What's New In Python 3.6

Editors: Elvis Pranskevichus <elvis@magic.io>, Yury Selivanov <yury@magic.io>

This article explains the new features in Python 3.6, compared to 3.5. Python 3.6 was released on December 23, 2016. See the changelog for a full list of changes.

See also: PEP 494 - Python 3.6 Release Schedule

Summary - Release highlights

New syntax features:

- PEP 498, formatted string literals.
- PEP 515, underscores in numeric literals.
- PEP 526, syntax for variable annotations.
- PEP 525, asynchronous generators.
- PEP 530: asynchronous comprehensions.

New library modules:

- `secrets`: PEP 506 - Adding A Secrets Module To The Standard Library.

CPython implementation improvements:

- The dict type has been reimplemented to use a more compact representation based on a proposal by Raymond Hettinger and similar to the PyPy dict implementation. This resulted in dictionaries using 20% to 25% less memory when compared to Python 3.5.
- Customization of class creation has been simplified with the new protocol.
- The class attribute definition order is now preserved.
- The order of elements in `**kwargs` now corresponds to the order in which keyword arguments were passed to the function.
- DTrace and SystemTap probing support has been added.
- The new PYTHONMALLOC environment variable can now be used to debug the interpreter memory allocation and access errors.

Significant improvements in the standard library:

- The `asyncio` module has received new features, significant usability and performance improvements, and a fair amount of bug fixes. Starting with Python 3.6 the `asyncio` module is no longer provisional and its API is considered stable.
- A new file system path protocol has been implemented to support path-like objects. All standard library functions operating on paths have been updated to work with the new protocol.
- The `datetime` module has gained support for Local Time Disambiguation.
- The `typing` module received a number of improvements.

- The `tracemalloc` module has been significantly reworked and is now used to provide better output for `ResourceWarning` as well as provide better diagnostics for memory allocation errors. See the PYTHONMALLOC section for more information.

Security improvements:

- The new `secrets` module has been added to simplify the generation of cryptographically strong pseudo-random numbers suitable for managing secrets such as account authentication, tokens, and similar.
- On Linux, `os.urandom()` now blocks until the system urandom entropy pool is initialized to increase the security. See the **PEP 524** for the rationale.
- The `hashlib` and `ssl` modules now support OpenSSL 1.1.0.
- The default settings and feature set of the `ssl` module have been improved.
- The `hashlib` module received support for the BLAKE2, SHA-3 and SHAKE hash algorithms and the `scrypt()` key derivation function.

Windows improvements:

- PEP 528 and PEP 529, Windows filesystem and console encoding changed to UTF-8.
- The `py.exe` launcher, when used interactively, no longer prefers Python 2 over Python 3 when the user doesn't specify a version (via command line arguments or a config file). Handling of shebang lines remains unchanged - "python" refers to Python 2 in that case.
- `python.exe` and `pythonw.exe` have been marked as long-path aware, which means that the 260 character path limit may no longer apply. See removing the MAX_PATH limitation for details.
- A `._pth` file can be added to force isolated mode and fully specify all search paths to avoid registry and environment lookup. See the documentation for more information.
- A `python36.zip` file now works as a landmark to infer `PYTHONHOME`. See the documentation for more information.

New Features

PEP 498: Formatted string literals

PEP 498 introduces a new kind of string literals: *f-strings*, or formatted string literals.

Formatted string literals are prefixed with `'f'` and are similar to the format strings accepted by `str.format()`. They contain replacement fields surrounded by curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the `format()` protocol:

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> width = 10
>>> precision = 4
```

```
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
```

See also:

PEP 498 - Literal String Interpolation.
PEP written and implemented by Eric V. Smith.

Feature documentation.

PEP 526: Syntax for variable annotations

PEP 484 introduced the standard for type annotations of function parameters, a.k.a. type hints. This PEP adds syntax to Python for annotating the types of variables including class variables and instance variables:

```
primes: List[int] = []

captain: str  # Note: no initial value!

