
What's New in Python

发行版本 3.14.0rc3

A. M. Kuchling

十月 01, 2025

Python Software Foundation
Email: docs@python.org

Contents

| | | |
|------|--------------------------------------------------------------------------------------|----|
| 1 | 摘要 -- 发布关键点 | 4 |
| 2 | 不兼容的更改 | 4 |
| 3 | 新的特性 | 5 |
| 3.1 | PEP 779: 自由线程 Python 已获官方支持 | 5 |
| 3.2 | PEP 734: 添加多解释器到标准库 | 5 |
| 3.3 | PEP 750: 模板字符串 | 6 |
| 3.4 | PEP 768: Cpython 的安全外部调试器接口 | 7 |
| 3.5 | PEP 784: 添加 Zstandard 到标准库 | 8 |
| 3.6 | 使用 PDB 远程附加到正在运行的 Python 进程 | 8 |
| 3.7 | PEP 758 – 允许不带括号的 <code>except</code> 和 <code>except*</code> 表达式 | 8 |
| 3.8 | PEP 649 和 749: 注解的延迟求值 | 9 |
| 3.9 | 改进的错误消息 | 10 |
| 3.10 | PEP 741: Python 配置 C API | 12 |
| 3.11 | asyncio 内省能力 | 13 |
| 3.12 | 一种新型的解释器 | 14 |
| 3.13 | 自由线程模式 | 15 |
| 3.14 | PyREPL 的语法高亮功能 | 15 |
| 3.15 | 实验性即时编译器 (JIT) 的二进制发布版本 | 16 |
| 3.16 | 并发安全的警告控制 | 16 |
| 3.17 | 增量式垃圾回收 | 16 |
| 4 | 平台支持 | 16 |
| 5 | 其他语言特性修改 | 16 |
| 5.1 | PEP 765: 禁止在 <code>finally</code> 代码块中使用会跳出该块的 <code>return/break/continue</code> 语句 | 18 |
| 6 | 新增模块 | 18 |
| 7 | 改进的模块 | 18 |
| 7.1 | <code>argparse</code> | 18 |
| 7.2 | <code>ast</code> (抽象语法树) | 18 |
| 7.3 | <code>asyncio</code> | 18 |
| 7.4 | <code>calendar</code> (日历) | 18 |
| 7.5 | <code>concurrent.futures</code> | 19 |
| 7.6 | <code>configparser</code> | 19 |
| 7.7 | <code>contextvars</code> | 19 |

| | | |
|----------|------------------|-----------|
| 7.8 | ctypes | 19 |
| 7.9 | curses | 20 |
| 7.10 | datetime | 20 |
| 7.11 | decimal | 20 |
| 7.12 | difflib | 20 |
| 7.13 | dis | 20 |
| 7.14 | errno | 20 |
| 7.15 | faulthandler | 20 |
| 7.16 | fnmatch | 20 |
| 7.17 | fractions | 21 |
| 7.18 | functools | 21 |
| 7.19 | getopt | 21 |
| 7.20 | getpass | 21 |
| 7.21 | graphlib | 21 |
| 7.22 | heapq | 21 |
| 7.23 | hmac | 21 |
| 7.24 | http | 21 |
| 7.25 | imaplib | 22 |
| 7.26 | inspect | 22 |
| 7.27 | io | 22 |
| 7.28 | json | 22 |
| 7.29 | linecache | 22 |
| 7.30 | logging.handlers | 22 |
| 7.31 | math | 22 |
| 7.32 | mimetypes | 22 |
| 7.33 | multiprocessing | 24 |
| 7.34 | operator | 24 |
| 7.35 | os | 24 |
| 7.36 | os.path | 24 |
| 7.37 | pathlib | 25 |
| 7.38 | pdb | 25 |
| 7.39 | pickle | 25 |
| 7.40 | platform | 25 |
| 7.41 | pydoc | 25 |
| 7.42 | socket | 26 |
| 7.43 | ssl | 26 |
| 7.44 | struct | 26 |
| 7.45 | symtable | 26 |
| 7.46 | sys | 26 |
| 7.47 | sys.monitoring | 26 |
| 7.48 | sysconfig | 27 |
| 7.49 | tarfile | 27 |
| 7.50 | threading | 27 |
| 7.51 | tkinter | 27 |
| 7.52 | turtle (海龟绘图) | 27 |
| 7.53 | types (类型) | 27 |
| 7.54 | typing | 27 |
| 7.55 | unicodedata | 28 |
| 7.56 | unittest | 28 |
| 7.57 | urllib | 28 |
| 7.58 | uuid | 29 |
| 7.59 | webbrowser | 29 |
| 7.60 | zipfile | 29 |
| 8 | 性能优化 | 29 |
| 8.1 | asyncio | 29 |
| 8.2 | base64 | 29 |
| 8.3 | bdb | 29 |

| | | |
|-----------|------------------------|-----------|
| 8.4 | difflib | 30 |
| 8.5 | gc | 30 |
| 8.6 | io | 30 |
| 8.7 | pathlib | 30 |
| 8.8 | pdb | 30 |
| 8.9 | uuid | 30 |
| 8.10 | zlib | 30 |
| 9 | 移除 | 31 |
| 9.1 | argparse | 31 |
| 9.2 | ast (抽象语法树) | 31 |
| 9.3 | asyncio | 31 |
| 9.4 | email | 33 |
| 9.5 | importlib.abc | 33 |
| 9.6 | itertools | 33 |
| 9.7 | pathlib | 33 |
| 9.8 | pkgutil | 33 |
| 9.9 | pty | 33 |
| 9.10 | sqlite3 | 33 |
| 9.11 | urllib | 34 |
| 10 | 弃用 | 34 |
| 10.1 | 新的弃用 | 34 |
| 10.2 | 计划在 Python 3.15 中移除 | 35 |
| 10.3 | 计划在 Python 3.16 中移除 | 36 |
| 10.4 | 计划在 Python 3.17 中移除 | 38 |
| 10.5 | 计划在 Python 3.19 中移除 | 38 |
| 10.6 | 计划在未来版本中移除 | 38 |
| 11 | CPython 字节码的改变 | 40 |
| 11.1 | Pseudo-instructions | 41 |
| 12 | C API 的变化 | 41 |
| 12.1 | C API 中的新特性 | 41 |
| 12.2 | 受限 C API 的变化 | 44 |
| 12.3 | 被移除的 C API | 44 |
| 12.4 | 已弃用的 C API | 44 |
| 13 | 构建的变化 | 47 |
| 13.1 | build-details.json | 48 |
| 13.2 | PGP 签名的停用 | 48 |
| 14 | 移植到 Python 3.14 | 48 |
| 14.1 | Python API 的变化 | 48 |
| 14.2 | C API 的变化 | 48 |
| | 索引 | 50 |

编者

Hugo van Kemenade

本文介绍了 Python 3.14 相比 3.13 的新增特性。

完整的详情可参阅 更新日志。

 参见

备注

预发布版用户应当了解到此文档目前处于草稿状态。它将随着 Python 3.14 的发布进程不断更新，因此即使已经阅读过较早的版本也仍然值得再次查看。

1 摘要 -- 发布关键点

Python 3.14 将成为 Python 编程语言的最新稳定版本，包含对语言本身、实现细节及标准库的一系列更新。

对于实现的最大变化包括模板字符串 (**PEP 750**)，标注的延迟求值 (**PEP 649**)，以及一种使用尾调用的新解释器类型。

标准库的变化包括新增 `annotationlib` 模块用于内省和包裹标注 (**PEP 749**)，新增 `compression.zstd` 模块用于 Zstandard 支持 (**PEP 784**)，以及 REPL 中的语法高亮，以及常规的弃用和移除，还有用户友好度和正确性方面的改进。

- **PEP 779**: 自由线程 *Python* 已获官方支持
- **PEP 649** 和 **749**: 标注的延迟求值
- **PEP 734**: 添加多解释器到标准库
- **PEP 741**: *Python* 配置 C API
- **PEP 750**: 模板字符串
- **PEP 758**: 允许不带括号的 *except* 和 *except** 表达式
- **PEP 761**: 停止使用 PGP 签名
- **PEP 765**: 不允许退出 *finally* 代码块的 *return/break/continue*
- 自由线程模式的改进
- **PEP 768**: *Cpython* 的安全外部调试器接口
- **PEP 784**: 添加 Zstandard 到标准库
- 一种新型的解释器
- *PyREPL* 中的语法高亮，以及 *unittest*, *argparse*, *json* 和 *calendar* CLI 中的彩色输出
- 针对实验性即时编译器的二进制发布包

2 不兼容的更改

在 macOS 和 Windows 以外的平台上，用于 `multiprocessing` 和 `ProcessPoolExecutor` 的默认 `start` 方法由 `fork` 改为 `forkserver`。

请参阅(1)和(2)了解详情。

如果你在 `multiprocessing` 或 `concurrent.futures` 中遇到 `NameError` 或 `pickle` 错误，请参阅 `forkserver` 的限制。

解释器会在保证安全的情况下避免某些引用计数修改。这可能导致 `sys.getrefcount()` 和 `Py_REFCNT()` 返回相比之前 Python 版本有所不同的值。请参阅下文了解详情。

3 新的特性

3.1 PEP 779: 自由线程 Python 已获官方支持

自由线程的 Python 构建版现在已被支持而不再是实验性的。这是自由线程 Python 获得官方支持但仍为可选项的阶段 II 的开始。

我们相信该项目走在正确的路径上，我们赞赏各位持续努力的贡献使得自由线程特性已准备好在 Python 社区获得更广泛的接受。

随着这些建议和这个 PEP 被接受，我们 Python 开发者社区应当广泛宣传自由线程现在已成为受支持的 Python 构建选项并在未来继续发展，它不会在没有适当的弃用计划的情况下被移除。

任何过渡到第三阶段（将自由线程作为 Python 的默认或唯一构建）的决定仍未决定，这取决于 CPython 本身和社区内部的许多因素。这个决定是为了将来。

➡ 参见

PEP 779 及其接受。

3.2 PEP 734: 添加多解释器到标准库

CPython 运行时支持在同一个进程中同时运行多个 Python 的副本并且已经这样做了 20 年以上。每个这样的单独副本被称为“解释器”。不过，该特性过去只能通过 C-API 来使用。

通过新增的 `concurrent.interpreters` 模块，这一限制已在 3.14 发布版中被移除。

使用多解释器的做法值得被考虑至少有两个重要的理由：

- 它们支持（对 Python 来说）全新的、用户友好的并发模型
- 真正的多核心并行

在某些应用场景中，软件中的并发机制能够在宏观层面上提高效率并简化软件设计。但与此同时，实现和维护除最简单形式之外的并发逻辑，通常对人脑来说是一项挑战。这一点尤其适用于普通线程（例如：`threading`），其中所有线程之间共享全部内存。

通过使用多个隔离的解释器，您可以利用一类在其他编程语言（如 Smalltalk、Erlang、Haskell 和 Go）中已被证明成功的并发模型，例如 CSP 或演员模型。可以将多个解释器视为线程，但它们之间的资源共享是可选的。

关于多核并行性：从 3.12 版本开始，各个解释器之间已实现足够的隔离，从而可以并行使用。（参见 [PEP 684](#)。）这使得 Python 能够解锁一系列此前受 GIL 限制的 CPU 密集型应用场景。

使用多个解释器在许多方面与 `multiprocessing` 类似，因为它们都提供了相互隔离的逻辑“进程”，默认情况下不共享任何资源，并且可以并行运行。然而，在使用多个解释器时，应用程序将占用更少的系统资源，并能以更高的效率运行（因为它仍处于同一个进程内）。可以将多个解释器看作是：拥有进程级别的隔离性，同时具备线程级别的执行效率。

尽管该特性已经存在数十年，但由于认知度较低且缺乏标准库模块的支持，多个解释器并未被广泛使用。因此，目前它仍存在一些显著的限制。不过，随着这一特性终于开始走向主流，这些限制将得到显著改善。

当前限制：

- 启动每个解释器尚未经过优化。
- 每个解释器目前占用的内存比实际需要的更多（我们下一步将重点推进解释器之间的内部共享，以优化这一问题）。
- 解释器之间真正实现对象或其他数据共享的选项还很有限（除了 `memoryview`）。
- PyPI 上的许多扩展模块目前尚不兼容多个解释器（标准库中的扩展模块是兼容的）。
- 目前，针对使用多个隔离解释器编写应用程序的方法，对大多数 Python 用户来说仍较为陌生。

这些限制的影响将取决于未来 CPython 的改进程度、解释器的使用方式，以及社区通过 PyPI 包所解决的问题。根据具体的应用场景，这些限制可能并不会造成太大影响，因此不妨尝试一下！

此外，未来的 CPython 版本将进一步减少甚至消除相关开销，并提供一些目前在 PyPI 上不太合适的工具。在此期间，大多数限制也可以通过扩展模块来解决，这意味着 PyPI 包可以填补 3.14 乃至回溯到 3.12（此时解释器终于实现真正隔离并停止共享 GIL）之间的功能缺口。同样地，我们预计 PyPI 上将逐渐出现基于多解释器构建的高级抽象库。

关于扩展模块，目前正在进行一些 PyPI 项目的更新，以及对 Cython、pybind11、nanobind 和 PyO3 等工具的支持工作。有关隔离扩展模块的具体步骤，可参考 [isolating-extensions-howto](#)。模块的隔离与支持自由线程所需的工作有大量重叠，因此社区在该领域的持续努力将有助于加快对多个解释器的支持进程。

在 3.14 中还增加了：`concurrent.futures.InterpreterPoolExecutor`。

➡ 参见

PEP 734.

3.3 PEP 750：模板字符串

模板字符串（t-字符串）是 f-字符串的通用化，使用 `t` 代替 `f` 前缀。与前者求值为 `str` 不同，t-字符串将求值为新增的 `string.templatelib.Template` 类型：

```
from string.templatelib import Template

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

随后 `template` 可以被用于操作该模板的结构以产生 `str` 或字符串型结果的函数。例如，对输入进行无害化：

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

再比如，根据数据生成 HTML 属性：

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

与使用 f-string 相比，`html` 函数能够访问包含原始信息的模板属性：静态字符串、插值表达式，以及来自原始作用域的值。与现有的模板系统不同，t-string 是基于广为人知的 f-string 语法规则构建的。因此，模板系统能够更好地受益于 Python 的工具生态，因为它们在语言特性、语法结构、作用域规则等方面都更加贴近 Python 本身。

编写模板处理程序非常简单：

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

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

使用此特性，开发者可以编写模板系统来对 SQL 做无害化处理，执行安全的 shell 操作，改进日志记录，处理 Web 开发中的现代概念（HTML、CSS 等等），以及实现轻量的、定制的业务 DSL。

（由 Jim Baker、Guido van Rossum、Paul Everitt、Koudai Aono、Lysandros Nikolaou、Dave Peck、Adam Turner、Jelle Zijlstra、Bénédikt Tran 和 Pablo Galindo Salgado 在 [gh-132661](#) 中贡献。）

➡ 参见

PEP 750。

3.4 PEP 768: Cpython 的安全外部调试器接口

PEP 768 引入了一个零开销的调试接口，允许调试器和性能分析工具安全地附加到正在运行的 Python 进程上。这是对 Python 调试能力的重要增强，使调试器可以避免使用不安全的替代方案。有关该特性如何用于实现新版 pdb 模块的远程附加功能，请参见 [下面](#)。

新的接口在不修改解释器正常执行路径或增加运行时开销的前提下，提供了安全的调试代码附加点。这使得工具能够在不停止或重启 Python 应用程序的情况下，实时地对其进行检查和交互——这对于高可用性系统和生产环境至关重要。

为方便起见，CPython 通过 sys 模块中的 `sys.remote_exec()` 函数实现该接口：

```
sys.remote_exec(pid, script_path)
```

此函数允许发送 Python 代码以便在目标进程中的下一个安全执行点上执行。不过，工具作者也可以直接实现在 PEP 中描述的协议，它详细讲解了用于安全附加到运行进程的底层机制。

以下是一个检查运行中 Python 进程内对象类型的简单示例：

```
import os
import sys
import tempfile

# 创建一个临时脚本
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:
    # 在进程 PID 1234 中执行：
    print("Behold! An offering:")
    sys.remote_exec(1234, script_path)
finally:
    os.unlink(script_path)
```

该调试接口在设计时已充分考虑安全性，并包含多种访问控制机制：

- PYTHON_DISABLE_REMOTE_DEBUG 环境变量。
- -X disable-remote-debug 命令行选项。
- --without-remote-debug 配置标志，用于在构建时完全禁用此功能。

关键实现细节在于：该接口复用了解释器现有的求值循环和安全点机制，既确保正常执行时零开销，又为外部进程提供了可靠的调试操作协调方式。

（由 Pablo Galindo Salgado，Matt Wozniski 和 Ivona Stojanovic 在 [gh-131591](#) 中贡献）

➡ 参见

PEP 768。

3.5 PEP 784: 添加 Zstandard 到标准库

新推出的 `compression` 包包含以下模块: `compression.lzma`、`compression.bz2`、`compression.gzip` 和 `compression.zlib`，它们分别重新导出了 `lzma`、`bz2`、`gzip` 和 `zlib` 模块。今后，`compression` 下的新导入名称将成为这些压缩模块的标准导入方式。不过，现有的模块名称尚未被弃用。任何对现有压缩模块的弃用或删除都不会早于 3.14 版本发布后的五年。

新引入的 `compression.zstd` 模块通过绑定 Meta 的 `zstd` 库提供了 Zstandard 格式的压缩和解压 API。Zstandard 是一种被广泛采用、高效且快速的压缩格式。除了 `compression.zstd` 中引入的 API 外，对 Zstandard 压缩归档文件的读写支持也已添加到 `tarfile`、`zipfile` 和 `shutil` 模块中。

下面是一个使用新模块压缩数据的示例：

