
What's New in Python

Release 3.10.0a5

A. M. Kuchling

March 01, 2021

Python Software Foundation
Email: docs@python.org

Contents

1	Summary – Release highlights	2
2	New Features	2
2.1	Parenthesized context managers	2
2.2	PEP 563: Postponed Evaluation of Annotations Becomes Default	3
2.3	PEP 613: TypeAlias Annotation	4
2.4	PEP 604: New Type Union Operator	4
2.5	PEP 612: Parameter Specification Variables	4
2.6	PEP 634: Structural Pattern Matching	5
2.7	Better error messages in the parser	9
3	Other Language Changes	9
4	New Modules	10
5	Improved Modules	10
5.1	argparse	10
5.2	base64	10
5.3	codecs	10
5.4	collections.abc	10
5.5	contextlib	10
5.6	curses	11
5.7	distutils	11
5.8	doctest	11
5.9	encodings	11
5.10	glob	11
5.11	inspect	11
5.12	linecache	12
5.13	os	12
5.14	pathlib	12
5.15	platform	12
5.16	py_compile	12
5.17	pyclbr	12
5.18	shelve	12
5.19	site	13
5.20	socket	13
5.21	sys	13
5.22	threading	13

5.23	traceback	13
5.24	types	13
5.25	typing	13
5.26	unittest	14
5.27	urllib.parse	14
5.28	xml	14
5.29	zipimport	14
6	Optimizations	14
7	Deprecated	15
8	Removed	15
9	Porting to Python 3.10	16
9.1	Changes in the Python API	16
10	CPython bytecode changes	17
11	Build Changes	17
12	C API Changes	18
12.1	New Features	18
12.2	Porting to Python 3.10	18
12.3	Deprecated	19
12.4	Removed	19
	Index	21

Release 3.10.0a5

Date March 01, 2021

This article explains the new features in Python 3.10, compared to 3.9.

For full details, see the changelog.

Note: Prerelease users should be aware that this document is currently in draft form. It will be updated substantially as Python 3.10 moves towards release, so it's worth checking back even after reading earlier versions.

1 Summary – Release highlights

2 New Features

2.1 Parenthesized context managers

Using enclosing parentheses for continuation across multiple lines in context managers is now supported. This allows formatting a long collection of context managers in multiple lines in a similar way as it was previously possible with import statements. For instance, all these examples are now valid:

```
with (CtxManager() as example):
    ...

with (
```

(continues on next page)

(continued from previous page)

```
    CtxManager1(),
    CtxManager2()
):
    ...

with (CtxManager1() as example,
      CtxManager2()):
    ...

with (CtxManager1(),
      CtxManager2() as example):
    ...

with (
    CtxManager1() as example1,
    CtxManager2() as example2
):
    ...
```

it is also possible to use a trailing comma at the end of the enclosed group:

```
with (
    CtxManager1() as example1,
    CtxManager2() as example2,
    CtxManager3() as example3,
):
    ...
```

This new syntax uses the non LL(1) capacities of the new parser. Check [PEP 617](#) for more details.

(Contributed by Guido van Rossum, Pablo Galindo and Lysandros Nikolaou in [bpo-12782](#) and [bpo-40334](#).)

2.2 PEP 563: Postponed Evaluation of Annotations Becomes Default

In Python 3.7, postponed evaluation of annotations was added, to be enabled with a `from __future__ import annotations` directive. In 3.10 this became the default behavior, even without that future directive. With this being default, all annotations stored in `__annotations__` will be strings. If needed, annotations can be resolved at runtime using `typing.get_type_hints()`. See [PEP 563](#) for a full description. Also, the `inspect.signature()` will try to resolve types from now on, and when it fails it will fall back to showing the string annotations. (Contributed by Batuhan Taskaya in [bpo-38605](#).)

- The `int` type has a new method `int.bit_count()`, returning the number of ones in the binary expansion of a given integer, also known as the population count. (Contributed by Niklas Fiekas in [bpo-29882](#).)
- The views returned by `dict.keys()`, `dict.values()` and `dict.items()` now all have a mapping attribute that gives a `types.MappingProxyType` object wrapping the original dictionary. (Contributed by Dennis Sweeney in [bpo-40890](#).)
- [PEP 618](#): The `zip()` function now has an optional `strict` flag, used to require that all the iterables have an equal length.

2.3 PEP 613: TypeAlias Annotation

PEP 484 introduced the concept of type aliases, only requiring them to be top-level unannotated assignments. This simplicity sometimes made it difficult for type checkers to distinguish between type aliases and ordinary assignments, especially when forward references or invalid types were involved. Compare:

```
StrCache = 'Cache[str]' # a type alias
LOG_PREFIX = 'LOG[DEBUG]' # a module constant
```

Now the `typing` module has a special annotation `TypeAlias` to declare type aliases more explicitly:

```
StrCache: TypeAlias = 'Cache[str]' # a type alias
LOG_PREFIX = 'LOG[DEBUG]' # a module constant
```

See **PEP 613** for more details.

(Contributed by Mikhail Golubev in [bpo-41923](#).)