class Starship:
    stats: Dict[str, int] = {}
```

Just as for function annotations, the Python interpreter does not attach any particular meaning to variable annotations and only stores them in the `__annotations__` attribute of a class or module.

In contrast to variable declarations in statically typed languages, the goal of annotation syntax is to provide an easy way to specify structured type metadata for third party tools and libraries via the abstract syntax tree and the `__annotations__` attribute.

See also:

PEP 526 - Syntax for variable annotations.
PEP written by Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, and Guido van Rossum. Implemented by Ivan Levkivskyi.

Tools that use or will use the new syntax: mypy, pytype, PyCharm, etc.

PEP 515: Underscores in Numeric Literals

PEP 515 adds the ability to use underscores in numeric literals for improved readability. For example:

```
>>> 1_000_000_000_000_000
1000000000000000
```

```
>>> 0x_FF_FF_FF_FF
4294967295
```

Single underscores are allowed between digits and after any base specifier. Leading, trailing, or multiple underscores in a row are not allowed.

The string formatting language also now has support for the `'_'` option to signal the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type `'d'`. For integer presentation types `'b'`, `'o'`, `'x'`, and `'X'`, underscores will be inserted every 4 digits:

```
>>> '{:_}'.format(1000000)
'1_000_000'
>>> '{:_x}'.format(0xFFFFFFFF)
'ffff_ffff'
```

See also:

PEP 515 - **Underscores in Numeric Literals**
PEP written by Georg Brandl and Serhiy Storchaka.

PEP 525: Asynchronous Generators

PEP 492 introduced support for native coroutines and `async` / `await` syntax to Python 3.5. A notable limitation of the Python 3.5 implementation is that it was not possible to use `await` and `yield` in the same function body. In Python 3.6 this restriction has been lifted, making it possible to define *asynchronous generators*:

```
async def ticker(delay, to):
    """Yield numbers from 0 to *to* every *delay* seconds."""
    for i in range(to):
        yield i
        await asyncio.sleep(delay)
```

The new syntax allows for faster and more concise code.

See also:

PEP 525 - **Asynchronous Generators**
PEP written and implemented by Yury Selivanov.

PEP 530: Asynchronous Comprehensions

PEP 530 adds support for using `async for` in list, set, dict comprehensions and generator expressions:

```
result = [i async for i in aiter() if i % 2]
```

Additionally, `await` expressions are supported in all kinds of comprehensions:

```
result = [await fun() for fun in funcs if await condition()]
```

See also:

PEP 530 - **Asynchronous Comprehensions**
PEP written and implemented by Yury Selivanov.

PEP 487: Simpler customization of class creation

It is now possible to customize subclass creation without using a metaclass. The new `__init_subclass__` classmethod will be called on the base class whenever a new subclass is created:

```
class PluginBase:
    subclasses = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.subclasses.append(cls)

class Plugin1(PluginBase):
    pass

class Plugin2(PluginBase):
    pass
```

In order to allow zero-argument `super()` calls to work correctly from `__init_subclass__()` implementations, custom metaclasses must ensure that the new `__classcell__` namespace entry is propagated to `type.__new__` (as described in Creating the class object).

See also:

PEP 487 - **Simpler customization of class creation**
PEP written and implemented by Martin Teichmann.

Feature documentation

PEP 487: Descriptor Protocol Enhancements

PEP 487 extends the descriptor protocol to include the new optional `__set_name__()` method. Whenever a new class is defined, the new method will be called on all descriptors included in the definition, providing them with a reference to the class being defined and the name given to the descriptor within the class namespace. In other words, instances of descriptors can now know the attribute name of the descriptor in the owner class:

```
class IntField:
    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise ValueError(f'expecting integer in {self.name}')
        instance.__dict__[self.name] = value