```
from compression import zstd
import math

data = str(math.pi).encode() * 20

compressed = zstd.compress(data)

ratio = len(compressed) / len(data)
print(f"达到的压缩比为 {ratio}")
```

可以看出，该 API 与 `lzma` 和 `bz2` 模块的 API 类似。

(由 Emma Harper Smith、Adam Turner、Gregory P. Smith、Tomas Roun、Victor Stinner 和 Rogdham 在 [gh-132983](#) 中贡献。)

➡ 参见

PEP 784。

3.6 使用 PDB 远程附加到正在运行的 Python 进程

`pdb` 模块现在支持通过新的 `-p PID` 命令行选项远程附加到正在运行的 Python 进程：

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

该操作将连接到指定 PID 的 Python 进程，并允许您进行交互式调试。请注意，由于 Python 解释器的工作原理，当附加到阻塞在系统调用或等待 I/O 的远程进程时，调试功能只有在执行下一条字节码指令或进程收到信号时才会生效。

该功能使用 [PEP 768](#) 和 `sys.remote_exec()` 函数来附加到远程进程，并向其发送 PDB 命令。

(由 Matt Wozniski 和 Pablo Galindo 在 [gh-131591](#) 中贡献。)

➡ 参见

PEP 768。

3.7 PEP 758 –允许不带括号的 `except` 和 `except*` 表达式

`except` 和 `except*` 表达式现在允许在存在多个异常类型且未使用 `as` 子句时省略括号。例如，以下表达式现在有效：

```
try:
    connect_to_server()
except TimeoutError, ConnectionRefusedError:
    print("遇到网络问题。")
```

(续下页)


```
# 同样适用于 except* (针对异常组):

try:
    connect_to_server()
except* TimeoutError, ConnectionRefusedError:
    print("遇到网络问题。")
```

请参阅 [PEP 758](#) 获取更多细节。

(由 Pablo Galindo 和 Brett Cannon 在 [gh-131831](#) 中贡献。)

参见

[PEP 758](#)。

3.8 PEP 649 和 749：注解的延迟求值

在函数、类和模块上的注解不再被立即求值。相反，注解会被存储在专用的注解函数中，仅在必要时进行求值（除非使用了 `from __future__ import annotations`）。这一特性在 [PEP 649](#) 和 [PEP 749](#) 中进行了规范。

这一改进旨在使 Python 中的注解在大多数情况下性能更高、更易用。定义注解的运行时开销被降至最低，同时仍可在运行时内省注解。如果注解包含前向引用，也不再需要将其包裹在字符串中。

新引入的 `annotationlib` 模块提供了检查延迟注解的工具。注解可以通过以下格式进行求值：VALUE 格式（将注解求值为运行时值，类似于早期 Python 版本的行为）、FORWARDREF 格式（用特殊标记替换未定义名称）、STRING 格式（以字符串形式返回注解）。

以下示例展示了这些格式的具体行为：

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

注解代码的影响

如果在代码中定义注解（例如用于静态类型检查器），那么这一变更可能不会产生影响：你可以继续保持与之前 Python 版本相同的注解书写方式。

你很可能可以移除注解中的引号字符串（这些通常用于前向引用）。同样地，如果使用 `from __future__ import annotations` 来避免在注解中书写字符串，当你仅支持 Python 3.14 及更新版本时，很可能可以移除该导入。不过，如果依赖读取注解的第三方库，这些库可能需要相应修改以支持无引号注解，才能正常工作。

访问 `__annotations__` 的影响

如果代码会读取对象的 `__annotations__` 属性，你可能需要进行调整以支持依赖注解延迟求值的代码。例如，可以像 `dataclasses` 模块现在所做的那样，使用 `annotationlib.get_annotations()` 函数并指定 FORWARDREF 格式。

外部包 `typing_extensions` 提供了 `annotationlib` 模块部分功能的向后兼容实现，包括 `Format` 枚举和 `get_annotations()` 函数。这些可用于编写跨版本代码，以利用 Python 3.14 中的新行为。

相关变更

Python 3.14 的变更旨在重构 `__annotations__` 的运行时行为，同时最小化对以下两类代码的影响：(1) 源代码中包含注解的代码 (2) 读取 `__annotations__` 的代码。不过，如果依赖注解行为的未文档化细节或标准库中的私有函数，你的代码可能在 Python 3.14 中存在多种兼容性问题。为确保代码的未来兼容性，请仅使用 `annotationlib` 模块中已文档化的功能。

特别需要注意的是，不要直接从类型对象的命名空间字典属性中读取注解。在类构造期间应使用 `annotationlib.get_annotate_from_class_namespace()` 函数，之后则使用 `annotationlib.get_annotations()` 函数。

在之前的版本中，有时可以通过带注解类的实例访问类注解。这种行为属于未文档化的意外实现，在 Python 3.14 中将不再有效。

`from __future__ import annotations`

在 Python 3.7 中，**PEP 563** 引入了 `from __future__ import annotations` 指令，该指令会将所有注解转换为字符串。此指令现已被视为弃用，预计将在未来的 Python 版本中移除。不过，在 Python 3.13（最后一个不支持注解延迟求值的 Python 版本）于 2029 年结束生命周期之前，这一移除操作不会实施。在 Python 3.14 中，使用 `from __future__ import annotations` 的代码行为保持不变。

（由 Jelle Zijlstra 在 [gh-119180](#) 中贡献；**PEP 649** 由 Larry Hastings 编写。）

参见

PEP 649 和 **PEP 749**。

3.9 改进的错误消息

- 解释器现在能在检测到 Python 关键字拼写错误时提供有用的建议。当遇到与 Python 关键字高度相似的单词时，解释器将在错误信息中建议正确的关键字。该功能可帮助程序员快速识别和修复常见的输入错误。例如：

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

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

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

>>> raise ValueError("Error")
Traceback (most recent call last):
  File "<stdin>", line 1
    raise ValueError("Error")
```

(续下页)

(接上页)

```
^^^^^^
SyntaxError: invalid syntax. Did you mean 'raise'?
```

虽然该功能主要针对最常见的拼写错误情况，但某些变体的拼写错误仍可能导致常规语法错误。(由 Pablo Galindo 在 [gh-132449](#) 中贡献。)

- 当一个解包赋值因变量数量不匹配而失败时，错误消息现在会在更多情况下显示实际接收到的数值数量。(由 Tushar Sadhwani 在 [gh-122239](#) 中贡献。)

```
>>> 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)
```

- 跟在 else 块后的 elif 语句现在会触发特定的错误提示。(由 Steele Farnsworth 在 [gh-129902](#) 中贡献。)

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

- 当以下语句 (pass、del、return、yield、raise、break、continue、assert、import、from) 被传递到 else 后的 if_expr，或者 pass、break、continue 之一被传递到 if 之前时，错误消息会明确标出需要 expression 的位置。(由 Sergey Miryanov 在 [gh-129515](#) 中贡献。)

```
>>> 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
```

- 当检测到未正确闭合的字符串时，错误消息会提示该字符串可能是字符串的一部分。(由 Pablo Galindo 在 [gh-88535](#) 中贡献。)

```
>>> "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?
```

- 当字符串前缀不兼容时，错误提示现在会明确显示哪些前缀存在冲突。(由 Nikita Sobolev 在 [gh-133197](#) 中贡献。)

```
>>> ub'abc'
File "<python-input-0>", line 1
    ub'abc'
```

(续下页)

```
^^
SyntaxError: 'u' and 'b' prefixes are incompatible
```

- 在以下场景中使用不兼容目标的 `as` 语句时，错误提示已得到改进：

- 导入：`import ... as ...`
- From 导入：`from ... import ... as ...`
- Except 处理器：`except ... as ...`
- 模式匹配 `case` 语句：`case ... as ...`

(由 Nikita Sobolev 在 [gh-123539](#)、[gh-123562](#) 和 [gh-123440](#) 中贡献。)

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

- 尝试向 `dict` 或 `set` 添加不可哈希类型的实例时，错误提示信息已改进。(由 CF Bolz-Tereick 和 Victor Stinner 在 [gh-132828](#) 中贡献。)

```
>>> 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 配置 C API

新增 `PyInitConfig` C API 用于配置 Python 初始化过程，无需依赖 C 结构体，同时支持未来进行 ABI 兼容性变更。

通过添加 `PyInitConfig_AddModule()` 函数来完成 [PEP 587](#) 规范中定义的 `PyConfig` C API，该函数可用于添加内置扩展模块；该特性此前被称为“`inittab`”。

新增 `PyConfig_Get()` 和 `PyConfig_Set()` 函数，用于获取和设置当前运行时配置。

PEP 587《Python 初始化配置》统一了所有配置 Python 初始化的方式。该 PEP 进一步将 Python 预初始化配置与初始化配置整合至单一 API 中。此外，为进一步简化 API 设计，本 PEP 仅提供“嵌入式 Python”这一种选择，而非 PEP 587 中的“Python”和“隔离模式”双选项方案。

较低层级的 PEP 587 `PyConfig` API 仍保留可用，适用于需要与 CPython 实现细节保持较高耦合度的使用场景（例如模拟 CPython CLI 的完整功能，包括其配置机制）。

(由 Victor Stinner 在 [gh-107954](#) 中贡献。)

➡ 参见

[PEP 741](#)。

3.11 asyncio 内省能力

新增了一个通过异步任务检查运行中 Python 进程的命令行接口，可通过以下方式使用：

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

该工具会检查指定的进程 ID (PID)，并显示当前正在运行的 asyncio 任务信息。其输出包含一个任务表格：以扁平列表形式展示所有任务及其名称、协程堆栈信息，以及各任务正在等待的其他任务。

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

该工具获取相同的信息，但会以可视化异步调用树的形式呈现，采用层级结构展示协程关系。此命令特别适用于调试长时间运行或卡死的异步程序，能帮助开发者快速定位程序阻塞位置、识别待处理任务，以及理清协程间的调用链关系。

例如给定以下代码：

```
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())
```

在运行中的进程上执行该新工具将生成如下表格：

```
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__ | 0x0 | | |
| 1935500 | 0x7fc930c18230 | Sundowning | TaskGroup._aexit -> TaskGroup.__aexit__ | -> main | Task-1 | |
| 1935500 | 0x7fc93173fa50 | TMBTE | TaskGroup._aexit -> TaskGroup.__aexit__ | -> album TaskGroup._aexit -> TaskGroup.__aexit__ | -> main Task-1 | |
| 1935500 | 0x7fc93173fdf0 | TNDNBTG | sleep -> play | TaskGroup._aexit -> TaskGroup.__aexit__ | -> album Sundowning | |
| 1935500 | 0x7fc930d32510 | Levitate | sleep -> play | TaskGroup._aexit -> TaskGroup.__aexit__ | -> album Sundowning | |
| 1935500 | 0x7fc930d32890 | DYWTYLM | sleep -> play | TaskGroup._aexit -> TaskGroup.__aexit__ | -> album TMBTE | |

(续下页)

(接上页)

```
1935500      0x7fc93161ec30      Aqua Regia      sleep -> play
↳      TaskGroup.__aexit__ -> TaskGroup.__aexit__ -> album      TMBTE
↳0x7fc93173fa50
```

或生成如下树状结构：

```
python -m asyncio pstree 12345

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

当检测到异步等待图中存在循环引用（可能表明存在编程问题）时，该工具将报错并列出阻碍树形结构构建的循环路径：