2.4 PEP 604: New Type Union Operator

A new type union operator was introduced which enables the syntax `X | Y`. This provides a cleaner way of expressing ‘either type `X` or type `Y`’ instead of using `typing.Union`, especially in type hints (annotations).

In previous versions of Python, to apply a type hint for functions accepting arguments of multiple types, `typing.Union` was used:

```
def square(number: Union[int, float]) -> Union[int, float]:
    return number ** 2
```

Type hints can now be written in a more succinct manner:

```
def square(number: int | float) -> int | float:
    return number ** 2
```

This new syntax is also accepted as the second argument to `isinstance()` and `issubclass()`:

```
>>> isinstance(1, int | str)
True
```

See `types-union` and **PEP 604** for more details.

(Contributed by Maggie Moss and Philippe Prados in [bpo-41428](#).)

2.5 PEP 612: Parameter Specification Variables

Two new options to improve the information provided to static type checkers for **PEP 484**’s `Callable` have been added to the `typing` module.

The first is the parameter specification variable. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. Examples of usage can be found in `typing.ParamSpec`. Previously, there was no easy way to type annotate dependency of parameter types in such a precise manner.

The second option is the new `Concatenate` operator. It’s used in conjunction with parameter specification variables to type annotate a higher order callable which adds or removes parameters of another callable. Examples of usage can be found in `typing.Concatenate`.

See `typing.Callable`, `typing.ParamSpec`, `typing.Concatenate` and **PEP 612** for more details.

(Contributed by Ken Jin in [bpo-41559](#).)

2.6 PEP 634: Structural Pattern Matching

Structural pattern matching has been added in the form of a *match statement* and *case statements* of patterns with associated actions. Patterns consist of sequences, mappings, primitive data types as well as class instances. Pattern matching enables programs to extract information from complex data types, branch on the structure of data, and apply specific actions based on different forms of data.

Syntax and operations

The generic syntax of pattern matching is:

```
match subject:
    case <pattern_1>:
        <action_1>
    case <pattern_2>:
        <action_2>
    case <pattern_3>:
        <action_3>
    case _:
        <action_wildcard>
```

A match statement takes an expression and compares its value to successive patterns given as one or more case blocks. Specifically, pattern matching operates by:

1. using data with type and shape (the `subject`)
2. evaluating the `subject` in the `match` statement
3. comparing the subject with each pattern in a `case` statement from top to bottom until a match is confirmed.
4. executing the action associated with the pattern of the confirmed match
5. If an exact match is not confirmed, the last case, a wildcard `_`, if provided, will be used as the matching case. If an exact match is not confirmed and a wildcard case does not exist, the entire match block is a no-op.

Declarative approach

Readers may be aware of pattern matching through the simple example of matching a subject (data object) to a literal (pattern) with the switch statement found in C, Java or JavaScript (and many other languages). Often the switch statement is used for comparison of an object/expression with case statements containing literals.

More powerful examples of pattern matching can be found in languages, such as Scala and Elixir. With structural pattern matching, the approach is “declarative” and explicitly states the conditions (the patterns) for data to match.

While an “imperative” series of instructions using nested “if” statements could be used to accomplish something similar to structural pattern matching, it is less clear than the “declarative” approach. Instead the “declarative” approach states the conditions to meet for a match and is more readable through its explicit patterns. While structural pattern matching can be used in its simplest form comparing a variable to a literal in a case statement, its true value for Python lies in its handling of the subject’s type and shape.

Simple pattern: match to a literal

Let's look at this example as pattern matching in its simplest form: a value, the subject, being matched to several literals, the patterns. In the example below, `status` is the subject of the match statement. The patterns are each of the case statements, where literals represent request status codes. The associated action to the case is executed after a match:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the Internet"
```

If the above function is passed a `status` of 418, "I'm a teapot" is returned. If the above function is passed a `status` of 500, the case statement with `_` will match as a wildcard, and "Something's wrong with the Internet" is returned. Note the last block: the variable name, `_`, acts as a *wildcard* and insures the subject will always match. The use of `_` is optional.

You can combine several literals in a single pattern using `|` ("or"):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Behavior without the wildcard

If we modify the above example by removing the last case block, the example becomes:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
```

Without the use of `_` in a case statement, a match may not exist. If no match exists, the behavior is a no-op. For example, if `status` of 500 is passed, a no-op occurs.

Patterns with a literal and variable

Patterns can look like unpacking assignments, and a pattern may be used to bind variables. In this example, a data point can be unpacked to its x-coordinate and y-coordinate:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
```

(continues on next page)

```
case _:
    raise ValueError("Not a point")
```

The first pattern has two literals, `(0, 0)`, and may be thought of as an extension of the literal pattern shown above. The next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

Patterns and classes

If you are using classes to structure your data, you can use as a pattern the class name followed by an argument list resembling a constructor. This pattern has the ability to capture class attributes into variables:

```
class Point:
    x: int
    y: int

def location(point):
    match point:
        case Point(x=0, y=0):
            print("Origin is the point's location.")
        case Point(x=0, y=y):
            print(f"Y={y} and the point is on the y-axis.")
        case Point(x=x, y=0):
            print(f"X={x} and the point is on the x-axis.")
        case Point():
            print("The point is located somewhere else on the plane.")
        case _:
            print("Not a point")
```

Patterns with positional parameters

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. data-classes). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `("x", "y")`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Nested patterns

Patterns can be arbitrarily nested. For example, if our data is a short list of points, it could be matched like this:

```
match points:
    case []:
        print("No points in the list.")
    case [Point(0, 0)]:
        print("The origin is the only point in the list.")
    case [Point(x, y)]:
        print(f"A single point {x}, {y} is in the list.")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two points on the Y axis at {y1}, {y2} are in the list.")
```

(continues on next page)

```
case _:
    print("Something else is found in the list.")
```

Complex patterns and the wildcard

To this point, the examples have used `_` alone in the last case statement. A wildcard can be used in more complex patterns, such as `('error', code, _)`. For example:

```
match test_variable:
    case ('warning', code, 40):
        print("A warning has been received.")
    case ('error', code, _):
        print(f"An error {code} occurred.")
```

In the above case, `test_variable` will match for `('error', code, 100)` and `('error', code, 800)`.

Guard

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"The point is located on the diagonal Y=X at {x}.")
    case Point(x, y):
        print(f"Point is not on the diagonal.")
```

Other Key Features

Several other key features:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. Technically, the subject must be an instance of `collections.abc.Sequence`. Therefore, an important exception is that patterns don't match iterators. Also, to prevent a common mistake, sequence patterns don't match strings.
- Sequence patterns support wildcards: `[x, y, *rest]` and `(x, y, *rest)` work similar to wildcards in unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dict. Unlike sequence patterns, extra keys are ignored. A wildcard `**rest` is also supported. (But `**_` would be redundant, so it not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

This binds `x1`, `y1`, `x2`, `y2` like you would expect without the `as` clause, and `p2` to the entire second item of the subject.

- Most literals are compared by equality. However, the singletons `True`, `False` and `None` are compared by identity.
- Named constants may be used in patterns. These named constants must be dotted names to prevent the constant from being interpreted as a capture variable:


```

from enum import Enum
class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :)")

```

For the full specification see [PEP 634](#). Motivation and rationale are in [PEP 635](#), and a longer tutorial is in [PEP 636](#).

2.7 Better error messages in the parser

When parsing code that contains unclosed parentheses or brackets the interpreter now includes the location of the unclosed bracket of parentheses instead of displaying *SyntaxError: unexpected EOF while parsing* or pointing to some incorrect location. For instance, consider the following code (notice the unclosed '{'):

```

expected = {9: 1, 18: 2, 19: 2, 27: 3, 28: 3, 29: 3, 36: 4, 37: 4,
            38: 4, 39: 4, 45: 5, 46: 5, 47: 5, 48: 5, 49: 5, 54: 6,
some_other_code = foo()

```

previous versions of the interpreter reported confusing places as the location of the syntax error:

```

File "example.py", line 3
    some_other_code = foo()
                      ^
SyntaxError: invalid syntax

```

but in Python3.10 a more informative error is emitted:

```

File "example.py", line 1
    expected = {9: 1, 18: 2, 19: 2, 27: 3, 28: 3, 29: 3, 36: 4, 37: 4,
               ^
SyntaxError: '{' was never closed

```

In a similar way, errors involving unclosed string literals (single and triple quoted) now point to the start of the string instead of reporting EOF/EOL.

These improvements are inspired by previous work in the PyPy interpreter.

(Contributed by Pablo Galindo in [bpo-42864](#) and Batuhan Taskaya in [bpo-40176](#).)

3 Other Language Changes

- Builtin and extension functions that take integer arguments no longer accept `Decimals`, `Fractions` and other objects that can be converted to integers only with a loss (e.g. that have the `__int__()` method but do not have the `__index__()` method). (Contributed by Serhiy Storchaka in [bpo-37999](#).)
- If `object.__ipow__()` returns `NotImplemented`, the operator will correctly fall back to `object.__pow__()` and `object.__rpow__()` as expected. (Contributed by Alex Shkop in [bpo-38302](#).)
- Assignment expressions can now be used unparenthesized within set literals and set comprehensions, as well as in sequence indexes (but not slices).

- Functions have a new `__builtins__` attribute which is used to look for builtin symbols when a function is executed, instead of looking into `__globals__['__builtins__']`. The attribute is initialized from `__globals__["__builtins__"]` if it exists, else from the current builtins. (Contributed by Mark Shannon in [bpo-42990](#).)

4 New Modules

- None yet.

5 Improved Modules

5.1 argparse

Misleading phrase “optional arguments” was replaced with “options” in `argparse` help. Some tests might require adaptation if they rely on exact output match. (Contributed by Raymond Hettinger in [bpo-9694](#).)

5.2 base64

Add `base64.b32hexencode()` and `base64.b32hexdecode()` to support the Base32 Encoding with Extended Hex Alphabet.

5.3 codecs

Add a `codecs.unregister()` function to unregister a codec search function. (Contributed by Hai Shi in [bpo-41842](#).)

