - `asyncio.AbstractChildWatcher`
  - `asyncio.FastChildWatcher`
  - `asyncio.MultiLoopChildWatcher`
  - `asyncio.PidfdChildWatcher`
  - `asyncio.SafeChildWatcher`
  - `asyncio.ThreadedChildWatcher`

  (Contributed by Kumar Aditya in gh-120804.)

- Removed implicit creation of event loop by `asyncio.get_event_loop()`. It now raises a `RuntimeError` if there is no current 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())
```

(continues on next page)

(continued from previous page)

```
    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())
```

9.4 collections.abc

- Remove `collections.abc.ByteString`. It had previously raised a `DeprecationWarning` since Python 3.12.

9.5 email

- Remove the *isdst* parameter from `email.utils.localtime()`. (Contributed by Hugo van Kemenade in gh-118798.)

9.6 importlib

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

  Use `importlib.resources.abc` classes instead:
  - `importlib.resources.abc.Traversable`
  - `importlib.resources.abc.TraversableResources`

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

9.7 itertools

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

9.8 pathlib

- Remove support for passing additional keyword arguments to `pathlib.Path`. In previous versions, any such arguments are ignored.
- Remove support for passing additional positional arguments to `pathlib.PurePath.relative_to()` and `is_relative_to()`. In previous versions, any such arguments are joined onto *other*.

9.9 pkgutil

- Remove deprecated `pkgutil.get_loader()` and `pkgutil.find_loader()`. These had previously raised a `DeprecationWarning` since Python 3.12. (Contributed by Bénédikt Tran in gh-97850.)

9.10 pty

- Remove deprecated `pty.master_open()` and `pty.slave_open()`. They had previously raised a `DeprecationWarning` since Python 3.12. Use `pty.openpty()` instead. (Contributed by Nikita Sobolev in gh-118824.)

9.11 sqlite3

- Remove `version` and `version_info` from `sqlite3`; 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.)
- Disallow using a sequence of parameters with named placeholders. This had previously raised a `DeprecationWarning` since Python 3.12; it will now raise a `sqlite3.ProgrammingError`. (Contributed by Erlend E. Aasland in gh-118928 and gh-101693.)

9.12 typing

- Remove `typing.ByteString`. It had previously raised a `DeprecationWarning` since Python 3.12.
- `typing.TypeAliasType` now supports star unpacking.

9.13 urllib

- Remove deprecated `Quoter` class from `urllib.parse`. It had previously raised a `DeprecationWarning` since Python 3.11. (Contributed by Nikita Sobolev in gh-118827.)
- Remove deprecated `URLopener` and `FancyURLopener` classes from `urllib.request`. They had previously raised a `DeprecationWarning` since Python 3.3.

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

9.14 Others

- Using `NotImplemented` in a boolean context will now raise a `TypeError`. It had previously raised a `DeprecationWarning` since Python 3.9. (Contributed by Jelle Zijlstra in gh-118767.)
- The `int()` built-in no longer delegates to `__trunc__()`. Classes that want to support conversion to integer must implement either `__int__()` or `__index__()`. (Contributed by Mark Dickinson in gh-119743.)

10 CPython bytecode changes

- Replaced the opcode `BINARY_SUBSCR` by `BINARY_OP` with oparg `NB_SUBSCR`. (Contributed by Irit Katriel in gh-100239.)

11 Porting to Python 3.14

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

11.1 Changes in the Python API

- `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 sets temporarily 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.

12 Build changes

- GNU Autoconf 2.72 is now required to generate `configure`. (Contributed by Erlend Aasland in gh-115765.)
- `#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.)

12.1 PEP 761: Discontinuation of PGP signatures

PGP signatures will not be available for CPython 3.14 and onwards. Users verifying artifacts must use Sigstore verification materials for verifying CPython artifacts. This change in release process is specified in **PEP 761**.

13 C API changes

13.1 New features

- Add `PyLong_GetSign()` function to get the sign of `int` objects. (Contributed by Sergey B Kirpichev in gh-116560.)
- Add a new `PyUnicodeWriter` API to create a Python `str` object:
  - `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.)

- 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_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 from/to Python `int`:
  - `PyLong_AsInt32()`
  - `PyLong_AsInt64()`
  - `PyLong_AsUInt32()`
  - `PyLong_AsUInt64()`
  - `PyLong_FromInt32()`
  - `PyLong_FromInt64()`
  - `PyLong_FromUInt32()`