```
python -m asyncio pstree 12345

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

cycle: Task-2 → Task-3 → Task-2
```

（由 Pablo Galindo、Łukasz Langa、Yury Selivanov 和 Marta Gomez Macias 在 [gh-91048](#) 中贡献。）

3.12 一种新型的解释器

CPython 新增了一种全新的解释器类型。它通过在实现各个 Python 操作码的小型 C 函数之间使用尾调用来运行，而不是使用单个大型 C case 语句。对于某些较新的编译器，这种解释器能带来显著的性能提升。我们在测试机器上的初步数据显示，Python 代码的运行速度最高可提升 30%，在不同平台和架构上运行 `pyperformance` 基准测试的几何平均提速为 3% 到 5%。作为对比的基线是使用 Clang 19 编译、未启用该新解释器的 Python 3.14。

此解释器目前仅适用于 x86-64 和 AArch64 架构上的 Clang 19 及更高版本。不过，我们预计 GCC 的未来版本也将支持此功能。

此功能目前为可选功能。我们强烈建议在新解释器中启用配置文件引导优化，因为这是我们测试过的唯一配置，可以验证其性能提升。有关如何构建 Python 的更多信息，请参阅 `--with-tail-call-interp`。

备注

这不应与 Python 函数的 **尾调用优化** 相混淆，该特性目前在 CPython 中尚未实现。

这一新型解释器属于 CPython 解释器的内部实现细节，完全不会改变 Python 程序的可见行为。它能提升程序性能，但不会引发任何其他变更。

⚠ 注意

本节原先报告的几何平均加速比为 9-15%，但这一数字现已谨慎地下调为 3% 到 5%。尽管我们预计实际性能提升可能高于这一范围，但由于在 Clang/LLVM 19 中发现了一个“编译器漏洞”<https://github.com/llvm/llvm-project/issues/106846>，我们对性能估算变得更加保守。该漏洞会导致常规解释器变慢，我们在早期并未意识到这一点，导致报告的性能数据不准确。对此我们深感抱歉，之前的数据实际上仅对 LLVM v19.1.x 和 v20.1.0 有效。目前，这个漏洞已经在 LLVM v20.1.1 和即将发布的 v21.1 中被修复，但在 LLVM v19.1.x 和 v20.1.0 中将不会得到修复。因此，使用这些 LLVM 版本进行的基准测试可能会生成不准确的数据。（感谢 Nelson Elhage 揭示了这个问题。）

（由 Ken Jin 在 [gh-128563](#) 中贡献，CPython 实现方案融合了 Mark Shannon、Garrett Gu、Haoran Xu 和 Josh Haberman 的设计理念。）

3.13 自由线程模式

自由线程模式（**PEP 703**，最初在 3.13 版本引入）已得到显著改进。PEP 703 中描述的实施方案已完成，包括 C API 变更，且解释器中的临时解决方案已被更持久的方案取代。专业化自适应解释器（**PEP 659**）现已在自由线程模式中启用，结合众多其他优化措施，其性能得到大幅提升。当前自由线程模式对单线程代码的性能影响已降至约 5-10%，具体取决于平台和使用的 C 编译器。

此项工作由众多贡献者共同完成：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 以及其他很多人。

部分贡献者受雇于 Meta，该公司持续为本项目提供了大量工程技术资源支持。

自 3.14 版本起，在 Windows 平台上为 CPython 的自由线程构建编译扩展模块时，构建后端现在需要显式指定预处理器变量 `Py_GIL_DISABLED`，因为该变量将不再由 C 编译器自动确定。对于运行中的解释器，可通过 `sysconfig.get_config_var()` 查询编译时使用的配置。

新增了一个标志位 `context_aware_warnings`。该标志在自由线程构建中默认为真（`True`），在启用 GIL 的构建中默认为假（`False`）。当标志为真时，`warnings.catch_warnings` 上下文管理器会使用上下文变量来处理警告过滤器，这使得该管理器在多线程或异步任务场景下能够保持可预测的行为。

新增了一个标志位 `thread_inherit_context`。该标志在自由线程构建中默认为真（`True`），在启用 GIL 的构建中默认为假（`False`）。当标志为真时，通过 `threading.Thread` 创建的线程会继承 `start()` 调用者的 `Context()` 副本。最重要的是，这使得由 `catch_warnings` 建立的警告过滤上下文能够被该上下文中启动的线程（或 `asyncio` 任务）“继承”。该标志还会影响其他使用上下文变量的模块，例如 `decimal` 上下文管理器。

3.14 PyREPL 的语法高亮功能

默认的 `interactive` 式 shell 现已支持实时 Python 语法高亮。除非设置了 `PYTHON_BASIC_REPL` 环境变量或使用了任何禁用颜色的环境变量，否则该功能默认启用。详情请参阅 [using-on-controlling-color](#)。

语法高亮的默认配色方案追求高对比度，并仅使用 4 位 VGA 标准 ANSI 颜色代码以确保最大兼容性。可通过实验性 `API_colorize.set_theme()` 自定义主题，该 API 既支持交互式调用，也可在 `PYTHONSTARTUP` 脚本中使用。

（由 Łukasz Langa 在 [gh-131507](#) 中贡献。）

3.15 实验性即时编译器 (JIT) 的二进制发布版本

官方 macOS 和 Windows 发布版二进制文件现已包含 实验性即时编译 (JIT) 器。虽然不建议在生产环境中使用，但可通过设置环境变量 `PYTHON_JIT=1` 进行测试。下游源码构建和再分发方可使用配置选项 `--enable-experimental-jit=yes-off` 实现类似行为。

该 JIT 编译器尚处于早期开发阶段，性能表现存在波动：启用后根据工作负载不同，可能产生 10% 的性能下降至 20% 的性能提升。为便于测试评估，`sys._jit` 命名空间提供了一组内省函数：`sys._jit.is_available()` 用于检测当前可执行文件是否支持 JIT 编译，而 `sys._jit.is_available()` 则可判断当前进程是否已启用 JIT 编译功能。

当前最显著的功能缺失是原生调试器（如 `gdb`）和性能分析工具（如 `perf`）无法展开 JIT 调用栈（而 Python 原生调试器如 `pdb` 和性能分析器如 `profile` 仍可无需修改直接使用）。此外，自由线程构建版本暂不支持 JIT 编译功能。

如遇任何错误或严重性能退化问题，请务必提交报告！

➡ 参见

PEP 744

3.16 并发安全的警告控制

`warnings.catch_warnings` 上下文管理器现在支持通过上下文变量处理警告过滤器，该功能需通过设置 `context_aware_warnings` 标志启用（可使用 `-x` 命令行选项或环境变量）。当结合多线程或异步任务使用 `catch_warnings` 时，此举能提供可预测的警告控制。该标志在自由线程构建中默认为真 (True)，在启用 GIL 的构建中默认为假 (False)。

（由 Neil Schemenauer 和 Kumar Aditya 在 [gh-130010](#) 中贡献。）

3.17 增量式垃圾回收

循环垃圾回收器现在采用增量式处理。这意味着对于较大的堆内存，最大暂停时间将减少一个数量级或更多。

现在垃圾回收机制仅包含两个代际：新生代和老年代。当未直接调用 `gc.collect()` 函数时，垃圾回收器的触发频率会稍有降低。而在调用时，它会回收新生代以及部分老年代对象（即增量回收），而非一次性回收一个或多个完整的代际。

`gc.collect()` 函数的行为略有变化：

- `gc.collect(1)`：执行一次增量式垃圾回收，而非专门回收第 1 代对象。
- 对 `gc.collect()` 的其他调用保持不变。

（由 Mark Shannon 在 [gh-108362](#) 中贡献。）

4 平台支持

- **PEP 776**: Emscripten 现在是官方支持的 **第 3 层级** 平台。作为此项工作的一部分，在 Emscripten libc 中有 25 以上的程序缺陷被修复。Emscripten 现在包括对 `ctypes`, `termios` 和 `fcntl` 的支持，以及对 PyREPL 的实验性支持。

（由 R. Hood Chatham 在 [gh-127146](#), [gh-127683](#) 和 [gh-136931](#) 中贡献。）

5 其他语言特性修改

- 默认的 interactive shell 现已支持导入自动补全功能。具体表现为：输入 `import foo` 后按下 `<tab>` 键会建议以 `foo` 开头的模块；类似地，输入 `from foo import b` 则会建议以 `b` 开头的 `foo` 子模块。需注意当前暂不支持模块属性的自动补全。（由 Tomas Roun 在 [gh-69605](#) 中贡献。）

- 内置函数 `map()` 现在新增了一个可选的仅限关键字参数 `strict`（与 `zip()` 类似），用于校验所有可迭代对象长度是否一致。（由 Wannes Boeykens 在 [gh-119793](#) 中贡献。）
- 现在即使代码被 `-O` 命令行选项优化移除，仍能检测出 `await` 和异步推导式的错误用法。例如，执行 `python -O -c 'assert await 1'` 现在会触发 `SyntaxError`。（由 Jelle Zijlstra 在 [gh-121637](#) 中贡献。）
- 现在即使代码被 `-O` 命令行选项优化移除，对 `__debug__` 的写入操作仍会被检测到。例如，执行 `python -O -c 'assert (__debug__ := 1)'` 现在会触发 `SyntaxError`。（由 Irit Katriel 在 [gh-122245](#) 中贡献。）
- 新增类方法 `float.from_number()` 和 `complex.from_number()`，用于将数值分别转换为 `float` 或 `complex` 类型。若参数为字符串则会引发错误。（由 Serhiy Storchaka 在 [gh-84978](#) 中贡献。）
- 实现了符合 C99 以来 C 标准规范的实数与复数混合运算规则。（由 Sergey B Kirpichev 在 [gh-69639](#) 中贡献。）
- 现在 Windows 平台已支持所有 Windows 代码页作为“cpXXX”编解码器使用。（由 Serhiy Storchaka 在 [gh-123803](#) 中贡献。）
- `super` 对象现在支持序列化和复制操作。（由 Serhiy Storchaka 在 [gh-125767](#) 中贡献。）
- `memoryview` 类型现已支持下标，使其成为 generic type。（由 Brian Schubert 在 [gh-126012](#) 中贡献。）
- 在新式字符串格式化（通过 `format()` 或 f-strings 实现）的浮点数表示类型中，现支持使用下划线和逗号作为小数部分的千位分隔符。（由 Sergey B Kirpichev 在 [gh-87790](#) 中贡献。）
- `bytes.fromhex()` 和 `bytearray.fromhex()` 方法现在支持 ASCII 格式的 bytes 及字节型对象作为输入参数。（由 Daniel Pope 在 [gh-129349](#) 中贡献。）
- 在正则表达式中，现支持将 `\z` 作为 `\Z` 的同义符使用。与行为存在微妙差异的 `\Z` 不同，`\z` 在其他多种正则表达式引擎中具有明确无歧义的解析方式。（由 Serhiy Storchaka 在 [gh-133306](#) 中贡献。）
- 在正则表达式中，`\B` 现在可以匹配空输入字符串，其行为现在始终与 `\b` 相反。（由 Serhiy Storchaka 在 [gh-124130](#) 中贡献。）
- iOS 和 macOS 应用程序现在可配置为将 `stdout` 和 `stderr` 输出内容重定向至系统日志。（由 Russell Keith-Magee 在 [gh-127592](#) 中贡献。）
- iOS 测试平台现已支持在测试运行时实时流式传输测试输出，并可用于运行除 CPython 之外的其他项目测试套件。（由 Russell Keith-Magee 在 [gh-127592](#) 中贡献。）
- 三参数 `pow()` 现在会在必要时尝试调用 `__rpow__()` 方法。此前该方法仅在双参数 `pow()` 和二元幂运算符中被调用。（由 Serhiy Storchaka 在 [gh-130104](#) 中贡献。）
- 新增基于 **HACL*** 项目形式化验证代码的 HMAC (**RFC 2104**) 内置实现。当 OpenSSL 的 HMAC 实现不可用时，该实现将作为备用方案。（由 Bénédict Tran 在 [gh-99108](#) 中贡献。）
- 导入时间分析标志现可通过新增的 `-X importtime=2` 选项追踪已加载（‘缓存’）模块。当导入此类模块时，`self` 和 `cumulative` 时间值将被替换为字符串 `cached`。当前规定 `-X importtime` 参数值大于 2 的选项保留供未来使用。（由 Noah Kim 和 Adam Turner 在 [gh-118655](#) 中贡献。）
- 当从纯 C 类型派生子类时，若子类未显式重写相关方法，新建类型的 C 槽位将不再被替换为封装版本。（由 Tomasz Pytel 在 [gh-132329](#) 中贡献。）
- 命令行选项 `-c` 现在会在执行前自动对其代码参数进行去缩进处理，该行为与 `textwrap.dedent()` 函数保持一致。（由 Jon Crall 和 Steven Sun 在 [gh-103998](#) 中贡献。）
- 当使用 `async with` 而非 `with` 进入支持同步上下文管理器协议的对象时，错误提示信息已改进。反之，对于异步上下文管理器协议的情况也做了相应优化。（由 Bénédict Tran 在 [gh-128398](#) 中贡献。）
- `-J` 不再是 Jython 的保留标志，目前不具任何特殊含义。（由 Adam Turner 在 [gh-133336](#) 中贡献。）
- 内置的 `int()` 不再委托给 `__trunc__()`。希望支持转换为 `int()` 的类必须实现 `__int__()` 或 `__index__()`。（由 Mark Dickinson 在 [gh-119743](#) 中贡献。）
- 在布尔上下文中使用 `NotImplemented` 现在会引发 `TypeError` 异常。该用法自 Python 3.9 起已引发 `DeprecationWarning` 弃用警告。（由 Jelle Zijlstra 在 [gh-118767](#) 中贡献。）

5.1 PEP 765: 禁止在 `finally` 代码块中使用会跳出该块的 `return/break/continue` 语句

编译器在检测到 `return`、`break` 或 `continue` 语句跳出 `finally` 代码块时，将触发 `SyntaxWarning`。此项变更遵循 [PEP 765](#) 规范。

6 新增模块

- `annotationlib`: 用于自省 注解的模块，详见 [PEP 749](#) 规范说明。（由 Jelle Zijlstra 在 [gh-119180](#) 中贡献。）

7 改进的模块

7.1 `argparse`

- `argparse.ArgumentParser` 的程序名称默认值现在会反映 Python 解释器定位 `__main__` 模块代码的方式。（由 Serhiy Storchaka 和 Alyssa Coghlan 在 [gh-66436](#) 中贡献。）
- `argparse.ArgumentParser` 新增可选形参 `suggest_on_error`，可在用户输入错误时提供参数选项及子解析器名称的建议。（由 Savannah Ostrowski 在 [gh-124456](#) 中贡献。）
- 为帮助文本启用颜色显示，你可以通过向 `argparse.ArgumentParser` 传递可选的 `color` 形参来禁用此功能。此外，该功能还可以通过 环境变量进行控制。（由 Hugo van Kemenade 在 [gh-130645](#) 中贡献。）

7.2 `ast`（抽象语法树）

- 新增 `ast.compare()` 函数用于比较两个抽象语法树（AST）。（由 Batuhan Taskaya 和 Jeremy Hylton 在 [gh-60191](#) 中贡献。）
- 新增对 AST 节点 `copy.replace()` 操作的支持。（由 Bénédict Tran 在 [gh-121141](#) 中贡献。）
- 在优化级别为 2 时，文档字符串现会从优化后的 AST 中移除。（由 Irit Katriel 在 [gh-123958](#) 中贡献。）
- AST 节点的 `repr()` 输出现在包含更多信息。（由 Tomas Roun 在 [gh-116022](#) 中贡献。）
- 当传入 AST 作为输入时，`ast.parse()` 现在会始终验证根节点类型是否合规。（由 Irit Katriel 在 [gh-130139](#) 中贡献。）
- 新增命令行接口选项 `--feature-version`、`--optimize` 和 `--show-empty`。（由 Semyon Moroz 在 [gh-133367](#) 中贡献。）

7.3 `asyncio`

- 名为 `create_task()` 的函数及方法现在接受任意关键字参数列表，所有关键字参数都将传递给 `Task` 构造器或自定义任务工厂（详见 `set_task_factory()`）。`name` 和 `context` 关键字参数不再具有特殊处理逻辑——名称现在应通过工厂的 `name` 关键字参数设置，而 `context` 可设为 `None`。
这会影响下列函数和方法: `asyncio.create_task()`、`asyncio.loop.create_task()`、`asyncio.TaskGroup.create_task()`。（由 Thomas Grainger 在 [gh-128307](#) 中贡献。）
- 有两个新的实用函数用于自省和打印程序的调用图: `capture_call_graph()` 和 `print_call_graph()`。（由 Yury Selivanov、Pablo Galindo Salgado 和 Łukasz Langa 在 [gh-91048](#) 中贡献。）

7.4 `calendar`（日历）

- 默认情况下，`calendar` 模块的 命令行文本输出会以彩色高亮显示当日日期，该功能可通过 环境变量控制。（由 Hugo van Kemenade 在 [gh-128317](#) 中贡献。）

7.5 concurrent.futures

- 新增 `InterpreterPoolExecutor` 类，该功能向 Python 代码暴露“子解释器”能力（同一进程中的多个 Python 解释器）。此实现与 [PEP 734](#) 提案的 API 相互独立。（由 Eric Snow 在 [gh-124548](#) 中贡献。）
- 默认的 `ProcessPoolExecutor` 的启动方法在非 macOS 和 Windows 平台上已从 `fork` 变更为 `forkserver`，在 macOS 和 Windows 平台上仍保持原有的 `spawn` 方式。

如需使用与线程不兼容的 `fork` 启动方法，必须通过向 `ProcessPoolExecutor` 提供 `mp_context` 多进程上下文参数来显式指定。

请参阅 `forkserver` 限制说明了解与 `fork` 方法的差异信息，以及此项变更对存在以下情况的现有代码可能产生的影响：(1) 使用可变全局共享变量 (2) 包含无法被 `pickle` 自动序列化的共享对象。

（由 Gregory P. Smith 在 [gh-84559](#) 中贡献）

- 新增 `concurrent.futures.ProcessPoolExecutor.terminate_workers()` 和 `concurrent.futures.ProcessPoolExecutor.kill_workers()` 方法，用于终止或强制终止给定进程池中的所有存活工作进程。（由 Charles Machalow 在 [gh-130849](#) 中贡献。）
- 为 `concurrent.futures.Executor.map()` 新增可选形参 `buffer_size`，用于限制已提交但尚未产出结果的任务数量。当缓冲区满时，对 `iterables` 的迭代将暂停，直至缓冲区产出结果。（由 Enzo Bonnal 和 Josh Rosenberg 在 [gh-74028](#) 中贡献。）

7.6 configparser

- 安全修复：不再写入无法读取的配置文件。尝试使用 `configparser.ConfigParser.write()` 写入包含定界符或以节头模式开头的键时，将触发 `configparser.InvalidWriteError`。（由 Jacob Lincoln 在 [gh-129270](#) 中贡献。）

7.7 contextvars

- `contextvars.Token` 现已支持上下文管理器协议。（由 Andrew Svetlov 在 [gh-129889](#) 中贡献。）

7.8 ctypes

- `Structure` 和 `Union` 中的位字段布局现已更紧密匹配平台默认行为（GCC/Clang 或 MSVC），特别值得注意的是字段间不再出现重叠现象。（由 Matthias Görgens 在 [gh-97702](#) 中贡献。）
- 现在可通过设置 `Structure._layout_` 类属性来匹配非默认 ABI。（由 Petr Viktorin 在 [gh-97702](#) 中贡献。）
- `Structure/Union` 的字段描述符类现以 `CField` 形式提供，并新增了辅助调试和内省的属性。（由 Petr Viktorin 在 [gh-128715](#) 中贡献。）
- 在 Windows 平台上，`COMError` 异常现已公开。（由 Jun Komoda 在 [gh-126686](#) 中贡献。）
- 在 Windows 平台上，`CopyComPointer()` 函数现已公开。（由 Jun Komoda 在 [gh-127275](#) 中贡献。）
- 新增 `ctypes.memoryview_at()` 函数，用于创建引用指定指针和长度的 `memoryview` 对象。该函数与 `ctypes.string_at()` 类似，但避免了缓冲区复制，通常在实现接收动态大小缓冲区的纯 Python 回调函数时特别有用。（由 Rian Hunter 在 [gh-112018](#) 中贡献。）
- 复数类型 `c_float_complex`、`c_double_complex` 和 `c_longdouble_complex` 现已在编译器和 `libffi` 库均支持 C 语言复数类型时可用。（由 Sergey B Kirpichev 在 [gh-61103](#) 中贡献。）
- 新增 `ctypes.util.dlllist()` 函数，用于列出当前进程已加载的共享库。（由 Brian Ward 在 [gh-119349](#) 中贡献。）
- 将 `ctypes.POINTER()` 类型缓存从全局内部缓存（`_pointer_type_cache`）迁移至对应 `ctypes` 类型的 `ctypes._CData.__pointer_type__` 属性，此举可防止某些情况下缓存无限增长。（由 Sergey Miryanov 在 [gh-100926](#) 中贡献。）
- `ctypes.py_object` 类型现已支持下标，使其成为 `generic type`。（由 Brian Schubert 在 [gh-132168](#) 中贡献。）

- `ctypes` 现已支持 自由线程构建。(由 Kumar Aditya 和 Peter Bierma 在 [gh-127945](#) 中贡献。)

7.9 curses

- 新增 `assume_default_colors()` 函数, 作为对 `use_default_colors()` 的改进, 允许修改颜色对 0 的默认值。(由 Serhiy Storchaka 在 [gh-133139](#) 中贡献。)

7.10 datetime

- 新增 `datetime.time.strptime()` 和 `datetime.date.strptime()` 方法。(由 Wannes Boeykens 在 [gh-41431](#) 中贡献。)

7.11 decimal

- 新增 `Decimal` 替代性构造函数 `Decimal.from_number()`。(由 Serhiy Storchaka 在 [gh-121798](#) 中贡献。)
- 公开 `decimal.IEEEContext()` 函数以支持创建符合 IEEE 754 (2008) 十进制交换格式的上下文。(由 Sergey B Kirpichev 在 [gh-53032](#) 中贡献。)

7.12 difflib

- 由 `difflib.HtmlDiff` 类生成的高亮显示更改的比较页面现在支持暗色模式。(由李佳昊在 [gh-129939](#) 中贡献。)

7.13 dis

- 新增对 指令完整源码位置信息 (而非仅行号) 的渲染支持, 该特性通过 `show_positions` 关键字参数添加到以下接口:

```
- dis.Bytecode
- dis.dis()
- dis.distb()
- dis.disassemble()
```

该特性同时通过 `dis --show-positions` 命令行选项提供。(由 Bénédict Tran 在 [gh-123165](#) 中贡献。)

- 新增 `dis --specialized` 命令行选项, 用于显示特化字节码。(由 Bénédict Tran 在 [gh-127413](#) 中贡献。)

7.14 errno

- 新增 `errno.EHWPOISON` 错误码。(由 James Roy 在 [gh-126585](#) 中贡献。)

7.15 faulthandler

- 添加对在 支持此功能的系统上打印 C 栈回溯的支持, 可通过 `faulthandler.dump_c_stack()` 或 `faulthandler.enable()` 中的 `c_stack` 参数实现。(由 Peter Bierma 在 [gh-127604](#) 中贡献。)

7.16 fnmatch

- 新增 `fnmatch.filterfalse()` 函数, 用于排除符合模式匹配的名称。(由 Bénédict Tran 在 [gh-74598](#) 中贡献。)

7.17 fractions

- 新增对实现了 `as_integer_ratio()` 方法的任意对象转换为 `Fraction` 的支持。(由 Serhiy Storchaka 在 [gh-82017](#) 中贡献。)
- 新增 `Fraction` 替代性构造器 `Fraction.from_number()`。(由 Serhiy Storchaka 在 [gh-121797](#) 中贡献。)

7.18 functools

- 新增对 `functools.partial()` 和 `functools.partialmethod()` 支持 `functools.Placeholder` 哨兵的功能，用于为位置参数预留位置。(由 Dominykas Grigonis 在 [gh-119127](#) 中贡献。)
- 允许将 `functools.reduce()` 的 *initial* 形参作为关键字参数传递。(由 Sayandip Dutta 在 [gh-125916](#) 中贡献。)

7.19 getopt

- 新增对可选参数选项的支持。(由 Serhiy Storchaka 在 [gh-126374](#) 中贡献。)
- 新增对按顺序返回混合选项和非选项参数的支持。(由 Serhiy Storchaka 在 [gh-126390](#) 中贡献。)

7.20 getpass

- 新增 `getpass.getpass()` 通过仅限关键字可选参数 `echo_char` 提供键盘反馈的功能。每当输入字符时会显示占位符，删除字符时则会移除该占位符。(由 Semyon Moroz 在 [gh-77065](#) 中贡献。)

7.21 graphlib

- 允许在排序尚未开始的情况下多次调用 `graphlib.TopologicalSorter.prepare()`。(由 Daniel Pope 在 [gh-130914](#) 中贡献。)

7.22 heapq

- 添加用于处理最大堆的函数：
 - `heapq.heapify_max()`,
 - `heapq.heappush_max()`,
 - `heapq.heappop_max()`,
 - `heapq.heapreplace_max()`
 - `heapq.heappushpop_max()`

7.23 hmac

- 新增基于 **HACL*** 项目形式化验证代码的 **HMAC (RFC 2104)** 内置实现。(由 Bénédict Tran 在 [gh-99108](#) 中贡献。)

7.24 http

- 由 `http.server` 模块生成的目录列表和错误页面允许浏览器应用其默认暗模式。(由 Yorik Hansen 在 [gh-123430](#) 中贡献。)
- `http.server` 模块现在支持使用 `http.server.HTTPSServer` 类通过 **HTTPS** 提供服务。此功能通过命令行界面 (`python -m http.server`) 通过以下选项公开：
 - `--tls-cert <path>`: TLS 证书文件的路径。
 - `--tls-key <path>`: 私钥文件的可选路径。
 - `--tls-password-file <path>`: 私钥密码文件的可选路径。

(由 Semyon Moroz 在 [gh-85162](#) 中贡献。)

7.25 imaplib

- 添加 `IMAP4.idle()`，实现了 [RFC 2177](#) 中定义的 IMAP4 IDLE 命令。(由 Forest 在 [gh-55454](#) 中贡献。)

7.26 inspect

- `inspect.signature()` 新增了一个参数 `annotation_format`，用于控制表示注解所使用的 `annotationlib.Format`。(由 Jelle Zijlstra 在 [gh-101552](#) 中贡献。)
- `inspect.Signature.format()` 新增了一个参数 `unquote_annotations`。如果该参数为 `True`，字符串注解将不带引号显示。(由 Jelle Zijlstra 在 [gh-101552](#) 中贡献。)
- 新增 `inspect.ispackage()` 函数，用于判断对象是否为 `package`。(由 Zhikang Yan 在 [gh-125634](#) 中贡献。)

7.27 io

- 使用 `read` 从非阻塞流读取文本时，如果操作无法立即返回字节，现在可能会引发 `BlockingIOError`。(由 Giovanni Siragusa 在 [gh-109523](#) 中贡献。)
- 添加协议 `io.Reader` 和 `io.Writer`，作为伪协议 `typing.IO`、`typing.TextIO` 和 `typing.BinaryIO` 的更简单替代。(由 Sebastian Rittau 在 [gh-127648](#) 中贡献。)

7.28 json

- 添加关于 JSON 序列化错误的说明，以便识别错误来源。(由 Serhiy Storchaka 在 [gh-122163](#) 中贡献。)
- 使 `json` 模块能够使用 `-m` 开关作为脚本运行：`python -m json`。请参阅 JSON 命令行界面文档。(由 Trey Hunner 在 [gh-122873](#) 中贡献。)
- 默认情况下，JSON 命令行界面的输出会以彩色突出显示。这可以通过环境变量进行控制。(由 Tomas Roun 在 [gh-131952](#) 中贡献。)

7.29 linecache

- `linecache.getline()` 可以检索冻结模块的源代码。(由 Tian Gao 在 [gh-131638](#) 中贡献。)

7.30 logging.handlers

- `logging.handlers.QueueListener` 现在实现了上下文管理器协议，允许在 `with` 语句中使用它。(由 Charles Machalow 在 [gh-132106](#) 中贡献。)
- `QueueListener.start` 现在如果监听器已启动，会引发 `RuntimeError`。(由 Charles Machalow 在 [gh-132106](#) 中贡献。)

7.31 math

- 为模块中的域错误添加了更详细的错误消息。(由 Charlie Zhao 和 Sergey B Kirpichev 在 [gh-101410](#) 中贡献。)

7.32 mimetypes

- 完善 `mimetypes` 模块的命令行文档。现在，在失败时返回码由 0 改为 1，在命令行形参错误时返回码由 1 改为 2。此外，错误会输出到 `stderr` 而非 `stdout`，且错误文本更为简洁。(由 Oleg Iarygin 和 Hugo van Kemenade 在 [gh-93096](#) 中贡献。)
- 添加用于字体的 MS 和 [RFC 8081](#) MIME 类型：
 - 嵌入式 OpenType: `application/vnd.ms-fontobject`

- OpenType 布局 (OTF) `font/otf`
- TrueType: `font/ttf`
- WOFF 1.0 `font/woff`
- WOFF 2.0 `font/woff2`

(由 Sahil Prajapati 和 Hugo van Kemenade 在 [gh-84852](#) 中贡献。)

- 添加用于 Matroska 视听数据容器结构的 **RFC 9559** MIME 类型, 包括:

- 仅有音频, 无视频: `audio/matroska(.mka)`
- 视频: `video/matroska(.mkv)`
- 立体视频: `video/matroska-3d(.mk3d)`

(由 Hugo van Kemenade 在 [gh-89416](#) 中贡献。)

- 新增符合 RFC 标准的图像 MIME 类型:

- **RFC 1494**: CCITT Group 3 (`.g3`)
- **RFC 3362**: Real-time Facsimile, T.38 (`.t38`)
- **RFC 3745**: JPEG 2000 (`.jp2`)、扩展格式 (`.jpx`) 和复合格式 (`.jpm`)
- **RFC 3950**: 标签图像文件格式传真扩展, TIFF-FX (`.tfx`)
- **RFC 4047**: 灵活图像传输系统 (`.fits`)
- **RFC 7903**: 增强型图元文件 (`.emf`) 和 Windows 图元文件 (`.wmf`)

(由 Hugo van Kemenade 在 [gh-85957](#) 中贡献。)

- 更多 MIME 类型更改:

- **RFC 2361**: 将 `.avi` 的类型更改为 `video/vnd.avi`, 将 `.wav` 的类型更改为 `audio/vnd.wave`
- **RFC 4337**: 添加 MPEG-4 `audio/mp4(.m4a)`
- **RFC 5334**: 添加 Ogg 媒体 (`.oga`, `.ogg` 和 `.ogx`)
- **RFC 6713**: 添加 `application/gzip(.gz)`
- **RFC 9639**: 添加 FLAC 格式的 `audio/flac(.flac)`
- 添加 7z `application/x-7z-compressed(.7z)`
- 非严格模式下, 添加 Android 安装包 `application/vnd.android.package-archive(.apk)`
- 添加 deb `application/x-debian-package(.deb)`
- 添加 glTF 二进制格式 `model/gltf-binary(.glb)`
- 添加 glTF JSON/ASCII 格式 `model/gltf+json(.gltf)`
- 添加 M4V `video/x-m4v(.m4v)`
- 添加 PHP `application/x-httpd-php(.php)`
- 添加 RAR `application/vnd.rar(.rar)`
- 添加 RPM `application/x-rpm(.rpm)`
- 添加 STL 格式 `model/stl(.stl)`
- 添加 Windows 媒体视频 `video/x-ms-wmv(.wmv)`
- 事实上: 添加 WebM 格式 `audio/webm(.weba)`
- **ECMA-376**: 添加 `.docx`、`.pptx` 和 `.xlsx` 类型
- **OASIS**: 添加 OpenDocument 格式的 `.odg`、`.odp`、`.ods` 和 `.odt` 类型
- **W3C**: 添加 EPUB 格式的 `application/epub+zip(.epub)`

(由 Hugo van Kemenade 在 [gh-129965](#) 中贡献。)

- 添加 [RFC 9512](#) 中针对 YAML 文件 (.yaml 和 .yml) 的 application/yaml MIME 类型。(由 Sasha "Nelie" Chernykh 和 Hugo van Kemenade 在 [gh-132056](#) 中贡献。)

7.33 multiprocessing

- 默认的启动方法在 macOS 和 Windows 以外的平台上已从 fork 改为 forkserver，而在 macOS 和 Windows 平台上，该方法原本就是 spawn。

如果需要使用与线程不兼容的 *fork* 方法，则必须通过 `multiprocessing.get_context()` 提供的上下文显式请求该方法（推荐方式），或者通过 `multiprocessing.set_start_method()` 更改默认方法。

请参阅 `forkserver` 限制说明了解与 *fork* 方法的差异信息，以及此项变更对存在以下情况的现有代码可能产生的影响：(1) 使用可变全局共享变量 (2) 包含无法被 `pickle` 自动序列化的共享对象。

(由 Gregory P. Smith 在 [gh-84559](#) 中贡献)

- `multiprocessing` 的 "forkserver" 启动方法现增加了控制套接字认证机制，不再仅依赖文件系统权限来限制其他进程可能触发的衍生工作进程及运行代码。(由 Gregory P. Smith 在 [gh-97514](#) 中贡献。)
- 针对 *list* 和 *dict* 类型的多进程代理对象增加了此前被忽略的缺失方法：
 - `clear()` 和 `copy()` 用于 *list* 的代理
 - `fromkeys()`、`reversed(d)`、`d | {}`、`{}` | `d`、`d |= {'b': 2}` 用于 *dict* 的代理

(由 Roy Hyunjin Han 在 [gh-103134](#) 中贡献。)

- 通过 `SyncManager.set()` 增加了对共享 `set` 对象的支持。`multiprocessing.Manager()` 方法中的 `set()` 现已可用。(由 Mingyu Park 在 [gh-129949](#) 中贡献。)
- 新增 `multiprocessing.Process.interrupt()` 方法，该方法通过发送 `SIGINT` 信号来终止子进程。这使得 `finally` 子句能够为被终止的进程打印栈回溯信息。(由 Artem Pulkhin 在 [gh-131913](#) 中贡献。)

7.34 operator

- 新增了两个函数: `operator.is_none()` 和 `operator.is_not_none()`，其中 `operator.is_none(obj)` 等价于 `obj is None`，而 `operator.is_not_none(obj)` 等价于 `obj is not None`。(由 Raymond Hettinger 和 Nico Mexis 在 [gh-115808](#) 中贡献。)

7.35 os

- 新增 `os.reload_environ()` 函数，用于根据 `os.putenv()`、`os.unsetenv()` 所做的变更，或同一进程中 Python 外部所做的环境变更，来更新 `os.environ` 和 `os.environb`。(由 Victor Stinner 在 [gh-120057](#) 中贡献。)
- 向 `os` 模块中添加了 `SCHED_DEADLINE` 和 `SCHED_NORMAL` 常量。(由 James Roy 在 [gh-127688](#) 中贡献。)
- 新增 `os.readinto()` 函数，用于从文件描述符读取数据到缓冲区对象中。(由 Cody Maloney 在 [gh-129205](#) 中贡献。)

7.36 os.path

- `os.path.realpath()` 的 *strict* 形参新增支持 `os.path.ALLOW_MISSING` 值。当启用该选项时：除 `FileNotFoundError` 外的错误将被重新引发；返回路径允许不存在，但保证已解析所有符号链接。(由 Petr Viktorin 为修复 [CVE 2025-4517](#) 贡献。)

7.37 pathlib

- 为 `pathlib.Path` 新增递归复制/移动文件和目录的方法：
 - `copy()` 将一个文件或目录树复制到目标位置。
 - `copy_into()` 会将内容复制到目标目录中。
 - `move()` 将一个文件或目录树移动到目标位置。
 - `move_into()` 会将内容移动到目标目录中。(由 Barney Gale 在 [gh-73991](#) 中贡献。)
- 新增 `pathlib.Path.info` 属性, 该属性存储一个实现了 `pathlib.types.PathInfo` 协议 (同样为新增) 的对象。此对象支持查询文件类型, 并在内部缓存 `stat()` 的结果。通过 `iterdir()` 生成的路径对象会利用扫描父目录时收集到的文件类型信息进行初始化。(由 Barney Gale 在 [gh-125413](#) 中贡献。)

7.38 pdb

- 硬编码断点 (`breakpoint()` 和 `pdb.set_trace()`) 现在会重用最近一次调用 `set_trace()` 的 `Pdb` 实例, 而不是每次都创建一个新实例。因此, 所有实例特定的数据 (如 `display` 和 `commands`) 在硬编码断点之间都会被保留下来。(由 Tian Gao 在 [gh-121450](#) 中贡献。)
- 为 `pdb.Pdb` 新增一个 `mode` 参数。当 `pdb` 处于 `inline` 模式时, 禁用 `restart` 命令。(由 Tian Gao 在 [gh-123757](#) 中贡献。)
- 当用户尝试在 `inline` 模式下退出 `pdb` 时, 会显示一个确认提示。输入 `y`、`Y`、按 `<Enter>` 或 `EOF` 将确认退出并调用 `sys.exit()`, 而非引发 `bdb.BdbQuit`。(由 Tian Gao 在 [gh-124704](#) 中贡献。)
- 像 `breakpoint()` 或 `pdb.set_trace()` 这样的内联断点将始终在调用帧处暂停程序, 而忽略 `skip` 模式 (如果有的话)。(由 Tian Gao 在 [gh-130493](#) 中贡献。)
- 在 `pdb` 的多行输入中, 行首的 `<tab>` 键现在会填充 4 个空格的缩进, 而不是插入 `\t` 字符。(由 Tian Gao 在 [gh-130471](#) 中贡献。)
- `pdb` 的多行输入中引入了自动缩进功能。当检测到新的代码块时, 它会要么保持上一行的缩进, 要么插入 4 个空格的缩进。(由 Tian Gao 在 [gh-133350](#) 中贡献。)
- 新增 `$_asynctask` 以在适用情况下访问当前的 `asyncio` 任务。(由 Tian Gao 在 [gh-124367](#) 中贡献。)
- 新增 `pdb.set_trace_async()` 以支持调试 `asyncio` 协程。此函数支持 `await` 语句。(由 Tian Gao 在 [gh-132576](#) 中贡献。)
- 在 `pdb` 中显示的源代码将带有语法高亮。除了新添加的 `pdb.Pdb` 的 `colorize` 参数外, 还可以使用与 `PyREPL` 相同的方法来控制此特性。(由 Tian Gao 和 Łukasz Langa 在 [gh-133355](#) 中贡献。)

7.39 pickle

- 将 `pickle` 模块的默认协议版本设置为 5。更多详细信息, 请参见 `pickle protocols`。
- 添加了关于 `pickle` 序列化错误的说明, 这些说明有助于识别错误的来源。(由 Serhiy Storchaka 在 [gh-122213](#) 中贡献。)

7.40 platform

- 新增 `platform.invalidate_caches()` 函数, 用于使缓存结果失效。(由 Bénédict Tran 在 [gh-122549](#) 中贡献。)

7.41 pydoc

- 帮助输出中的 注解现在通常以更接近原始源代码中的格式显示。(由 Jelle Zijlstra 在 [gh-101552](#) 中贡献。)

7.42 socket

- 改进并修复对蓝牙套接字的支持。
 - 修复了 NetBSD 和 DragonFly BSD 系统上对蓝牙套接字的支持。(由 Serhiy Storchaka 在 [gh-132429](#) 中贡献。)
 - 修复了 FreeBSD 系统上对 BTPROTO_HCI 的支持。(由 Victor Stinner 在 [gh-111178](#) 中贡献。)
 - 新增对 FreeBSD 系统上 BTPROTO_SCO 的支持。(由 Serhiy Storchaka 在 [gh-85302](#) 中贡献。)
 - 新增对 FreeBSD 系统上 BTPROTO_L2CAP 地址中 *cid* 和 *bdaddr_type* 的支持。(由 Serhiy Storchaka 在 [gh-132429](#) 中贡献。)