5.4 collections.abc

The `__args__` of the parameterized generic for `collections.abc.Callable` are now consistent with `typing.Callable`. `collections.abc.Callable` generic now flattens type parameters, similar to what `typing.Callable` currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. To allow this change, `types.GenericAlias` can now be subclassed, and a subclass will be returned when subscripting the `collections.abc.Callable` type. Note that a `TypeError` may be raised for invalid forms of parameterizing `collections.abc.Callable` which may have passed silently in Python 3.9. (Contributed by Ken Jin in [bpo-42195](#).)

5.5 contextlib

Add a `contextlib.aclosing()` context manager to safely close async generators and objects representing asynchronously released resources. (Contributed by Joongi Kim and John Belmonte in [bpo-41229](#).)

Add asynchronous context manager support to `contextlib.nullcontext()`. (Contributed by Tom Gringauz in [bpo-41543](#).)

5.6 curses

The extended color functions added in ncurses 6.1 will be used transparently by `curses.color_content()`, `curses.init_color()`, `curses.init_pair()`, and `curses.pair_content()`. A new function, `curses.has_extended_color_support()`, indicates whether extended color support is provided by the underlying ncurses library. (Contributed by Jeffrey Kintscher and Hans Petter Jansson in [bpo-36982](#).)

The `BUTTON5_*` constants are now exposed in the `curses` module if they are provided by the underlying curses library. (Contributed by Zackery Spytz in [bpo-39273](#).)

5.7 distutils

The entire `distutils` package is deprecated, to be removed in Python 3.12. Its functionality for specifying package builds has already been completely replaced by third-party packages `setuptools` and `packaging`, and most other commonly used APIs are available elsewhere in the standard library (such as `platform`, `shutil`, `subprocess` or `sysconfig`). There are no plans to migrate any other functionality from `distutils`, and applications that are using other functions should plan to make private copies of the code. Refer to [PEP 632](#) for discussion.

The `bdist_wininst` command deprecated in Python 3.8 has been removed. The `bdist_wheel` command is now recommended to distribute binary packages on Windows. (Contributed by Victor Stinner in [bpo-42802](#).)

5.8 doctest

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

5.9 encodings

`encodings.normalize_encoding()` now ignores non-ASCII characters. (Contributed by Hai Shi in [bpo-39337](#).)

5.10 glob

Added the `root_dir` and `dir_fd` parameters in `glob()` and `iglob()` which allow to specify the root directory for searching. (Contributed by Serhiy Storchaka in [bpo-38144](#).)

5.11 inspect

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

Added `globalns` and `localns` parameters in `signature()` and `inspect.Signature.from_callable()` to retrieve the annotations in given local and global namespaces. (Contributed by Batuhan Taskaya in [bpo-41960](#).)

5.12 linecache

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

5.13 os

Added `os.cpu_count()` support for VxWorks RTOS. (Contributed by Peixing Xin in [bpo-41440](#).)

Added a new function `os.eventfd()` and related helpers to wrap the `eventfd2` syscall on Linux. (Contributed by Christian Heimes in [bpo-41001](#).)

Added `os.splice()` that allows to move data between two file descriptors without copying between kernel address space and user address space, where one of the file descriptors must refer to a pipe. (Contributed by Pablo Galindo in [bpo-41625](#).)

Added `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` for macOS. (Contributed by Dong-hee Na in [bpo-43106](#).)

5.14 pathlib

Added slice support to `PurePath.parents`. (Contributed by Joshua Cannon in [bpo-35498](#))

Added negative indexing support to `PurePath.parents`. (Contributed by Yaroslav Pankovych in [bpo-21041](#))

5.15 platform

Added `platform.freedesktop_os_release()` to retrieve operation system identification from [freedesktop.org os-release](#) standard file. (Contributed by Christian Heimes in [bpo-28468](#))

5.16 py_compile

Added `--quiet` option to command-line interface of `py_compile`. (Contributed by Gregory Schevchenko in [bpo-38731](#).)

5.17 pycbr

Added an `end_lineno` attribute to the `Function` and `Class` objects in the tree returned by `pycbr.readline()` and `pycbr.readline_ex()`. It matches the existing (start) `lineno`. (Contributed by Aviral Srivastava in [bpo-38307](#).)

5.18 shelve

The `shelve` module now uses `pickle.DEFAULT_PROTOCOL` by default instead of `pickle` protocol 3 when creating shelves. (Contributed by Zackery Spitz in [bpo-34204](#).)

5.19 site

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

5.20 socket

The exception `socket.timeout` is now an alias of `TimeoutError`. (Contributed by Christian Heimes in [bpo-42413](#).)

5.21 sys

Add `sys.orig_argv` attribute: the list of the original command line arguments passed to the Python executable. (Contributed by Victor Stinner in [bpo-23427](#).)

Add `sys.stdlib_module_names`, containing the list of the standard library module names. (Contributed by Victor Stinner in [bpo-42955](#).)

5.22 threading

Added `threading.gettrace()` and `threading.getprofile()` to retrieve the functions set by `threading.settrace()` and `threading.setprofile()` respectively. (Contributed by Mario Corchero in [bpo-42251](#).)

Add `threading.__excepthook__` to allow retrieving the original value of `threading.excepthook()` in case it is set to a broken or a different value. (Contributed by Mario Corchero in [bpo-42308](#).)