  - `PyLong_FromUInt64()`

  (Contributed by Victor Stinner in gh-120389.)
- Add `PyBytes_Join(sep, iterable)` function, similar to `sep.join(iterable)` in Python. (Contributed by Victor Stinner in gh-121645.)
- Add `Py_HashBuffer()` to compute and return the hash value of a buffer. (Contributed by Antoine Pitrou and Victor Stinner in gh-122854.)
- Add functions to get and set the current runtime Python configuration (**PEP 741**):
  - `PyConfig_Get()`
  - `PyConfig_GetInt()`
  - `PyConfig_Set()`
  - `PyConfig_Names()`

  (Contributed by Victor Stinner in gh-107954.)
- Add functions to configure the Python initialization (**PEP 741**):
  - `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 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 `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 `PyUnicode_Equal()` function to the limited C API: test if two strings are equal. (Contributed by Victor Stinner in gh-124502.)
- Add `PyType_Freeze()` function to make a type immutable. (Contributed by Victor Stinner in gh-121654.)
- Add `PyUnstable_Object_EnableDeferredRefcount()` for enabling deferred reference counting, as outlined in **PEP 703**.
- Add `PyMonitoring_FireBranchLeftEvent()` and `PyMonitoring_FireBranchRightEvent()` for generating `BRANCH_LEFT` and `BRANCH_RIGHT` events, respectively.
- Add `Py_fopen()` function to open a file. Similar to the `fopen()` function, but the *path* parameter is a Python object and an exception is set on error. Add also `Py_fclose()` function to close a file. (Contributed by Victor Stinner in gh-127350.)
- 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 macros `Py_PACK_VERSION()` and `Py_PACK_FULL_VERSION()` for bit-packing Python version numbers. (Contributed by Petr Viktorin in gh-128629.)
- Add `PyUnstable_IsImmortal()` for determining whether an object is immortal, for debugging purposes.
- 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 support for a new `p` format unit in `Py_BuildValue()` that allows to take a C integer and produce a Python `bool` object. (Contributed by Pablo Galindo in bpo-45325.)
- 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.
- Add `PyUnstable_Object_IsUniquelyReferenced()` as a replacement for `Py_REFCNT(op) == 1` on free threaded builds. (Contributed by Peter Bierma in gh-133140.)

13.2 Limited C API changes

- In the limited C API 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 these macros never worked in the limited C API. Keep `PySequence_Fast()` in the limited C API. (Contributed by Victor Stinner in gh-91417.)

13.3 Porting to Python 3.14

- `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 determine 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()`
  - `_PyMutex_IsLocked()` : `PyMutex_IsLocked()`
  - `_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()`

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

13.4 Deprecated

- The `Py_HUGE_VAL` macro is soft deprecated, use `Py_INFINITY` instead. (Contributed by Sergey B Kirpichev in gh-120026.)
- Macros `Py_IS_NAN`, `Py_IS_INFINITY` and `Py_IS_FINITE` are soft deprecated, use instead `isnan`, `isinf` and `isfinite` available from `math.h` since C99. (Contributed by Sergey B Kirpichev in gh-119613.)
- Non-tuple sequences are 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 previously undocumented function `PySequence_In()` is soft deprecated. Use `PySequence_Contains()` instead. (Contributed by Yuki Kobayashi in gh-127896.)
- The `PyMonitoring_FireBranchEvent` function is deprecated and should be replaced with calls to `PyMonitoring_FireBranchLeftEvent()` and `PyMonitoring_FireBranchRightEvent()`.
- 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)`.
  - `_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 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

- Deprecated private functions (gh-128863):
  - `_PyBytes_Join()`: use `PyBytes_Join()`.
  - `_PyDict_GetItemStringWithError()`: use `PyDict_GetItemStringRef()`.
  - `_PyDict_Pop()`: `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_WriteUTF8(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.

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.

13.5 Removed

- Creating `immutable types` with mutable bases was deprecated since 3.12 and now raises a `TypeError`.
- Remove `PyDictObject.ma_version_tag` member which was deprecated since 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.)
- The undocumented APIs `Py_C_RECURSION_LIMIT` and `PyThreadState.c_recursion_remaining`, added in 3.13, are removed without a deprecation period. Please use `Py_EnterRecursiveCall()` to guard against runaway recursion in C code. (Removed in gh-133079, see also gh-130396.)

Index