 - 新增对 Linux 系统上 BTPROTO_HCI 地址中 *channel* 的支持。(由 Serhiy Storchaka 在 [gh-70145](#) 中贡献。)
 - 在 Linux 系统上, 允许将整数作为 BTPROTO_HCI 的地址。(由 Serhiy Storchaka 在 [gh-132099](#) 中贡献。)
 - 在 BTPROTO_L2CAP 中, `getsockname()` 会返回 *cid*。(由 Serhiy Storchaka 在 [gh-132429](#) 中贡献。)
 - 新增了许多常量。(由 Serhiy Storchaka 在 [gh-132734](#) 中贡献。)

7.43 ssl

- 通过 `ssl.HAS_PHA` 指示 `ssl` 模块是否支持 TLSv1.3 握手后客户端认证 (PHA)。(由 Will Childs-Klein 在 [gh-128036](#) 中贡献。)

7.44 struct

- 在 `struct` 模块中支持 `float complex` 和 `double complex` 这两种 C 类型 (分别对应格式字符 'F' 和 'D')。(由 Sergey B Kirpichev 在 [gh-121249](#) 中贡献。)

7.45 symtable

- 公开以下 `symtable.Symbol` 方法:
 - `is_comp_cell()`
 - `is_comp_iter()`
 - `is_free_class()`(由 Bénédict Tran 在 [gh-120029](#) 中贡献。)

7.46 sys

- 之前未写入文档的特殊函数 `sys.getobjects()`, 它仅存在于某些专用的 Python 构建版, 现在可以从其他解释器而非调用它的解释器返回对象。
- 新增 `sys._is_immortal()` 函数, 用于判断一个对象是否为 `immortal`。(由 Peter Bierma 在 [gh-128509](#) 中贡献。)
- 在 FreeBSD 上, `sys.platform` 将不再包含主版本号。它将始终为 'freebsd', 而不是 'freebsd13' 或 'freebsd14'。
- 为 `sys._clear_type_cache()` 函数引发 `DeprecationWarning` 警告。该函数在 Python 3.13 中已被弃用, 但此前并未引发运行时警告。

7.47 sys.monitoring

- 新增了两个事件: `BRANCH_LEFT` 和 `BRANCH_RIGHT`。`BRANCH` 事件已被弃用。

7.48 sysconfig

- 在 Windows 系统上，为 `sysconfig.get_config_vars()` 新增 `ABIFLAGS` 键。（由 Xuehai Pan 在 [gh-131799](#) 中贡献。）

7.49 tarfile

- `data_filter()` 现在会对符号链接目标进行规范化处理，以避免路径遍历攻击。（由 Petr Viktorin 在 [gh-127987](#) 和 [CVE 2025-4138](#) 中贡献。）
- `extractall()` 现在在目录被移除或被其他类型的文件替换时，会跳过对目录属性的修复。（由 Petr Viktorin 在 [gh-127987](#) 和 [CVE 2024-12718](#) 中贡献。）
- `extract()` 和 `extractall()` 现在在以下两种情况会（重新）应用提取过滤器：一是用另一个归档成员的副本替换链接（硬链接或符号链接）时，二是修复目录属性时。前者会引发一个新异常 `LinkFallbackError`。（由 Petr Viktorin 针对 [CVE 2025-4330](#) 和 [CVE 2024-12718](#) 贡献。）
- `extract()` 和 `extractall()` 在 `errorlevel()` 为 0 时，不再提取被拒绝的成员。（由 Matt Prodan 和 Petr Viktorin 在 [gh-112887](#) 以及 [CVE 2025-4435](#) 中贡献。）

7.50 threading

- `threading.Thread.start()` 现在会将操作系统线程名称设置为 `threading.Thread.name`。（由 Victor Stinner 在 [gh-59705](#) 中贡献。）

7.51 tkinter

- 使 tkinter 控件的 `after()` 和 `after_idle()` 方法支持通过关键字传递参数。（由 Zhikang Yan 在 [gh-126899](#) 中贡献。）
- 新增为 `tkinter.OptionMenu` 和 `tkinter.ttk.OptionMenu` 指定名称的功能。（由 Zhikang Yan 在 [gh-130482](#) 中贡献。）

7.52 turtle（海龟绘图）

- 为 `turtle.fill()`、`turtle.poly()` 和 `turtle.no_animation()` 添加了上下文管理器。（由 Marie Roald 和 Yngve Mardal Moe 在 [gh-126350](#) 中贡献。）

7.53 types（类型）

- `types.UnionType` 现在是 `typing.Union` 的别名。更多详情请参见下方。（由 Jelle Zijlstra 在 [gh-105499](#) 中贡献。）

7.54 typing

- `types.UnionType` 和 `typing.Union` 现在互为别名，这意味着旧式联合（通过 `Union[int, str]` 创建）和新式联合（`int | str`）现在都会创建相同运行时类型的实例。这统一了两种语法的行为，但也带来了一些行为差异，可能会影响那些在运行时对类型进行内省的用户：
 - 创建联合的两种语法现在在 `repr()` 中会生成相同的字符串表示形式。例如，`repr(Union[int, str])` 现在会返回 `"int | str"`，而非 `"typing.Union[int, str]"`。
 - 使用旧式语法创建的联合不再被缓存。此前，多次运行 `Union[int, str]` 会返回同一个对象（`Union[int, str] is Union[int, str]` 的结果为 `True`），但现在会返回两个不同的对象。用户应当使用 `==` 来比较联合是否相等，而非 `is`。新式联合从未以这种方式被缓存。这一变化可能会增加某些程序的内存占用，这些程序会通过下标 `typing.Union` 创建大量联合。不过，有几个因素会抵消这一开销：由于 [PEP 649](#) 的引入，Python 3.14 中默认不再对注解中使用的联合进行求值；`types.UnionType` 的实例本身比早期 Python 版本中 `Union[]` 返回的对象小得多；此外，移除缓存也节省了部分空间。因此，对于大多数用户而言，这一变化不太可能导致内存占用显著增加。

- 此前，旧式联合是通过私有类 `typing._UnionGenericAlias` 实现的。该类在当前实现中已不再需要，但为了向后兼容性而被保留，计划在 **Python 3.17** 中移除。用户应使用有文档记录的内省辅助函数，如 `typing.get_origin()` 和 `typing.get_args()`，而非依赖私有实现细节。
- 现在可以在 `isinstance()` 检查中使用 `typing.Union` 本身。例如，`isinstance(int | str, typing.Union)` 将返回 `True`；而此前这会引发 `TypeError`。
- `typing.Union` 对象的 `__args__` 属性不再可写。
- `typing.Union` 对象现在禁止设置任何属性。此前版本中仅能设置双下划线属性，且该行为既无文档支持，实际使用中也存在诸多潜在问题。

(由 Jelle Zijlstra 在 [gh-105499](#) 中贡献。)

- `typing.TypeAliasType` 现在支持星号解包操作。

7.55 unicodedata

- Unicode 数据库已更新到 16.0.0 版本。

7.56 unittest

- `unittest` 模块的输出现在默认启用彩色显示，可通过 环境变量 进行控制。(由 Hugo van Kemenade 在 [gh-127221](#) 中贡献。)
- `unittest` 发现机制重新支持将 `namespace package` 作为起始目录，该功能曾在 **Python 3.11** 中被移除。(由 Jacob Walls 在 [gh-80958](#) 中贡献。)
- 在 `TestCase` 类中新增了多个方法，这些方法可提供更专门化的测试。
 - `assertHasAttr()` 和 `assertNotHasAttr()` **check whether the object has a particular attribute.**
 - `assertIsSubclass()` 和 `assertNotIsSubclass()` 用于检查对象是否是某个特定类的子类，或者是否是某个类元组中任一类的子类。
 - `assertStartsWith()`、`assertNotStartsWith()`、`assertEndsWith()` 和 `assertNotEndsWith()` 用于检查 **Unicode 字符串或字节串** 是否以特定字符串开头或结尾。

(由 Serhiy Storchaka 在 [gh-71339](#) 中贡献。)

7.57 urllib

- 升级 `urllib.request` 的 HTTP 摘要认证算法，支持 **RFC 7616** 中规定的 SHA-256 摘要认证。(由 Calvin Bui 在 [gh-128193](#) 中贡献。)
- 改进解析和生成 `file:` URL 时的易用性和标准合规性。

在 `urllib.request.url2pathname()` 中：

- 当新的 `require_scheme` 参数设为 `true` 时，接受完整的 URL。
- 如果 URL 的权限部分与本地主机名匹配，则舍弃该部分。
- 当新的 `resolve_host` 参数设为 `true` 时，若 URL 的权限部分解析为本地 IP 地址，则舍弃该部分。
- 丢弃 URL 查询和片段组件。
- 如果 URL 的权限部分不是本地的，则引发 `URLError`，但在 **Windows** 系统上，仍会像以前一样返回 UNC 路径。

在 `urllib.request.pathname2url()` 中：

- 当新的 `add_scheme` 参数设为 `true` 时，返回完整的 URL。
- 当路径以斜杠开头时，包含一个空的 URL 权限部分。例如，路径 `/etc/hosts` 会被转换为 `URL:///etc/hosts`。

在 Windows 系统上，盘符不再转换为大写，且非紧跟在驱动器号后的 `:` 字符不再引发 `OSError` 异常。

(由 Barney Gale 在 [gh-125866](#) 中贡献。)

7.58 uuid

- 通过 `uuid.uuid6()`、`uuid.uuid7()` 和 `uuid.uuid8()` 分别添加对 UUID 第 6、7、8 版的支持，具体规范见 [RFC 9562](#)。(由 Bénédict Tran 在 [gh-89083](#) 中贡献。)
- `uuid.NIL` 和 `uuid.MAX` 现已可用，分别用于表示 [RFC 9562](#) 中定义的空 UUID (Nil UUID) 和最大 UUID (Max UUID) 格式。(由 Nick Pope 在 [gh-128427](#) 中贡献。)
- 允许通过 `python -m uuid --count` 一次性生成多个 UUID。(由 Simon Legner 在 [gh-131236](#) 中贡献。)

7.59 webbrowser

- `BROWSER` 环境变量中的名称现在可以引用 `webbrowser` 模块中已注册的浏览器，而不必总是生成新的浏览器命令。

这使得可以将 `BROWSER` 设置为 macOS 上受支持的浏览器之一的值。

7.60 zipfile

- 新增了 `ZipInfo._for_archive` 方法，用于为 `ZipInfo` 对象解析合适的默认值，该对象由 `ZipFile.writestr` 方法使用。(由 Bénédict Tran 在 [gh-123424](#) 中贡献。)
- `zipfile.ZipFile.writestr()` 现在会遵从 `SOURCE_DATE_EPOCH` 环境变量。该变量可由发行版统一设置，以便构建工具通过读取该变量来生成可复现的输出。(由李佳昊在 [gh-91279](#) 中贡献。)

8 性能优化

- 几个标准库模块的导入时间已得到改进，包括 `annotationlib`、`ast`、`asyncio`、`base64`、`cmd`、`csv`、`gettext`、`importlib.util`、`locale`、`mimetypes`、`optparse`、`pickle`、`pprint`、`pstats`、`shlex`、`socket`、`string`、`subprocess`、`threading`、`tomllib`、`types` 和 `zipfile`。

(由 Adam Turner、Bénédict Tran、Chris Markiewicz、Eli Schwartz、Hugo van Kemenade、Jelle Zijlstra 等人在 [gh-118761](#) 中贡献。)

8.1 asyncio

- 标准基准测试结果提升了 10-20%，这是在为 `native tasks` 实现新的每线程双链表后取得的，同时也减少了内存使用。这使得诸如 `python -m asyncio pstree` 等外部内省工具能够内省在所有线程中运行的 `asyncio` 任务的调用图。(由 Kumar Aditya 在 [gh-107803](#) 中贡献。)
- 该模块现在对自由线程构建提供了一流的支持。这使得多个事件循环可以在不同线程中并行执行，并且随着线程数量的增加线性扩展。(由 Kumar Aditya 在 [gh-128002](#) 中贡献。)

8.2 base64

- `b16decode()` 的速度现在提升了高达六倍。(由 Bénédict Tran、Chris Markiewicz 和 Adam Turner 在 [gh-118761](#) 中贡献。)

8.3 bdb

- 基本调试器现在有了基于 `sys.monitoring` 的后端，可以通过将 `'monitoring'` 传递给 `Bdb` 类的新 `backend` 参数来选择。(由 Tian Gao 在 [gh-124533](#) 中贡献。)

8.4 difflib

- `IS_LINE_JUNK()` 函数的速度现在提升了一倍。(由 Adam Turner 和 Semyon Moroz 在 [gh-130167](#) 中贡献。)

8.5 gc

- 新的增量垃圾回收器意味着对于较大的堆，最大暂停时间减少了至少一个数量级。

由于这种优化，`get_threshold()` 和 `set_threshold()` 的结果含义发生了变化，同时还有 `get_count()` 和 `get_stats()`。

- 为了向后兼容，`get_threshold()` 继续返回一个三项元组。第一个值是年轻代回收的阈值，与之前相同；第二个值决定了老年代回收的扫描速率（默认值为 10，值越高意味着老年代回收扫描越慢）。第三个值现在没有意义，总是为零。
- `set_threshold()` 现在会忽略第二个之后的任何项。
- `get_count()` 和 `get_stats()` 方法仍返回相同格式的结果。唯一的区别在于，结果不再分别对应新生代、老化代和老年代，而是对应新生代以及老年代的待老化空间和回收空间。

总之，尝试操控循环垃圾回收器行为的代码可能无法完全按预期工作，但几乎不会造成危害。其他所有代码都将正常运行。

(由 Mark Shannon 在 [gh-108362](#) 中贡献。)

8.6 io

- 打开和读取文件现在执行更少的系统调用。完整读取一个小型操作系统缓存文件的速度提高了最多 15%。(由 Cody Maloney 和 Victor Stinner 在 [gh-120754](#) 和 [gh-90102](#) 中贡献。)

8.7 pathlib

- `Path.read_bytes` 现在使用无缓冲模式打开文件，完整读取的速度提高了 9% 到 17%。(由 Cody Maloney 在 [gh-120754](#) 中贡献。)

8.8 pdb

- `pdb` 现在支持两种后端，基于 `sys.settrace()` 或 `sys.monitoring`。使用 `pdb CLI` 或 `breakpoint()` 将始终使用 `sys.monitoring` 后端。显式实例化 `pdb.Pdb` 及其派生类将默认使用 `sys.settrace()` 后端，这是可配置的。(由 Tian Gao 在 [gh-124533](#) 中贡献。)

8.9 uuid

- `uuid3()` 和 `uuid5()` 对于 16 字节名称的速度现在大约提高了 40%，对于 1024 字节名称的速度提高了 20%。更长名称的性能保持不变。(由 Bénédict Tran 在 [gh-128150](#) 中贡献。)
- `uuid4()` 的速度现在提高了约 30%。(由 Bénédict Tran 在 [gh-128150](#) 中贡献。)

8.10 zlib

- 在 Windows 上，`zlib-ng` 现在用作默认二进制文件中 `zlib` 模块的实现。已知 `zlib-ng` 与之前使用的 `zlib` 实现之间没有不兼容性。这应在所有压缩级别上带来更好的性能。

值得注意的是，`zlib.Z_BEST_SPEED` (1) 可能会导致比之前的实现显著更低的压缩率，同时显著减少压缩所需的时间。

(由 Steve Dower 在 [gh-91349](#) 中贡献。)

9 移除

9.1 argparse

- 移除 `BooleanOptionalAction` 的 *type*、*choices* 和 *metavar* 形参。它们自 Python 3.12 起已被弃用。(由 Nikita Sobolev 在 [gh-118805](#) 中贡献。)
- 在参数组上调用 `add_argument_group()` 现在会引发 `ValueError`。类似地，在互斥组上调用 `add_argument_group()` 或 `add_mutually_exclusive_group()` 现在都会引发 `ValueError`。这种“嵌套调用”从未被支持，经常无法正常工作，且通过继承关系被意外暴露。该功能自 Python 3.11 起已被弃用。(由 Savannah Ostrowski 在 [gh-127186](#) 中贡献。)

9.2 ast (抽象语法树)

- 移除以下自 Python 3.8 起作为 `Constant` 的已弃用别名，且自 Python 3.12 起已发出弃用警告的类：

- `Bytes`
- `Ellipsis`
- `NameConstant`
- `Num`
- `Str`

由于这些移除操作，当自定义的 `NodeVisitor` 子类访问抽象语法树 (AST) 时，用户定义的 `visit_Num`、`visit_Str`、`visit_Bytes`、`visit_NameConstant` 和 `visit_Ellipsis` 方法将不再被调用。请改用定义 `visit_Constant` 方法。

(由 Alex Waygood 在 [gh-119562](#) 中贡献。)

- 移除以下 `ast.Constant` 为兼容现已移除的 AST 类而保留的已弃用属性：

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

改用 `Constant.value`。(由 Alex Waygood 在 [gh-119562](#) 中贡献。)

9.3 asyncio

- 移除以下自 Python 3.12 起已被弃用的类、方法和函数：

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

(由 Kumar Aditya 在 [gh-120804](#) 中贡献。)

- `asyncio.get_event_loop()` 现在如果没有当前事件循环，会引发 `RuntimeError` 异常，且不再隐式创建事件循环。

(由 Kumar Aditya 在 [gh-126353](#) 中贡献。)

当前存在几种使用 `asyncio.get_event_loop()` 的模式，其中大多数可替换为 `asyncio.run()`。如果正在运行异步函数，直接使用 `asyncio.run()` 即可。

之前：

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

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

之后：

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

asyncio.run(main())
```