5.23 traceback

The `format_exception()`, `format_exception_only()`, and `print_exception()` functions can now take an exception object as a positional-only argument. (Contributed by Zackery Spytz and Matthias Bussonnier in [bpo-26389](#).)

5.24 types

Reintroduced the `types.EllipsisType`, `types.NoneType` and `types.NotImplementedType` classes, providing a new set of types readily interpretable by type checkers. (Contributed by Bas van Beek in [bpo-41810](#).)

5.25 typing

The behavior of `typing.Literal` was changed to conform with [PEP 586](#) and to match the behavior of static type checkers specified in the PEP.

1. `Literal` now de-duplicates parameters.
2. Equality comparisons between `Literal` objects are now order independent.
3. `Literal` comparisons now respects types. For example, `Literal[0] == Literal[False]` previously evaluated to `True`. It is now `False`. To support this change, the internally used type cache now supports differentiating types.
4. `Literal` objects will now raise a `TypeError` exception during equality comparisons if one of their parameters are not immutable. Note that declaring `Literal` with mutable parameters will not throw an error:

```
>>> from typing import Literal
>>> Literal[{0}]
>>> Literal[{0}] == Literal[{False}]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

(Contributed by Yurii Karabas in [bpo-42345](#).)

5.26 unittest

Add new method `assertNoLogs()` to complement the existing `assertLogs()`. (Contributed by Kit Yan Choi in [bpo-39385](#).)

5.27 urllib.parse

Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separators in `urllib.parse.parse_qs()` and `urllib.parse.parse_qsl()`. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with `&` as the default. This change also affects `cgi.parse()` and `cgi.parse_multipart()` as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in [bpo-42967](#).)

5.28 xml

Add a `LexicalHandler` class to the `xml.sax.handler` module. (Contributed by Jonathan Gossage and Zackery Spytz in [bpo-35018](#).)

5.29 zipimport

Add methods related to [PEP 451](#): `find_spec()`, `zipimport.zipimporter.create_module()`, and `zipimport.zipimporter.exec_module()`. (Contributed by Brett Cannon in [bpo-42131](#).)

6 Optimizations

- Constructors `str()`, `bytes()` and `bytearray()` are now faster (around 30–40% for small objects). (Contributed by Serhiy Storchaka in [bpo-41334](#).)
- The `runpy` module now imports fewer modules. The `python3 -m module-name` command startup time is 1.3x faster in average. (Contributed by Victor Stinner in [bpo-41006](#).)
- The `LOAD_ATTR` instruction now uses new “per opcode cache” mechanism. It is about 36% faster now for regular attributes and 44% faster for slots. (Contributed by Pablo Galindo and Yury Selivanov in [bpo-42093](#) and Guido van Rossum in [bpo-42927](#), based on ideas implemented originally in PyPy and MicroPython.)
- When building Python with `--enable-optimizations` now `-fno-semantic-interposition` is added to both the compile and link line. This speeds builds of the Python interpreter created with `--enable-shared` with `gcc` by up to 30%. See [this article](#) for more details. (Contributed by Victor Stinner and Pablo Galindo in [bpo-38980](#).)
- Function parameters and their annotations are no longer computed at runtime, but rather at compilation time. They are stored as a tuple of strings at the bytecode level. It is now around 100% faster to create a function with parameter annotations. (Contributed by Yurii Karabas and Inada Naoki in [bpo-42202](#).)

- Substring search functions such as `str1 in str2` and `str2.find(str1)` now sometimes use Crochemore & Perrin’s “Two-Way” string searching algorithm to avoid quadratic behavior on long strings. (Contributed by Dennis Sweeney in [bpo-41972](#))

7 Deprecated

- Starting in this release, there will be a concerted effort to begin cleaning up old import semantics that were kept for Python 2.7 compatibility. Specifically, `find_loader()`/`find_module()` (superseded by `find_spec()`), `load_module()` (superseded by `exec_module()`), `module_repr()` (which the import system takes care of for you), the `__package__` attribute (superseded by `__spec__.parent`), the `__loader__` attribute (superseded by `__spec__.loader`), and the `__cached__` attribute (superseded by `__spec__.cached`) will slowly be removed (as well as other classes and methods in `importlib`). `ImportWarning` and/or `DeprecationWarning` will be raised as appropriate to help identify code which needs updating during this transition.
- The entire `distutils` namespace is deprecated, to be removed in Python 3.12. Refer to the [module changes](#) section for more information.
- Non-integer arguments to `random.randrange()` are deprecated. The `ValueError` is deprecated in favor of a `TypeError`. (Contributed by Serhiy Storchaka and Raymond Hettinger in [bpo-37319](#).)
- The various `load_module()` methods of `importlib` have been documented as deprecated since Python 3.6, but will now also trigger a `DeprecationWarning`. Use `exec_module()` instead. (Contributed by Brett Cannon in [bpo-26131](#).)