    # this is the new initializer:
    def __set_name__(self, owner, name):
        self.name = name

class Model:
    int_field = IntField()
```

See also:

PEP 487 - **Simpler customization of class creation**
PEP written and implemented by Martin Teichmann.

Feature documentation

PEP 519: Adding a file system path protocol

File system paths have historically been represented as `str` or `bytes` objects. This has led to people who write code which operate on file system paths to assume that such objects are only one of those two types (an `int` representing a file descriptor does not count as that is not a file path). Unfortunately that assumption prevents alternative object representations of file system paths like `pathlib` from working with pre-existing code, including Python's standard library.

To fix this situation, a new interface represented by `os.PathLike` has been defined. By implementing the `__fspath__()` method, an object signals that it represents a path. An object can then provide a low-level representation of a file system path as a `str` or `bytes` object. This means an object is considered path-like if it implements `os.PathLike` or is a `str` or `bytes` object which represents a file system path. Code can use `os.fspath()`, `os.fsdecode()`, or `os.fsencode()` to explicitly get a `str` and/or `bytes` representation of a path-like object.

The built-in `open()` function has been updated to accept `os.PathLike` objects, as have all relevant functions in the `os` and `os.path` modules, and most other functions and classes in the standard library. The `os.DirEntry` class and relevant classes in `pathlib` have also been updated to implement `os.PathLike`.

The hope is that updating the fundamental functions for operating on file system paths will lead to third-party code to implicitly support all path-like objects without any code changes, or at least very minimal ones (e.g. calling `os.fspath()` at the beginning of code before operating on a path-like object).

Here are some examples of how the new interface allows for `pathlib.Path` to be used more easily and transparently with pre-existing code:

```
>>> import pathlib
>>> with open(pathlib.Path("README")) as f:
...     contents = f.read()
...
>>> import os.path
>>> os.path.splitext(pathlib.Path("some_file.txt"))
('some_file', '.txt')
>>> os.path.join("/a/b", pathlib.Path("c"))
'/a/b/c'
>>> import os
>>> os.fspath(pathlib.Path("some_file.txt"))
'some_file.txt'
```

(Implemented by Brett Cannon, Ethan Furman, Dusty Phillips, and Jelle Zijlstra.)

See also:

PEP 519 - **Adding a file system path protocol**
PEP written by Brett Cannon and Koos Zevenhoven.

PEP 495: Local Time Disambiguation

In most world locations, there have been and will be times when local clocks are moved back. In those times, intervals are introduced in which local clocks show the same time twice in the same day. In these situations, the information displayed on a local clock (or stored in a Python datetime instance) is insufficient to identify a particular moment in time.

PEP 495 adds the new *fold* attribute to instances of `datetime.datetime` and `datetime.time` classes to differentiate between two moments in time for which local times are the same:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

The values of the `fold` attribute have the value `0` for all instances except those that represent the second (chronologically) moment in time in an ambiguous case.

See also:

PEP 495 - Local Time Disambiguation
PEP written by Alexander Belopolsky and Tim Peters, implementation by Alexander Belopolsky.

PEP 529: Change Windows filesystem encoding to UTF-8

Representing filesystem paths is best performed with str (Unicode) rather than bytes. However, there are some situations where using bytes is sufficient and correct.

Prior to Python 3.6, data loss could result when using bytes paths on Windows. With this change, using bytes to represent paths is now supported on Windows, provided those bytes are encoded with the encoding returned by `sys.getfilesystemencoding()`, which now defaults to `'utf-8'`.

Applications that do not use str to represent paths should use `os.fsencode()` and `os.fsdecode()` to ensure their bytes are correctly encoded. To revert to the previous behaviour, set `PYTHONLEGACYWINDOWSFSENCODING` or call `sys._enablelegacywindowsfsencoding()`.

See PEP 529 for more information and discussion of code modifications that may be required.

PEP 528: Change Windows console encoding to UTF-8

The default console on Windows will now accept all Unicode characters and provide correctly read str objects to Python code. `sys.stdin`, `sys.stdout` and `sys.stderr` now default to utf-8 encoding.

This change only applies when using an interactive console, and not when redirecting files or pipes. To revert to the previous behaviour for interactive console use, set `PYTHONLEGACYWINDOWSSTDIO`.

See also:

PEP 528 - Change Windows console encoding to UTF-8
PEP written and implemented by Steve Dower.

PEP 520: Preserving Class Attribute Definition Order

Attributes in a class definition body have a natural ordering: the same order in which the names appear in the source. This order is now preserved in the new class's `__dict__` attribute.

Also, the effective default class *execution* namespace (returned from type.__prepare__()) is now an insertion-order-preserving mapping.

See also:

PEP 520 - **Preserving Class Attribute Definition Order**
PEP written and implemented by Eric Snow.

PEP 468: Preserving Keyword Argument Order

`**kwargs` in a function signature is now guaranteed to be an insertion-order-preserving mapping.

See also:

PEP 468 - **Preserving Keyword Argument Order**
PEP written and implemented by Eric Snow.

New dict implementation

The dict type now uses a "compact" representation based on a proposal by Raymond Hettinger which was first implemented by PyPy. The memory usage of the new `dict()` is between 20% and 25% smaller compared to Python 3.5.

The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new dict implementation in the language for a few releases before changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5).

(Contributed by INADA Naoki in bpo-27350. Idea originally suggested by Raymond Hettinger.)

PEP 523: Adding a frame evaluation API to CPython

While Python provides extensive support to customize how code executes, one place it has not done so is in the evaluation of frame objects. If you wanted some way to intercept frame evaluation in Python there really wasn't any way without directly manipulating function pointers for defined functions.

PEP 523 changes this by providing an API to make frame evaluation pluggable at the C level. This will allow for tools such as debuggers and JITs to intercept frame evaluation before the execution of Python code begins. This enables the use of alternative evaluation implementations for Python code, tracking frame evaluation, etc.

This API is not part of the limited C API and is marked as private to signal that usage of this API is expected to be limited and only applicable to very select, low-level use-cases. Semantics of the API will change with Python as necessary.

See also:

PEP 523 - Adding a frame evaluation API to CPython
PEP written by Brett Cannon and Dino Viehland.

PYTHONMALLOC environment variable

The new `PYTHONMALLOC` environment variable allows setting the Python memory allocators and installing debug hooks.

It is now possible to install debug hooks on Python memory allocators on Python compiled in release mode using `PYTHONMALLOC=debug`. Effects of debug hooks:

- Newly allocated memory is filled with the byte `0xCB`
- Freed memory is filled with the byte `0xDB`
- Detect violations of the Python memory allocator API. For example, `PyObject_Free()` called on a memory block allocated by `PyMem_Malloc()`.
- Detect writes before the start of a buffer (buffer underflows)
- Detect writes after the end of a buffer (buffer overflows)
- Check that the GIL is held when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called.

Checking if the GIL is held is also a new feature of Python 3.6.

See the `PyMem_SetupDebugHooks()` function for debug hooks on Python memory allocators.

It is now also possible to force the usage of the `malloc()` allocator of the C library for all Python memory allocations using `PYTHONMALLOC=malloc`. This is helpful when using external memory debuggers like Valgrind on a Python compiled in release mode.

On error, the debug hooks on Python memory allocators now use the `tracemalloc` module to get the traceback where a memory block was allocated.

Example of fatal error on buffer overflow using `python3.6 -X tracemalloc=5` (store 5 frames in traces):

```
Debug memory block at address p=0x7fbcd41666f8: API 'o'
    4 bytes originally requested
    The 7 pad bytes at p-7 are FORBIDDENBYTE, as expected.
    The 8 pad bytes at tail=0x7fbcd41666fc are not all FORBIDDENBYTE (0:
        at tail+0: 0x02 *** OUCH
        at tail+1: 0xfb
        at tail+2: 0xfb
        at tail+3: 0xfb
        at tail+4: 0xfb
        at tail+5: 0xfb
        at tail+6: 0xfb
        at tail+7: 0xfb
```