如果需要启动某些持续运行的服务，例如监听套接字的服务器，请使用 `asyncio.run()` 配合 `asyncio.Event` 实现。

之前：

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

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

之后：

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

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

asyncio.run(main())
```

如果需要在事件循环中运行某些任务，同时在其前后执行阻塞代码，请使用 `asyncio.Runner`。

之前：

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

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

之后：

```

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 email

- 移除 `email.utils.localtime()` 函数的 `isdst` 形参。该参数自 Python 3.12 起已被弃用且一直被忽略。(由 Hugo van Kemenade 在 [gh-118798](#) 中贡献。)

9.5 importlib.abc

- 移除已弃用的 `importlib.abc` 类：
 - `ResourceReader` (使用 `TraversableResources`)
 - `Traversable` (使用 `Traversable`)
 - `TraversableResources` (使用 `TraversableResources`)(由 Jason R. Coombs 和 Hugo van Kemenade 在 [gh-93963](#) 中贡献。)

9.6 itertools

- 移除 `itertools` 迭代器对复制 (`copy`)、深度复制 (`deepcopy`) 和序列化 (`pickle`) 操作的支持。自 Python 3.12 起, 这些操作已触发 `DeprecationWarning` 警告。(由 Raymond Hettinger 在 [gh-101588](#) 中贡献。)

9.7 pathlib

- 移除向 `Path` 传递额外关键字参数的支持。在先前版本中，此类参数均会被忽略。(由 Barney Gale 在 [gh-74033](#) 中贡献。)
- 移除向 `PurePath.relative_to()` 和 `is_relative_to()` 方法传递额外位置参数的支持。在先前版本中，此类参数会被拼接到 `other` 参数上。(由 Barney Gale 在 [gh-78707](#) 中贡献。)

9.8 pkgutil

- 移除 `get_loader()` 和 `find_loader()` 函数，这两个函数自 Python 3.12 起已被弃用。（由 [Bénédikt Tran](#) 在 [gh-97850](#) 中贡献。）

9.9 pty

- 移除 `master_open()` 和 `slave_open()` 函数，这两个函数自 Python 3.12 起已被弃用。请改用 `pty.openpty()` 函数。

9.10 sqlite3

- 从 `sqlite3` 模块中移除 `version` 和 `version_info`；请使用 `sqlite_version` 和 `sqlite_version_info` 来获取运行时 SQLite 库的实际版本号。（由 Hugo van Kemenade 在 [gh-118924](#) 中贡献。）
- 现在，使用带具名占位符的形参序列会引发 `ProgrammingError` 异常，该用法自 Python 3.12 起已被弃用。（由 Erlend E. Aasland 在 [gh-118928](#) 和 [gh-101693](#) 中贡献。）

9.11 urllib

- 从 `urllib.parse` 模块中移除 `Quoter` 类，该类自 Python 3.11 起已被弃用。（由 Nikita Sobolev 在 [gh-118827](#) 中贡献。）
- 从 `urllib.request` 模块中移除 `URLOpener` 和 `FancyURLOpener` 类，这两个类自 Python 3.3 起已被弃用。

`myopener.open()` 可以替换为 `urlopen()`。`myopener.retrieve()` 可以替换为 `urlretrieve()`。对 `opener` 类的自定义操作可以通过向 `build_opener()` 传递定制化的处理程序来实现替换。（由 Barney Gale 在 [gh-84850](#) 中贡献。）

10 弃用

10.1 新的弃用

- 传入一个复数作为 `complex()` 构造器中的 *real* 或 *imag* 参数的做法现已被弃用；复数应当仅作为单个位置参数被传入。（由 Serhiy Storchaka 在 [gh-109218](#) 中贡献。）
- `argparse`:
 - 将未写入文档的关键字参数 *prefix_chars* 传给 `add_argument_group()` 方法的做法现已被弃用。（由 Savannah Ostrowski 在 [gh-125563](#) 中贡献。）
 - 已弃用 `argparse.FileType` 类型转换器。任何涉及资源管理的操作都应在参数解析完成之后在下游进行处理。（由 Serhiy Storchaka 在 [gh-58032](#) 中贡献。）

- `asyncio`:
 - The `asyncio.iscoroutinefunction()` is now deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in [gh-122875](#).)
 - The `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
 - * `asyncio.AbstractEventLoopPolicy`
 - * `asyncio.DefaultEventLoopPolicy`
 - * `asyncio.WindowsSelectorEventLoopPolicy`
 - * `asyncio.WindowsProactorEventLoopPolicy`
 - * `asyncio.get_event_loop_policy()`
 - * `asyncio.set_event_loop_policy()`

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

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

```
import asyncio

async def main():
    ...

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

（由 Kumar Aditya 在 [gh-127949](#) 中贡献。）

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

- On non-Windows platforms, setting `Structure._pack_` to use a MSVC-compatible default memory layout is now deprecated in favor of setting `Structure._layout_` to 'ms', and will be removed in Python 3.19. (Contributed by Petr Viktorin in [gh-131747](#).)
- Calling `ctypes.POINTER()` on a string is now deprecated. Use incomplete types for self-referential structures. Also, the internal `ctypes._pointer_type_cache` is deprecated. See `ctypes.POINTER()` for updated implementation details. (Contributed by Sergey Myrianov in [gh-100926](#).)
- `functools`: Calling the Python implementation of `functools.reduce()` with *function* or *sequence* as keyword arguments is now deprecated; the parameters will be made positional-only in Python 3.16. (Contributed by Kirill Podoprigora in [gh-121676](#).)
- `logging`: Support for custom logging handlers with the *strm* argument is now deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in [gh-115032](#).)
- `mimetypes`: Valid extensions are either empty or must start with '.' for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in [gh-75223](#).)
- `nturl2path`: 该模块现已弃用, 请改用 `urllib.request.url2pathname()` 和 `pathname2url()`。(由 Barney Gale 在 [gh-125866](#) 中贡献。)
- `os`: The `os.popen()` and `os.spawn*` functions are now soft deprecated. They should no longer be used to write new code. The `subprocess` module is recommended instead. (Contributed by Victor Stinner in [gh-120743](#).)
- `pathlib`: `pathlib.PurePath.as_uri()` is now deprecated and scheduled for removal in Python 3.19. Use `pathlib.Path.as_uri()` instead. (Contributed by Barney Gale in [gh-123599](#).)
- `pdb`: The undocumented `pdb.Pdb.curframe_locals` attribute is now a deprecated read-only property, which will be removed in a future version of Python. The low overhead dynamic frame locals access added in Python 3.13 by [PEP 667](#) means the frame locals cache reference previously stored in this attribute is no longer needed. Derived debuggers should access `pdb.Pdb.curframe.f_locals` directly in Python 3.13 and later versions. (Contributed by Tian Gao in [gh-124369](#) and [gh-125951](#).)
- `symtable`: Deprecate `symtable.Class.get_methods()` due to the lack of interest, scheduled for removal in Python 3.16. (Contributed by B               in [gh-119698](#).)
- `tkinter`: `tkinter.Variable` 的方法 `trace_variable()`、`trace_vdelete()` 和 `trace_vinfo()` 现已被弃用。请改用 `trace_add()`、`trace_remove()` 和 `trace_info()`。(由 Serhiy Storchaka 在 [gh-120220](#) 中贡献。)
- `urllib.parse`: Accepting objects with false values (like 0 and []) except empty strings, bytes-like objects and None in `parse_qs1()` and `parse_qs()` is now deprecated. (Contributed by Serhiy Storchaka in [gh-116897](#).)

10.2 计划在 Python 3.15 中移除

- 导入系统:
 - 当设置 `__spec__.cached` 失败时在模块上设置 `__cached__` 的做法已被弃用。在 Python 3.15 中, `__cached__` 将不会再被导入系统或标准库纳入考虑。([gh-97879](#))
 - 当设置 `__spec__.parent` 失败时在模块上设置 `__package__` 的做法已被弃用。在 Python 3.15 中, `__package__` 将不会再被导入系统或标准库纳入考虑。([gh-97879](#))
- `ctypes`:
 - 未写入文档的 `ctypes.SetPointerType()` 函数自 Python 3.13 起已被弃用。
- `http.server`:
 - 过时且很少被使用的 `CGIHTTPRequestHandler` 自 Python 3.13 起已被弃用。不存在直接的替代品。对于建立带有请求处理器的 Web 服务程序来说 任何东西都比 CGI 要好。
 - 用于 `python -m http.server` 命令行界面的 `--cgi` 旗标自 Python 3.13 起已被弃用。

- `importlib`:
 - `load_module()` 方法: 改用 `exec_module()`。
- `locale`:
 - `getdefaultlocale()` 函数自 Python 3.11 起已被弃用。最初计划在 Python 3.13 中移除它 ([gh-90817](#)), 但已被推迟至 Python 3.15。请改用 `getlocale()`, `setlocale()` 和 `getencoding()`。(由 Hugo van Kemenade 在 [gh-111187](#) 中贡献。)
- `pathlib`:
 - `PurePath.is_reserved()` 自 Python 3.13 起已被弃用。请使用 `os.path.isreserved()` 来检测 Windows 上的保留路径。
- `platform`:
 - `java_ver()` 自 Python 3.13 起已被弃用。此函数仅对 Jython 支持有用, 具有令人困惑的 API, 并且大部分未经测试。
- `sysconfig`:
 - `sysconfig.is_python_build()` 的 `check_home` 参数自 Python 3.12 起已被弃用。
- `threading`:
 - 在 Python 3.15 中 `RLock()` 将不再接受参数。传入参数的做法自 Python 3.14 起已被弃用, 因为 Python 版本不接受任何参数, 而 C 版本允许任意数量的位置或关键字参数, 但会忽略所有参数。
- `types`:
 - `types.CodeType`: 访问 `co_lnotab` 的做法自 3.10 起已根据 [PEP 626](#) 被弃用并曾计划在 3.12 中移除, 但在 3.12 中实际仅设置了 `DeprecationWarning`。可能会在 3.15 中移除。(由 Nikita Sobolev 在 [gh-101866](#) 中贡献。)
- `typing`:
 - 未写入文档的用于创建 `NamedTuple` 类的关键字参数语法 (例如 `Point = NamedTuple("Point", x=int, y=int)`) 自 Python 3.13 起已被弃用。请改用基于类的语法或函数语法。
 - 当使用 `TypedDict` 的函数式语法时, 不向 `fields` 形参传递值 (`TD = TypedDict("TD")`) 或传递 `None` (`TD = TypedDict("TD", None)`) 的做法自 Python 3.13 起已被弃用。请改用 `class TD(TypedDict): pass` 或 `TD = TypedDict("TD", {})` 来创建一个零字段的 `TypedDict`。
 - `typing.no_type_check_decorator()` 装饰器自 Python 3.13 起已被弃用。存在于 `typing` 模块八年之后, 它仍未被任何主要类型检查器所支持。
- `wave`:
 - `Wave_read` 和 `Wave_write` 类的 `getmark()`, `setmark()` 和 `getmarkers()` 方法自 Python 3.13 起已被弃用。
- `zipimport`:
 - `load_module()` 自 Python 3.10 起已被弃用。请改用 `exec_module()`。(由李佳昊在 [gh-125746](#) 中贡献。)

10.3 计划在 Python 3.16 中移除

- 导入系统:
 - 当设置 `__spec__.loader` 失败时在模块上设置 `__loader__` 的做法已被弃用。在 Python 3.16 中, `__loader__` 将不会再被设置或是被导入系统或标准库纳入考虑。
- `array`:
 - `'u'` 格式代码 (`wchar_t`) 自 Python 3.3 起已在文档中弃用并自 Python 3.13 起在运行时弃用。对于 Unicode 字符请改用 `'w'` 格式代码 (`Py_UCS4`)。

- `asyncio`:

- `asyncio.iscoroutinefunction()` 已被弃用并将在 Python 3.16 中移除，请改用 `inspect.iscoroutinefunction()`。(由李佳昊和 Kumar Aditya 在 [gh-122875](#) 中贡献。)
- `asyncio` 策略系统已被弃用并将在 Python 3.16 中移除。具体而言，是弃用了下列类和函数：

- * `asyncio.AbstractEventLoopPolicy`
 - * `asyncio.DefaultEventLoopPolicy`
 - * `asyncio.WindowsSelectorEventLoopPolicy`
 - * `asyncio.WindowsProactorEventLoopPolicy`
 - * `asyncio.get_event_loop_policy()`
 - * `asyncio.set_event_loop_policy()`

用户应当使用 `asyncio.run()` 或 `asyncio.Runner` 并附带 `loop_factory` 以使用想要的事件循环实现。

例如，在 Windows 上使用 `asyncio.SelectorEventLoop`:

```
import asyncio

async def main():
    ...

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

(由 Kumar Aditya 在 [gh-127949](#) 中贡献。)

- `builtins`:

- 对布尔类型 `~True` 或 `~False` 执行按位取反的操作自 Python 3.12 起已被弃用，因为它会产生奇怪和不直观的结果 (`-2` and `-1`)。请改用 `not x` 来对布尔值执行逻辑否操作。对于需要对下层整数执行按位取反操作的少数场合，请显式地将其转换为 `int(~int(x))`。

- `functools`:

- 调用 `functools.reduce()` 的 Python 实现并传入 *function* 或 *sequence* 作为关键字参数的做法自 Python 3.14 起已被弃用。

- `logging`:

使用 *strm* 参数对自定义日志记录处理器提供支持的做法已被弃用并计划在 Python 3.16 中移除。改为使用 *stream* 参数定义处理器。(由 Mariusz Felisiak 在 [gh-115032](#) 中贡献。)

- `mimetypes`:

- 有效扩展以“.”开头或在 `mimetypes.MimeTypes.add_type()` 为空。未加点的扩展已弃用，在 Python 3.16 中将引发 `ValueError`。(由 Hugo van Kemenade 在 [gh-75223](#) 中贡献。)

- `shutil`:

- `ExecError` 异常自 Python 3.14 起已被弃用。它自 Python 3.4 起就未被 `shutil` 中的任何函数所使用，现在是 `RuntimeError` 的一个别名。

- `symtable`:

- `Class.get_methods` 方法自 Python 3.14 起被弃用。

- `sys`:

- `_enablelegacywindowsfsencoding()` 函数自 Python 3.13 起被弃用。请改用 `PYTHONLEGACYWINDOWSFSENCODING` 环境变量。

- `sysconfig`:

- 自 Python 3.14 起，`sysconfig.expand_makefile_vars()` 函数已被弃用。请使用 `sysconfig.get_paths()` 的 `vars` 参数代替。

- `tarfile`:
 - 未写入文档也未被使用的 `TarFile.tarfile` 属性自 Python 3.13 起被弃用。

10.4 计划在 Python 3.17 中移除

- `collections.abc`:
 - `collections.abc.ByteString` 计划在 Python 3.17 中移除。

使用 `isinstance(obj, collections.abc.Buffer)` 来测试 `obj` 是否在运行时实现了缓冲区协议。要用于类型标注，则使用 `Buffer` 或是显式指明你的代码所支持类型的并集 (例如 `bytes | bytearray | memoryview`)。

`ByteString` 原本是想作为 `bytes` 和 `bytearray` 的超类型的抽象基类提供。不过，由于 ABC 不能有任何方法，知道一个对象是 `ByteString` 的实例并不能真正告诉你有关该对象的任何有用信息。其他常见缓冲区类型如 `memoryview` 同样不能被当作是 `ByteString` 的子类型 (无论是在运行时还是对于静态类型检查器)。

请参阅 [PEP 688](#) 了解详情。(由 Shantanu Jain 在 [gh-91896](#) 中贡献。)
- `typing`:
 - 在 Python 3.14 之前，旧式的联合是通过私有类 `typing._UnionGenericAlias` 实现的。实现已不再需要该类，但为向后兼容性保留了该类，并计划在 Python 3.17 中删除。用户应使用记录在案的自省助手函数，如 `typing.get_origin()` 和 `typing.get_args()`，而不是依赖于私有的实现细节。
 - `typing.ByteString` 自 Python 3.9 起已被弃用，计划在 Python 3.17 中移除。

使用 `isinstance(obj, collections.abc.Buffer)` 来测试 `obj` 是否在运行时实现了缓冲区协议。要用于类型标注，则使用 `Buffer` 或是显式指明你的代码所支持类型的并集 (例如 `bytes | bytearray | memoryview`)。

`ByteString` 原本是想作为 `bytes` 和 `bytearray` 的超类型的抽象基类提供。不过，由于 ABC 不能有任何方法，知道一个对象是 `ByteString` 的实例并不能真正告诉你有关该对象的任何有用信息。其他常见缓冲区类型如 `memoryview` 同样不能被当作是 `ByteString` 的子类型 (无论是在运行时还是对于静态类型检查器)。

请参阅 [PEP 688](#) 了解详情。(由 Shantanu Jain 在 [gh-91896](#) 中贡献。)

10.5 计划在 Python 3.19 中移除

- `ctypes`:
 - 在非 Windows 平台上，通过设置 `_pack_` 而非 `_layout_`，隐式切换到与 MSVC 兼容的结构布局。

10.6 计划在未来版本中移除

以下 API 将会被移除，尽管具体时间还未确定。

- `argparse`:
 - 嵌套参数组和嵌套互斥组已被弃用。
 - 将未写入文档的关键字参数 `prefix_chars` 传递给 `add_argument_group()` 的做法现在已被弃用。
 - `argparse.FileType` 类型转换器已弃用。
- `builtins`:
 - `bool(NotImplemented)`。
 - 生成器: `throw(type, exc, tb)` 和 `athrow(type, exc, tb)` 签名已被弃用: 请改用 `throw(exc)` 和 `athrow(exc)`，即单参数签名。