- `zimport.zipimporter.load_module()` has been deprecated in preference for `exec_module()`. (Contributed by Brett Cannon in [bpo-26131](#).)
- The use of `load_module()` by the import system now triggers an `ImportWarning` as `exec_module()` is preferred. (Contributed by Brett Cannon in [bpo-26131](#).)
- `sqlite3.OptimizedUnicode` has been undocumented and obsolete since Python 3.3, when it was made an alias to `str`. It is now deprecated, scheduled for removal in Python 3.12. (Contributed by Erlend E. Aasland in [bpo-42264](#).)
- The undocumented built-in function `sqlite3.enable_shared_cache` is now deprecated, scheduled for removal in Python 3.12. Its use is strongly discouraged by the SQLite3 documentation. See the [SQLite3 docs](#) for more details. If shared cache must be used, open the database in URI mode using the `cache=shared` query parameter. (Contributed by Erlend E. Aasland in [bpo-24464](#).)

8 Removed

- Removed special methods `__int__`, `__float__`, `__floordiv__`, `__mod__`, `__divmod__`, `__rfloordiv__`, `__rmod__` and `__rdivmod__` of the `complex` class. They always raised a `TypeError`. (Contributed by Serhiy Storchaka in [bpo-41974](#).)
- The `ParserBase.error()` method from the private and undocumented `_markupbase` module has been removed. `html.parser.HTMLParser` is the only subclass of `ParserBase` and its `error()` implementation has already been removed in Python 3.5. (Contributed by Berker Peksag in [bpo-31844](#).)
- Removed the `unicodedata.ucnhash_CAPI` attribute which was an internal PyCapsule object. The related private `_PyUnicode_Name_CAPI` structure was moved to the internal C API. (Contributed by Victor Stinner in [bpo-42157](#).)
- Removed the `parser` module, which was deprecated in 3.9 due to the switch to the new PEG parser, as well as all the C source and header files that were only being used by the old parser, including `node.h`, `parser.h`, `graminit.h` and `grammar.h`.

- Removed the Public C API functions `PyParser_SimpleParseStringFlags()`, `PyParser_SimpleParseStringFlagsFilename()`, `PyParser_SimpleParseFileFlags()` and `PyNode_Compile()` that were deprecated in 3.9 due to the switch to the new PEG parser.
- Removed the `formatter` module, which was deprecated in Python 3.4. It is somewhat obsolete, little used, and not tested. It was originally scheduled to be removed in Python 3.6, but such removals were delayed until after Python 2.7 EOL. Existing users should copy whatever classes they use into their code. (Contributed by Dong-hee Na and Terry J. Reedy in [bpo-42299](#).)
- Removed the `PyModule_GetWarningsModule()` function that was useless now due to the `_warnings` module was converted to a builtin module in 2.6. (Contributed by Hai Shi in [bpo-42599](#).)
- Remove deprecated aliases to collections-abstract-base-classes from the `collections` module. (Contributed by Victor Stinner in [bpo-37324](#).)
- The `loop` parameter has been removed from most of `asyncio`'s high-level API following deprecation in Python 3.8. The motivation behind this change is multifold:
 1. This simplifies the high-level API.
 2. The functions in the high-level API have been implicitly getting the current thread's running event loop since Python 3.7. There isn't a need to pass the event loop to the API in most normal use cases.
 3. Event loop passing is error-prone especially when dealing with loops running in different threads.

Note that the low-level API will still accept `loop`. See [Changes in the Python API](#) for examples of how to replace existing code.

(Contributed by Yurii Karabas, Andrew Svetlov, Yury Selivanov and Kyle Stanley in [bpo-42392](#).)

9 Porting to Python 3.10

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

9.1 Changes in the Python API

- The *etype* parameters of the `format_exception()`, `format_exception_only()`, and `print_exception()` functions in the `traceback` module have been renamed to *exc*. (Contributed by Zackery Spytz and Matthias Bussonnier in [bpo-26389](#).)
- `atexit`: At Python exit, if a callback registered with `atexit.register()` fails, its exception is now logged. Previously, only some exceptions were logged, and the last exception was always silently ignored. (Contributed by Victor Stinner in [bpo-42639](#).)
- `collections.abc.Callable` generic now flattens type parameters, similar to what `typing.Callable` currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. Code which accesses the arguments via `typing.get_args()` or `__args__` need to account for this change. Furthermore, `TypeError` may be raised for invalid forms of parameterizing `collections.abc.Callable` which may have passed silently in Python 3.9. (Contributed by Ken Jin in [bpo-42195](#).)
- `socket.htons()` and `socket.ntohs()` now raise `OverflowError` instead of `DeprecationWarning` if the given parameter will not fit in a 16-bit unsigned integer. (Contributed by Erlend E. Aasland in [bpo-42393](#).)
- The `loop` parameter has been removed from most of `asyncio`'s high-level API following deprecation in Python 3.8.

A coroutine that currently look like this:

```
async def foo(loop):
    await asyncio.sleep(1, loop=loop)
```


Should be replaced with this:

```
async def foo():
    await asyncio.sleep(1)
```

If `foo()` was specifically designed *not* to run in the current thread's running event loop (e.g. running in another thread's event loop), consider using `asyncio.run_coroutine_threadsafe()` instead.

(Contributed by Yurii Karabas, Andrew Svetlov, Yury Selivanov and Kyle Stanley in [bpo-42392](#).)

- The `types.FunctionType` constructor now inherits the current builtins if the `globals` dictionary has no `"__builtins__"` key, rather than using `{"None": None}` as builtins: same behavior as `eval()` and `exec()` functions. Defining a function with `def function(...): ...` in Python is not affected, `globals` cannot be overridden with this syntax: it also inherits the current builtins. (Contributed by Victor Stinner in [bpo-42990](#).)

10 CPython bytecode changes

- The `MAKE_FUNCTION` instruction accepts tuple of strings as annotations instead of dictionary. (Contributed by Yurii Karabas and Inada Naoki in [bpo-42202](#))

11 Build Changes

- The C99 functions `snprintf()` and `vsnprintf()` are now required to build Python. (Contributed by Victor Stinner in [bpo-36020](#).)
- `sqlite3` requires SQLite 3.7.15 or higher. (Contributed by Sergey Fedoseev and Erlend E. Aasland [bpo-40744](#) and [bpo-40810](#).)
- The `atexit` module must now always be built as a built-in module. (Contributed by Victor Stinner in [bpo-42639](#).)
- Added `--disable-test-modules` option to the `configure` script: don't build nor install test modules. (Contributed by Xavier de Gaye, Thomas Petazzoni and Peixing Xin in [bpo-27640](#).)
- Add `--with-wheel-pkg-dir=PATH` option to the `./configure` script. If specified, the `ensurepip` module looks for `setuptools` and `pip` wheel packages in this directory: if both are present, these wheel packages are used instead of `ensurepip` bundled wheel packages.

Some Linux distribution packaging policies recommend against bundling dependencies. For example, Fedora installs wheel packages in the `/usr/share/python-wheels/` directory and don't install the `ensurepip._bundled` package.

(Contributed by Victor Stinner in [bpo-42856](#).)

- Add a new configure `--without-static-libpython` option to not build the `libpythonMAJOR.MINOR.a` static library and not install the `python.o` object file.

(Contributed by Victor Stinner in [bpo-43103](#).)

- The `configure` script now uses the `pkg-config` utility, if available, to detect the location of Tcl/Tk headers and libraries. As before, those locations can be explicitly specified with the `--with-tcltk-includes` and `--with-tcltk-libs` configuration options. (Contributed by Manolis Stamatogiannakis in [bpo-42603](#).)

12 C API Changes

12.1 New Features

- The result of `PyNumber_Index()` now always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`. (Contributed by Serhiy Storchaka in [bpo-40792](#).)
- Add a new `orig_argv` member to the `PyConfig` structure: the list of the original command line arguments passed to the Python executable. (Contributed by Victor Stinner in [bpo-23427](#).)
- The `PyDateTime_DATE_GET_TZINFO()` and `PyDateTime_TIME_GET_TZINFO()` macros have been added for accessing the `tzinfo` attributes of `datetime.datetime` and `datetime.time` objects. (Contributed by Zackery Spytz in [bpo-30155](#).)
- Add a `PyCodec_Unregister()` function to unregister a codec search function. (Contributed by Hai Shi in [bpo-41842](#).)
- The `PyIter_Send()` function was added to allow sending value into iterator without raising `StopIteration` exception. (Contributed by Vladimir Matveev in [bpo-41756](#).)
- Added `PyUnicode_AsUTF8AndSize()` to the limited C API. (Contributed by Alex Gaynor in [bpo-41784](#).)
- Added `PyModule_AddObjectRef()` function: similar to `PyModule_AddObject()` but don't steal a reference to the value on success. (Contributed by Victor Stinner in [bpo-1635741](#).)
- Added `Py_NewRef()` and `Py_XNewRef()` functions to increment the reference count of an object and return the object. (Contributed by Victor Stinner in [bpo-42262](#).)
- The `PyType_FromSpecWithBases()` and `PyType_FromModuleAndSpec()` functions now accept a single class as the *bases* argument. (Contributed by Serhiy Storchaka in [bpo-42423](#).)
- The `PyType_FromModuleAndSpec()` function now accepts `NULL tp_doc` slot. (Contributed by Hai Shi in [bpo-41832](#).)
- The `PyType_GetSlot()` function can accept static types. (Contributed by Hai Shi and Petr Viktorin in [bpo-41073](#).)
- Add a new `PySet_CheckExact()` function to the C-API to check if an object is an instance of `set` but not an instance of a subtype. (Contributed by Pablo Galindo in [bpo-43277](#).)

12.2 Porting to Python 3.10

- The `PY_SSIZE_T_CLEAN` macro must now be defined to use `PyArg_ParseTuple()` and `Py_BuildValue()` formats which use `#`: `es#`, `et#`, `s#`, `u#`, `y#`, `z#`, `U#` and `Z#`. See Parsing arguments and building values and the [PEP 353](#). (Contributed by Victor Stinner in [bpo-40943](#).)
- Since `Py_REFCNT()` is changed to the inline static function, `Py_REFCNT(obj) = new_refcnt` must be replaced with `Py_SET_REFCNT(obj, new_refcnt)`: see `Py_SET_REFCNT()` (available since Python 3.9). For backward compatibility, this macro can be used:

```
#if PY_VERSION_HEX < 0x030900A4
#  define Py_SET_REFCNT(obj, refcnt) ((Py_REFCNT(obj) = (refcnt)), (void)0)
#endif
```

(Contributed by Victor Stinner in [bpo-39573](#).)

- Calling `PyDict_GetItem()` without GIL held had been allowed for historical reason. It is no longer allowed. (Contributed by Victor Stinner in [bpo-40839](#).)
- `PyUnicode_FromUnicode(NULL, size)` and `PyUnicode_FromStringAndSize(NULL, size)` raise `DeprecationWarning` now. Use `PyUnicode_New()` to allocate Unicode object without initial data. (Contributed by Inada Naoki in [bpo-36346](#).)

- The private `_PyUnicode_Name_CAPI` structure of the PyCapsule API `unicodedata.ucnhash_CAPI` has been moved to the internal C API. (Contributed by Victor Stinner in [bpo-42157](#).)
- `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()` and `Py_GetProgramName()` functions now return `NULL` if called before `Py_Initialize()` (before Python is initialized). Use the new Python Initialization Configuration API to get the Python Path Configuration.. (Contributed by Victor Stinner in [bpo-42260](#).)
- `PyList_SET_ITEM()`, `PyTuple_SET_ITEM()` and `PyCell_SET()` macros can no longer be used as l-value or r-value. For example, `x = PyList_SET_ITEM(a, b, c)` and `PyList_SET_ITEM(a, b, c) = x` now fail with a compiler error. It prevents bugs like `if (PyList_SET_ITEM(a, b, c) < 0) ... test`. (Contributed by Zackery Spytz and Victor Stinner in [bpo-30459](#).)
- The non-limited API files `odictobject.h`, `parser_interface.h`, `picklebufobject.h`, `pyarena.h`, `pyctype.h`, `pydebug.h`, `pyfpe.h`, and `pytime.h` have been moved to the `Include/cpython` directory. These files must not be included directly, as they are already included in `Python.h`: Include Files. If they have been included directly, consider including `Python.h` instead. (Contributed by Nicholas Sim in [bpo-35134](#))

12.3 Deprecated

- The `PyUnicode_InternImmortal()` function is now deprecated and will be removed in Python 3.12: use `PyUnicode_InternInPlace()` instead. (Contributed by Victor Stinner in [bpo-41692](#).)

12.4 Removed

- `PyObject_AsCharBuffer()`, `PyObject_AsReadBuffer()`, `PyObject_CheckReadBuffer()`, and `PyObject_AsWriteBuffer()` are removed. Please migrate to new buffer protocol; `PyObject_GetBuffer()` and `PyBuffer_Release()`. (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `Py_UNICODE_str*` functions manipulating `Py_UNICODE*` strings. (Contributed by Inada Naoki in [bpo-41123](#).)
 - `Py_UNICODE_strlen`: use `PyUnicode_GetLength()` or `PyUnicode_GET_LENGTH`
 - `Py_UNICODE_strcat`: use `PyUnicode_CopyCharacters()` or `PyUnicode_FromFormat()`
 - `Py_UNICODE_strcpy`, `Py_UNICODE_strncpy`: use `PyUnicode_CopyCharacters()` or `PyUnicode_Substring()`
 - `Py_UNICODE_strcmp`: use `PyUnicode_Compare()`
 - `Py_UNICODE_strncmp`: use `PyUnicode_Tailmatch()`
 - `Py_UNICODE_strchr`, `Py_UNICODE_strrchr`: use `PyUnicode_FindChar()`
- Removed `PyUnicode_GetMax()`. Please migrate to new ([PEP 393](#)) APIs. (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `PyLong_FromUnicode()`. Please migrate to `PyLong_FromUnicodeObject()`. (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `PyUnicode_AsUnicodeCopy()`. Please use `PyUnicode_AsUCS4Copy()` or `PyUnicode_AsWideCharString()` (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `_Py_CheckRecursionLimit` variable: it has been replaced by `ceval.recursion_limit` of the `PyInterpreterState` structure. (Contributed by Victor Stinner in [bpo-41834](#).)
- Removed undocumented macros `Py_ALLOW_RECURSION` and `Py_END_ALLOW_RECURSION` and the `recursion_critical` field of the `PyInterpreterState` structure. (Contributed by Serhiy Stor-chaka in [bpo-41936](#).)

- Removed the undocumented `PyOS_InitInterrupts()` function. Initializing Python already implicitly installs signal handlers: see `PyConfig.install_signal_handlers`. (Contributed by Victor Stinner in [bpo-41713](#).)

Index

P

Python Enhancement Proposals

- PEP 353, [18](#)
- PEP 393, [19](#)
- PEP 451, [14](#)
- PEP 484, [4](#)
- PEP 563, [3](#)
- PEP 586, [13](#)
- PEP 604, [4](#)
- PEP 612, [4](#)
- PEP 613, [4](#)
- PEP 617, [3](#)
- PEP 618, [3](#)
- PEP 632, [11](#)
- PEP 634, [9](#)
- PEP 635, [9](#)
- PEP 636, [9](#)