- 目前 Python 接受数字类字面值后面紧跟关键字的写法，例如 `0 in x, 1 or x, 0 if 1 else 2`。它允许像 `[0x1 for x in y]` 这样令人困惑且有歧义的表达式（它可以被解读为 `[0x1 for x in y]` 或者 `[0x1f or x in y]`）。如果数字类字面值后面紧跟关键字 `and`, `else`, `for`, `if`, `in`, `is` 和 `or` 中的一个将会引发语法警告。在未来的版本中它将改为语法错误。（[gh-87999](#)）
- 对 `__index__()` 和 `__int__()` 方法返回非 `int` 类型的支持：将要求这些方法必须返回 `int` 的子类的实例。
- 对 `__float__()` 方法返回 `float` 的子类的支持：将要求这些方法必须返回 `float` 的实例。
- 对 `__complex__()` 方法返回 `complex` 的子类的支持：将要求这些方法必须返回 `complex` 的实例。
- 将 `int()` 委托给 `__trunc__()` 方法。
- 传入一个复数作为 `complex()` 构造器中的 *real* 或 *imag* 参数的做法现在已被弃用；它应当仅作为单个位置参数被传入。（由 [Serhiy Storchaka](#) 在 [gh-109218](#) 中贡献。）
- `calendar`: `calendar.January` 和 `calendar.February` 常量已被弃用并由 `calendar.JANUARY` 和 `calendar.FEBRUARY` 替代。（由 [Prince Roshan](#) 在 [gh-103636](#) 中贡献。）
- `codecs`: `codecs.open()` 请改用 `open()`。（[gh-133038](#)）
- `codeobject.co_lnotab`: 改用 `codeobject.co_lines()` 方法。
- `datetime`:
 - `utcnow()`: 使用 `datetime.datetime.now(tz=datetime.UTC)`。
 - `utcfromtimestamp()`: 使用 `datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC)`。
- `gettext`: 复数值必须是一个整数。
- `importlib`:
 - `cache_from_source()` *debug_override* 形参已被弃用：改用 *optimization* 形参。
- `importlib.metadata`:
 - `EntryPoints` 元组接口。
 - 返回值中隐式的 `None`。
- `logging`: `warn()` 方法自 Python 3.3 起已被弃用，请改用 `warning()`。
- `mailbox`: 对 `StringIO` 输入和文本模式的使用已被弃用，改用 `BytesIO` 和二进制模式。
- `os`: 在多线程的进程中调用 `os.register_at_fork()`。
- `pydoc.ErrorDuringImport`: 使用元组值作为 *exc_info* 形参的做法已被弃用，应使用异常实例。
- `re`: 现在对于正则表达式中的数字分组引用和分组名称将应用更严格的规则。现在只接受 ASCII 数字序列作为数字引用。字节串模式和替换字符串中的分组名称现在只能包含 ASCII 字母和数字以及下划线。（由 [Serhiy Storchaka](#) 在 [gh-91760](#) 中贡献。）
- `sre_compile`, `sre_constants` 和 `sre_parse` 模块。
- `shutil`: `rmtree()` 的 *onerror* 形参在 Python 3.12 中已被弃用；请改用 *onexc* 形参。
- `ssl` 选项和协议：
 - `ssl.SSLContext` 不带 `protocol` 参数的做法已被弃用。
 - `ssl.SSLContext`: `set_npn_protocols()` 和 `selected_npn_protocol()` 已被弃用：请改用 `ALPN`。
 - `ssl.OP_NO_SSL*` 选项
 - `ssl.OP_NO_TLS*` 选项
 - `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` 的方法：
 - `threading.Condition.notifyAll()`: 使用 `notify_all()`。
 - `threading.Event.isSet()`: 使用 `is_set()`。
 - `threading.Thread.isDaemon()`, `threading.Thread.setDaemon()`: 使用 `threading.Thread.daemon` 属性。
 - `threading.Thread.getName()`, `threading.Thread.setName()`: 使用 `threading.Thread.name` 属性。
 - `threading.currentThread()`: 使用 `threading.current_thread()`。
 - `threading.activeCount()`: 使用 `threading.active_count()`。
- `typing.Text` ([gh-92332](#))。
- 内部类 `typing._UnionGenericAlias` 不再用于实现 `typing.Union`。为了保护使用该私有类的用户的兼容性，将至少在 **Python 3.17** 之前提供兼容性。（由 [Jelle Zijlstra](#) 在 [gh-105499](#) 中贡献。）
- `unittest.IsolatedAsyncioTestCase`: 从测试用例返回不为 `None` 的值的做法已被弃用。
- `urllib.parse` 函数已被弃用：改用 `urlparse()`
 - `splitattr()`
 - `splithost()`
 - `splitnport()`
 - `splitpasswd()`
 - `splitport()`
 - `splitquery()`
 - `splittag()`
 - `splitttype()`
 - `splituser()`
 - `splitvalue()`
 - `to_bytes()`
- `wsgiref.SimpleHandler.stdout.write()` 不应执行部分写入。
- `xml.etree.ElementTree`: 对 `Element` 的真值测试已被弃用。在未来的发布版中它将始终返回 `True`。建议改用显式的 `len(elem)` 或 `elem is not None` 测试。
- `sys._clear_type_cache()` 已弃用，请改用 `sys._clear_internal_caches()`。

11 CPython 字节码的改变

- 将操作码 `BINARY_SUBSCR` 替换为附带操作数 `NB_SUBSCR` 的 `BINARY_OP` 操作码。（由 [Irit Katriel](#) 在 [gh-100239](#) 中贡献。）

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

11.1 Pseudo-instructions

- Add the `ANNOTATIONS_PLACEHOLDER` pseudo instruction to support partially executed module-level annotations with *deferred evaluation of annotations*. (Contributed by Jelle Zijlstra in [gh-130907](#).)
- Add the `BINARY_OP_EXTEND` pseudo instruction, which executes a pair of functions (guard and specialization functions) accessed from the inline cache. (Contributed by Irit Katriel in [gh-100239](#).)
- Add three specializations for `CALL_KW`; `CALL_KW_PY` for calls to Python functions, `CALL_KW_BOUND_METHOD` for calls to bound methods, and `CALL_KW_NON_PY` for all other calls. (Contributed by Mark Shannon in [gh-118093](#).)
- Add the `JUMP_IF_TRUE` and `JUMP_IF_FALSE` pseudo instructions, conditional jumps which do not impact the stack. Replaced by the sequence `COPY 1, TO_BOOL, POP_JUMP_IF_TRUE/FALSE`. (Contributed by Irit Katriel in [gh-124285](#).)
- Add the `LOAD_CONST_MORTAL` pseudo instruction. (Contributed by Mark Shannon in [gh-128685](#).)
- Add the `LOAD_CONST_IMMORTAL` pseudo instruction, which does the same as `LOAD_CONST`, but is more efficient for immortal objects. (Contributed by Mark Shannon in [gh-125837](#).)
- Add the `NOT_TAKEN` pseudo instruction, used by `sys.monitoring` to record branch events (such as `BRANCH_LEFT`). (Contributed by Mark Shannon in [gh-122548](#).)

12 C API 的变化

12.1 C API 中的新特性

- 增加 `Py_PACK_VERSION()` 和 `Py_PACK_FULL_VERSION()`，两个用于对 Python 版本号进行位打包操作的新宏。这在比较 `Py_Version` 或 `PY_VERSION_HEX` 时很有用处。(由 Petr Viktorin 在 [gh-128629](#) 中贡献。)
- 新增 `PyBytes_Join(sep, iterable)` 函数，其功能类似于 Python 中的 `sep.join(iterable)` 操作。(由 Victor Stinner 在 [gh-121645](#) 中贡献。)
- 增加了用于操作当前运行时 Python 解释器配置的函数 (*PEP 741: Python configuration C API*):
 - `PyConfig_Get()`
 - `PyConfig_GetInt()`
 - `PyConfig_Set()`

- PyConfig_Names()

(由 Victor Stinner 在 [gh-107954](#) 中贡献。)

- 增加了用于配置 Python 初始化 (*PEP 741: Python configuration C API*) 的函数:

- 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()

(由 Victor Stinner 在 [gh-107954](#) 中贡献。)

- 增加 Py_fopen() 函数用于打开文件。该函数功能类似于标准 C fopen() 函数, 但它接受一个 Python 对象作为 path 形参并会在出错时设置异常。对应的新增函数 Py_fclose() 函数应当用来关闭文件。(由 Victor Stinner 在 [gh-127350](#) 中贡献。)
- 新增 Py_HashBuffer() 函数用于计算并返回缓冲区的哈希值。(由 Antoine Pitrou 和 Victor Stinner 在 [gh-122854](#) 中贡献。)
- 新增 PyImport_ImportModuleAttr() 和 PyImport_ImportModuleAttrString() 辅助函数, 用于导入模块并获取该模块的属性。(由 Victor Stinner 在 [gh-128911](#) 中贡献。)
- 新增 PyIter_NextItem() 函数以替代返回值含义模糊的 PyIter_Next() 函数。(由 Irit Katriel 和 Erlend Aasland 在 [gh-105201](#) 中贡献。)
- 新增 PyLong_GetSign() 函数用于获取 int 对象的符号位。(由 Sergey B Kirpichev 在 [gh-116560](#) 中贡献。)
- 新增 PyLong_IsPositive()、PyLong_IsNegative() 和 PyLong_IsZero() 三个函数, 分别用于检查 PyLongObject 是否为正数、负数或零。(由 James Roy 和 Sergey B Kirpichev 在 [gh-126061](#) 中贡献。)
- 新增用于在 C <stdint.h> 数字类型与 Python int 对象之间进行转换的新函数:

- PyLong_AsInt32()

- PyLong_AsInt64()

- PyLong_AsUInt32()

- PyLong_AsUInt64()

- PyLong_FromInt32()

- PyLong_FromInt64()

- PyLong_FromUInt32()

- PyLong_FromUInt64()

(由 Victor Stinner 在 [gh-120389](#) 中贡献。)

- 新增用于 Python int 对象的导入导出 API (参见 [PEP 757](#)):
 - `PyLong_GetNativeLayout()`
 - `PyLong_Export()`
 - `PyLong_FreeExport()`
 - `PyLongWriter_Create()`
 - `PyLongWriter_Finish()`
 - `PyLongWriter_Discard()`
 (由 [Sergey B Kirpichev](#) 和 [Victor Stinner](#) 在 [gh-102471](#) 中贡献。)
- 增加 `PyMonitoring_FireBranchLeftEvent()` 和 `PyMonitoring_FireBranchRightEvent()` 分别用于生成 `BRANCH_LEFT` 和 `BRANCH_RIGHT` 事件。(由 [Mark Shannon](#) 在 [gh-122548](#) 中贡献。)
- 新增 `PyType_Freeze()` 函数用于将类型设为不可变。(由 [Victor Stinner](#) 在 [gh-121654](#) 中贡献。)
- 增加 `PyType_GetBaseByToken()` 和 `Py_tp_token` 槽位用于简化超类的标识, 该特性尝试解决在 [PEP 630](#) 中提到的类型检查问题。(在 [gh-124153](#) 中贡献。)
- 新增 `PyUnicode_Equal()` 函数用于测试两个字符串是否相等。此函数也被加入到受限 C API 中。(由 [Victor Stinner](#) 在 [gh-124502](#) 中贡献。)
- 新增 `PyUnicodeWriter` API 用于创建 Python str 对象, 具有下列函数:
 - `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()`
 (由 [Victor Stinner](#) 在 [gh-119182](#) 中贡献。)
- 在 `PyArg_ParseTuple()` 及相关函数中, `k` 和 `K` 格式现在会优先使用 `__index__()` 方法 (如果可用), 与其他整数格式的处理方式保持一致。(由 [Serhiy Storchaka](#) 在 [gh-112068](#) 中贡献。)
- 在 `Py_BuildValue()` 中增加对新的 `p` 格式单元的支持, 它可根据 C 整数产生 Python bool 对象。(由 [Pablo Galindo](#) 在 [bpo-45325](#) 中贡献。)
- 增加 `PyUnstable_IsImmortal()` 以确定一个对象是否为 `immortal`, 用于调试目的。(由 [Peter Bierma](#) 在 [gh-128509](#) 中贡献。)
- 新增 `PyUnstable_Object_EnableDeferredRefCount()` 函数用于启用延迟引用计数功能, 该功能如 [PEP 703](#) 所述。
- 新增 `PyUnstable_Object_IsUniquelyReferenced()` 函数, 作为 `Py_REFCNT(op) == 1` 的替代方案。(由 [Peter Bierma](#) 在 [gh-133140](#) 中贡献)

- 增加 `PyUnstable_Object_IsUniqueReferencedTemporary()` 用于确定一个对象是否为解释器操作数栈上的唯一临时对象。该函数在某些情况下可被用作通过检查 `Py_REFCNT()` 是否为 1 来判断作为参数传给 C API 函数的 Python 对象的替代方案。(由 Sam Gross 在 [gh-133164](#) 中贡献。)

12.2 受限 C API 的变化

- 在受限 C API 3.14 及更新版本中, `Py_TYPE()` 和 `Py_REFCNT()` 现在被实现为不透明函数来隐藏实现细节。(由 Victor Stinner 在 [gh-120600](#) 和 [gh-124127](#) 中贡献。)
- 从受限 C API 移除了 `PySequence_Fast_GET_SIZE`, `PySequence_Fast_GET_ITEM` 和 `PySequence_Fast_ITEMS` 宏, 因为它们在受限 C API 始终是不可用的。(由 Victor Stinner 在 [gh-91417](#) 中贡献。)

12.3 被移除的 C API

- 使用可变基类创建不可变类型的做法自 Python 3.12 起已被弃用, 现在将会引发 `TypeError`。(由 Nikita Sobolev 在 [gh-119775](#) 中贡献。)
- 移除在 Python 3.12 中已被弃用的 `PyDictObject.ma_version_tag` 成员。请改用 `PyDict_AddWatcher()` API。(由 Sam Gross 在 [gh-124296](#) 中贡献。)
- 移除私有函数 `_Py_InitializeMain()`。该函数是 **PEP 587** 在 Python 3.8 中引入的 provisional API。(由 Victor Stinner 在 [gh-129033](#) 中贡献。)
- 移除了未写入文档的 API `Py_C_RECURSION_LIMIT` 和 `PyThreadState.c_recursion_remaining`。它们在 3.13 中增加并且未宣布弃用即被移除。请在 C 代码中使用 `Py_EnterRecursiveCall()` 来防范无限递归问题。(由 Petr Viktorin 在 [gh-133079](#) 中移除, 另请参阅 [gh-130396](#)。)

12.4 已弃用的 C API

- 现在 `Py_HUGE_VAL` 宏已被设为 `soft deprecated`。请改用 `Py_INFINITY`。(由 Sergey B Kirpichev 在 [gh-120026](#) 中贡献。)
- 现在 `Py_IS_NAN`, `Py_IS_INFINITY` 和 `Py_IS_FINITE` 等宏已被设为 `soft deprecated`。请改用 `isnan`, `isinf` 和 `isfinite`, 它们自 C99 起在 `math.h` 中可用。(由 Sergey B Kirpichev 在 [gh-119613](#) 中贡献。)
- 非元组序列当 `items` 包含存储借入缓冲区或 `borrowed reference` 的格式单元的情况下在 `PyArg_ParseTuple()` 和其他参数解析函数中用作 `(items)` 格式单元参数的做法现在已被弃用。(由 Serhiy Storchaka 在 [gh-50333](#) 中贡献。)
- 现在 `_PyMonitoring_FireBranchEvent` 函数已被弃用并应当替换为对 `PyMonitoring_FireBranchLeftEvent()` 和 `PyMonitoring_FireBranchRightEvent()` 的调用。
- 之前未写入文档的函数 `PySequence_In()` 现在被设为 `soft deprecated`。请改用 `PySequence_Contains()`。(由 Yuki Kobayashi 在 [gh-127896](#) 中贡献。)

计划在 Python 3.15 中移除

- `The PyImport_ImportModuleNoBlock()`: 改用 `PyImport_ImportModule()`。
- `PyWeakref_GetObject()` 和 `PyWeakref_GET_OBJECT()`: 改用 `PyWeakref_GetRef()`。在 Python 3.12 及更旧的版本中可以使用 `pythoncapi-compat` 项目来获取 `PyWeakref_GetRef()`。
- `Py_UNICODE` 类型和 `Py_UNICODE_WIDE` 宏: 改用 `wchar_t`。
- `PyUnicode_AsDecodedObject()`: 改用 `PyCodec_Decode()`。
- `PyUnicode_AsDecodedUnicode()`: 改用 `PyCodec_Decode()`; 请注意某些编解码器(例如“base64”)可能返回 `str` 以外的类型, 比如 `bytes`。
- `PyUnicode_AsEncodedObject()`: 改用 `PyCodec_Encode()`。
- `PyUnicode_AsEncodedUnicode()`: 使用 `PyCodec_Encode()` 代替; 请注意, 某些编解码器(如“base64”)可能返回 `bytes` 之外的类型, 如 `str`。

- Python 初始化函数, Python 3.13 中弃用:

- `Py_GetPath()`: 使用 `PyConfig_Get("module_search_paths")` (`sys.path`) 代替。
- `Py_GetPrefix()`: 使用 `PyConfig_Get("base_prefix")` (`sys.base_prefix`) 代替。如果需要处理 **virtual environments**, 请使用 `PyConfig_Get("prefix")` (`sys.prefix`)。
- `Py_GetExecPrefix()`: 使用 `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) 代替。如果需要处理 **virtual environments**, 请使用 `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`)。
- `Py_GetProgramFullPath()`: 使用 `PyConfig_Get("executable")` (`sys.executable`) 代替。
- `Py_GetProgramName()`: 使用 `PyConfig_Get("executable")` (`sys.executable`) 代替。
- `Py_GetPythonHome()`: 使用 `PyConfig_Get("home")` 或 `PYTHONHOME` 环境变量代替。

在 Python 3.13 和更旧的版本中可以使用 [pythoncapi-compat](#) 项目 来获取 `PyConfig_Get()`。

- 用于配置 Python 的初始化的函数, 在 Python 3.11 中已弃用:

- `PySys_SetArgvEx()`: 改为设置 `PyConfig.argv`。
- `PySys_SetArgv()`: 改为设置 `PyConfig.argv`。
- `Py_SetProgramName()`: 改为设置 `PyConfig.program_name`。
- `Py_SetPythonHome()`: 改为设置 `PyConfig.home`。
- `PySys_ResetWarnOptions()`: 改为清除 `sys.warnoptions` 和 `warnings.filters`。

`Py_InitializeFromConfig()` API 应与 `PyConfig` 一起使用。

- 全局配置变量:

- `Py_DebugFlag`: 改用 `PyConfig.parser_debug` 或 `PyConfig_Get("parser_debug")`。
- `Py_VerboseFlag`: 改用 `PyConfig.verbose` 或 `PyConfig_Get("verbose")`。
- `Py_QuietFlag`: 改用 `PyConfig.quiet` 或 `PyConfig_Get("quiet")`。
- `Py_InteractiveFlag`: 改用 `PyConfig.interactive` 或 `PyConfig_Get("interactive")`。
- `Py_InspectFlag`: 改用 `PyConfig.inspect` 或 `PyConfig_Get("inspect")`。
- `Py_OptimizeFlag`: 改用 `PyConfig.optimization_level` 或 `PyConfig_Get("optimization_level")`。
- `Py_NoSiteFlag`: 改用 `PyConfig.site_import` 或 `PyConfig_Get("site_import")`。
- `Py_BytesWarningFlag`: 改用 `PyConfig.bytes_warning` 或 `PyConfig_Get("bytes_warning")`。
- `Py_FrozenFlag`: 使用 `PyConfig.pathconfig_warnings` 或 `PyConfig_Get("pathconfig_warnings")` 代替。
- `Py_IgnoreEnvironmentFlag`: 使用 `PyConfig.use_environment` 或 `PyConfig_Get("use_environment")` 代替。
- `Py_DontWriteBytecodeFlag`: 使用 `PyConfig.write_bytecode` 或 `PyConfig_Get("write_bytecode")` 代替。
- `Py_NoUserSiteDirectory`: 使用 `PyConfig.user_site_directory` 或 `PyConfig_Get("user_site_directory")` 代替。
- `Py_UnbufferedStdioFlag`: 使用 `PyConfig.buffered_stdio` 或 `PyConfig_Get("buffered_stdio")` 代替。
- `Py_HashRandomizationFlag`: 使用 `PyConfig.use_hash_seed` 和 `PyConfig.hash_seed` 或 `PyConfig_Get("hash_seed")` 代替。
- `Py_IsolatedFlag`: 使用 `PyConfig.isolated` 或 `PyConfig_Get("isolated")` 代替。

- `Py_LegacyWindowsFSEncodingFlag`: 使用 `PyPreConfig.legacy_windows_fs_encoding` 或 `PyConfig_Get("legacy_windows_fs_encoding")` 代替。
- `Py_LegacyWindowsStdioFlag`: 使用 `PyConfig.legacy_windows_stdio` 或 `PyConfig_Get("legacy_windows_stdio")` 代替。
- `Py_FileSystemDefaultEncoding`, `Py_HasFileSystemDefaultEncoding`: 使用 `PyConfig.filesystem_encoding` 或 `PyConfig_Get("filesystem_encoding")` 代替。
- `Py_FileSystemDefaultEncodeErrors`: 使用 `PyConfig.filesystem_errors` 或 `PyConfig_Get("filesystem_errors")` 代替。
- `Py_UTF8Mode`: 使用 `PyPreConfig.utf8_mode` 或 `PyConfig_Get("utf8_mode")` 代替。(参见 `Py_PreInitialize()`)。

`Py_InitializeFromConfig()` API 应与 `PyConfig` 一起使用, 以设置这些选项。或者使用 `PyConfig_Get()` 在运行时获取这些选项。

计划在 Python 3.16 中移除

- 捆绑的 `libmpdec` 副本。

计划在 Python 3.18 中移除

- 以下私有函数已被弃用, 并计划在 Python 3.18 中移除:
 - `_PyBytes_Join()`: 使用 `PyBytes_Join()`。
 - `_PyDict_GetItemStringWithError()`: 使用 `PyDict_GetItemStringRef()`。
 - `_PyDict_Pop()`: 使用 `PyDict_Pop()`。
 - `_PyLong_Sign()`: 使用 `PyLong_GetSign()`。
 - `_PyLong_FromDigits()` 和 `_PyLong_New()`: 使用 `PyLongWriter_Create()`。
 - `_PyThreadState_UncheckedGet()`: 使用 `PyThreadState_GetUnchecked()`。
 - `_PyUnicode_AsString()`: 使用 `PyUnicode_AsUTF8()`。
 - `_PyUnicodeWriter_Init()`: 将 `_PyUnicodeWriter_Init(&writer)` 替换为 `writer = PyUnicodeWriter_Create(0)`。
 - `_PyUnicodeWriter_Finish()`: 将 `_PyUnicodeWriter_Finish(&writer)` 替换为 `PyUnicodeWriter_Finish(writer)`。
 - `_PyUnicodeWriter_Dealloc()`: 将 `_PyUnicodeWriter_Dealloc(&writer)` 替换为 `PyUnicodeWriter_Discard(writer)`。
 - `_PyUnicodeWriter_WriteChar()`: 将 `_PyUnicodeWriter_WriteChar(&writer, ch)` 替换为 `PyUnicodeWriter_WriteChar(writer, ch)`。
 - `_PyUnicodeWriter_WriteStr()`: 将 `_PyUnicodeWriter_WriteStr(&writer, str)` 替换为 `PyUnicodeWriter_WriteStr(writer, str)`。
 - `_PyUnicodeWriter_WriteSubstring()`: 将 `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` 替换为 `PyUnicodeWriter_WriteSubstring(writer, str, start, end)`。
 - `_PyUnicodeWriter_WriteASCIIString()`: 请将 `_PyUnicodeWriter_WriteASCIIString(&writer, str)` 替换为 `PyUnicodeWriter_WriteASCII(writer, str)`。
 - `_PyUnicodeWriter_WriteLatin1String()`: 将 `_PyUnicodeWriter_WriteLatin1String(&writer, str)` 替换为 `PyUnicodeWriter_WriteUTF8(writer, str)`。
 - `_PyUnicodeWriter_Prepare()`: (无替代)。
 - `_PyUnicodeWriter_PrepareKind()`: (无替代)。
 - `_Py_HashPointer()`: 使用 `Py_HashPointer()`。

- `_Py_fopen_obj()`: 使用 `Py_fopen()`。

`pythoncapi-compat` 项目 可被用于在 Python 3.13 及更早版本中获取这些新的公有函数。(由 Victor Stinner 在 [gh-128863](#) 中贡献。)

计划在未来版本中移除

以下 API 已被弃用，将被移除，但目前尚未确定移除日期。

- `Py_TPFLAGS_HAVE_FINALIZE`: 自 Python 3.8 起不再需要。
- `PyErr_Fetch()`: 改用 `PyErr_GetRaisedException()`。
- `PyErr_NormalizeException()`: 改用 `PyErr_GetRaisedException()`。
- `PyErr_Restore()`: 改用 `PyErr_SetRaisedException()`。
- `PyModule_GetFilename()`: 改用 `PyModule_GetFilenameObject()`。
- `PyOS_AfterFork()`: 改用 `PyOS_AfterFork_Child()`。
- `PySlice_GetIndicesEx()`: 改用 `PySlice_Unpack()` 和 `PySlice_AdjustIndices()`。
- `PyUnicode_READY()`: 自 Python 3.12 起不再需要
- `PyErr_Display()`: 改用 `PyErr_DisplayException()`。
- `_PyErr_ChainExceptions()`: 改用 `_PyErr_ChainExceptions1()`。
- `PyBytesObject.ob_shash` 成员: 改为调用 `PyObject_Hash()`。
- 线程本地存储 (TLS) API:
 - `PyThread_create_key()`: 改用 `PyThread_tss_alloc()`。
 - `PyThread_delete_key()`: 改用 `PyThread_tss_free()`。
 - `PyThread_set_key_value()`: 改用 `PyThread_tss_set()`。
 - `PyThread_get_key_value()`: 改用 `PyThread_tss_get()`。
 - `PyThread_delete_key_value()`: 改用 `PyThread_tss_delete()`。
 - `PyThread_ReInitTLS()`: 自 Python 3.7 起不再需要。

13 构建的变化

- 生成 `configure` 文件现在需要 GNU Autoconf 2.72 版本。(由 Erlend Aasland 在 [gh-115765](#) 中贡献。)
- `wasm32-unknown-emsripten` 现已成为 **PEP 11** 标准中的第三级 (Tier 3) 支持平台。(由 R. Hood Chatham 在 [gh-127146](#)、[gh-127683](#) 和 [gh-136931](#) 中贡献。)
- 现在可以通过 `Py_NO_LINK_LIB` 关闭基于 `#pragma` 与 `python3*.lib` 的链接。(由 Jean-Christophe Fillion-Robin 在 [gh-82909](#) 中贡献。)
- CPython 现在默认启用了一组推荐的编译器选项以增强安全性。如需禁用这些选项，可使用 `--disable-safety` `configure` 选项；若希望启用更大范围的编译器安全选项（但会牺牲性能），则可使用 `--enable-slower-safety` 选项。
- `WITH_FREELISTS` 宏和 `--without-freelists` `configure` 选项已被移除。
- 新的 `configure` 配置选项 `--with-tail-call-interp` 可用于启用实验性的尾调用解释器。有关更多详情，请参阅[一种新型的解釋器](#)。
- 要禁用新的远程调试支持，请使用 `--without-remote-debug` `configure` 选项。出于安全原因，这可能会有用。

13.1 build-details.json

现在的 Python 安装包中包含一个新文件: `build-details.json`。这是一个静态的 JSON 文档, 包含 CPython 的构建详情, 无需运行代码即可进行内省。这对于 Python 启动器、交叉编译等使用场景很有帮助。

`build-details.json` 必须安装在与平台无关的标准库目录中。这对应于 `'stdlib'` `sysconfig` 安装路径, 可通过运行 `sysconfig.get_path('stdlib')` 找到该路径。

参见

PEP 739 —— `build-details.json 1.0` —— Python 构建详情的静态描述文件

13.2 PGP 签名的停用

Python 3.14 及未来的版本将不再提供 PGP (Pretty Good Privacy) 签名。要验证 CPython 制品, 用户必须使用 [Sigstore](#) 验证材料。自 Python 3.11 起的发布版就已采用 [Sigstore](#) 进行签名。

发布流程中的这一变更已在 **PEP 761** 中明确规定。

14 移植到 Python 3.14

本节列出了先前描述的更改以及可能需要更改代码的其他错误修正。

14.1 Python API 的变化

- `functools.partial` 现在是一个方法描述符。若需保持原有行为, 请将其包装在 `staticmethod()` 中。(由 Serhiy Storchaka 和 Dominykas Grigonis 在 [gh-121027](#) 中贡献。)
- 垃圾回收器现在是增量式的, 这意味着 `gc.collect()` 的行为有轻微改变:
 - `gc.collect(1)`: 执行一次增量式垃圾回收, 而非专门回收第 1 代对象。
 - 对 `gc.collect()` 的其他调用保持不变。
- `locale.nl_langinfo()` 函数现在会在某些情况下临时设置 `LC_CTYPE` 区域设置。这种临时修改会影响其他线程。(由 Serhiy Storchaka 在 [gh-69998](#) 中贡献。)
- `types.UnionType` 现在是 `typing.Union` 的别名, 这会导致部分行为发生变化。更多细节请参见 [上文](#)。(由 Jelle Zijlstra 在 [gh-105499](#) 中贡献。)
- 注解的运行时行为在多个方面发生了变化; 详见 [上文](#)。虽然大多数与注解交互的代码应该能继续正常工作, 但某些未记录的细节可能会有不同的表现。

14.2 C API 的变化

- `Py_Finalize()` 现在会删除所有已驻留字符串。这一变更将不向下兼容任何在调用 `Py_Finalize()` 后仍拥有驻留字符串并在后续调用 `Py_Initialize()` 时重复使用的 C 扩展模块。此行为引发的任何问题通常会导致在后续 `Py_Initialize()` 调用执行期间因访问未初始化内存而引发崩溃。要修复此问题, 应使用地址静化器来标识任何来自驻留字符串的释放后使用并在模块关闭期间释放它。(由 Eddie Elizondo 在 [gh-113601](#) 中贡献。)
- Unicode 异常对象 C API 现在会在异常参数不是 `UnicodeError` 对象时抛出 `TypeError` 异常。(由 Bénédict Tran 在 [gh-127691](#) 中贡献。)
- 解释器在将对象加载到操作数栈时, 会通过尽可能借用引用来避免部分引用计数的修改。这可能导致引用计数值比先前 Python 版本更小。之前通过检查 `Py_REFCNT()` 是否为 1 来判断函数参数是否未被其他代码引用的 C 扩展 API, 现在应改用更安全的替代方案 `PyUnstable_Object_IsUniqueReferencedTemporary()`。
- 以下私有函数已提升为公开 C API:

- `_PyBytes_Join()`: `PyBytes_Join()`
- `_PyLong_IsNegative()`: `PyLong_IsNegative()`
- `_PyLong_IsPositive()`: `PyLong_IsPositive()`
- `_PyLong_IsZero()`: `PyLong_IsZero()`
- `_PyLong_Sign()`: `PyLong_GetSign()`
- `_PyUnicodeWriter_Dealloc()`: `PyUnicodeWriter_Discard()`
- `_PyUnicodeWriter_Finish()`: `PyUnicodeWriter_Finish()`
- `_PyUnicodeWriter_Init()`: 使用 `PyUnicodeWriter_Create()`
- `_PyUnicodeWriter_Prepare()`: (无替代)
- `_PyUnicodeWriter_PrepareKind()`: (无替代)
- `_PyUnicodeWriter_WriteChar()`: `PyUnicodeWriter_WriteChar()`
- `_PyUnicodeWriter_WriteStr()`: `PyUnicodeWriter_WriteStr()`
- `_PyUnicodeWriter_WriteSubstring()`: `PyUnicodeWriter_WriteSubstring()`
- `_PyUnicode_EQ()`: `PyUnicode_Equal()`
- `_PyUnicode_Equal()`: `PyUnicode_Equal()`
- `_Py_GetConfig()`: `PyConfig_Get()` 和 `PyConfig_GetInt()`
- `_Py_HashBytes()`: `Py_HashBuffer()`
- `_Py_fopen_obj()`: `Py_fopen()`
- `PyMutex_IsLocked()`: `PyMutex_IsLocked()`

在 Python 3.13 和更早的版本中可以使用 [pythoncapi-compat project](#) 来充分利用这些新函数。

索引

非字母

环境变量

BROWSER, 29
PYTHON_BASIC_REPL, 15
PYTHON_DISABLE_REMOTE_DEBUG, 7
PYTHON_JIT, 16
PYTHONHOME, 45
PYTHONLEGACYWINDOWSFSENCODING, 37
PYTHONSTARTUP, 15

B

BROWSER, 29

C

Common Vulnerabilities and Exposures

CVE 2024-12718, 27
CVE 2025-4138, 27
CVE 2025-4330, 27
CVE 2025-4435, 27
CVE 2025-4517, 24

P

Python 增强建议; PEP 11, 47
Python 增强建议; PEP 11#tier-3, 16
Python 增强建议; PEP 563, 10
Python 增强建议; PEP 587, 12, 44
Python 增强建议; PEP 626, 36
Python 增强建议; PEP 630#type-checking, 43
Python 增强建议; PEP 649, 4, 9, 10, 27
Python 增强建议; PEP 659, 15
Python 增强建议; PEP 667, 35
Python 增强建议; PEP 684, 5
Python 增强建议; PEP 688#current-options, 38
Python 增强建议; PEP 703, 15, 43
Python 增强建议; PEP 734, 6, 19
Python 增强建议; PEP 739, 48
Python 增强建议; PEP 741, 12
Python 增强建议; PEP 744, 16
Python 增强建议; PEP 745, 4
Python 增强建议; PEP 749, 4, 9, 10, 18
Python 增强建议; PEP 750, 4, 7
Python 增强建议; PEP 757, 43
Python 增强建议; PEP 758, 9
Python 增强建议; PEP 761, 48
Python 增强建议; PEP 765, 18
Python 增强建议; PEP 768, 7, 8
Python 增强建议; PEP 776, 16
Python 增强建议; PEP 779, 5
Python 增强建议; PEP 784, 4, 8
PYTHON_BASIC_REPL, 15
PYTHON_DISABLE_REMOTE_DEBUG, 7
PYTHON_JIT, 16
PYTHONHOME, 45
PYTHONLEGACYWINDOWSFSENCODING, 37

PYTHONSTARTUP, 15

R

RFC

RFC 1494, 23
RFC 2104, 17, 21
RFC 2177, 22
RFC 2361, 23
RFC 3362, 23
RFC 3745, 23
RFC 3950, 23
RFC 4047, 23
RFC 4337, 23
RFC 5334, 23
RFC 6713, 23
RFC 7616, 28
RFC 7903, 23
RFC 8081, 22
RFC 9512, 24
RFC 9559, 23
RFC 9562, 29
RFC 9639, 23