The Latest Features of Python 3.12

Dmitrii Ivashchenko

Lead Software Engineer

The Latest Features of Python 3.12

⚙️ Typing

🛠️ Syntax

⏱️ Performance

🐞 Debugging

Typing Improvements

- Generics with New Syntax
- TypedDict
- Override Decorator
- Advanced Type Parameters
- Type Aliases
- Lazy Evaluation and Scope Rules

Generics with New Syntax

Before

```python
from typing import TypeVar, Iterable,
Generic

T = TypeVar('T')

def max(args: Iterable[T]) -> T:
   pass

class List(Generic[T]):
   def __getitem__(self, index: int) -> T:
       pass
```

Python 3.12

```python
def max[T](args: Iterable[T]) -> T:
    pass

class List[T]:
    def __getitem__(self, index: int) -> T:
        pass
```

TypedDict

Before

```python
def foo(**kwargs: int):
    pass

def foo(**kwargs: str):
    pass


from typing import Union

def foo(**kwargs: dict[str, Union[str, int]]):
    pass
```

Python 3.12

```python
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

def foo(**kwargs: Unpack[Movie]):
    pass
```

Override Decorator

```python
from typing import override

class Base:
    def get_color(self) -> str:
        return "blue"

class GoodChild(Base):
    @override  # Correctly overrides Base.get_color
    def get_color(self) -> str:
        return "yellow"

class BadChild(Base):
    @override  # Error: Method name mismatch, does not override
    def get_colour(self) -> str:
        return "red"
```

Advanced Type Parameters

```
# ParamSpec
type IntFunc[**P] = Callable[P, int]

# TypeVarTuple
type LabeledTuple[*Ts] = tuple[str, *Ts]

# TypeVar with bound
type HashableSequence[T: Hashable] = Sequence[T]

# TypeVar with constraints
type IntOrStrSequence[T: (int, str)] = Sequence[T]
```

Type Aliases

Before

```python
from typing import Tuple, TypeVar

PointOld = Tuple[float, float]

T = TypeVar('T')
PointGenericOld = Tuple[T, T]
```

Python 3.12

```python
type Point = tuple[float, float]

type Point[T] = tuple[T, T]
```

Lazy Evaluation

- Lazy evaluation for type aliases and type variable bounds/constraints.
- Evaluation occurs only when necessary for attribute access.
- Enables mutually recursive type aliases and complex type constructs.

```python
# Python 3.12 Example
type Alias = 1/0  # Lazily evaluated

# Accessing __value__ triggers evaluation
try:
   Alias.__value__
except ZeroDivisionError:
   print("ZeroDivisionError caught!")
```

Syntax Improvements

- Reusing Quotes Within f-strings
- Arbitrary Nesting of f-strings
- Multi-line Expressions and Comments in f-strings
- Backslashes and Unicode Characters in f-strings
- Improved Error Messaging for f-strings
- itertools.batched

Reusing Quotes Within f-strings

Python 3.12 now allows the same quotes to be reused inside f-strings, enabling more straightforward and intuitive string formatting.

```
books = [
    'Beyond Good and Evil',
    'Thus Spoke Zarathustra',
    'Meditations'
]

reading_list = f"This is the reading list: {", ".join(books)}"

print(reading_list)
```

Arbitrary Nesting of f-strings

The new update allows for arbitrary nesting of f-strings, making complex string constructions more manageable.

```python
# Before
nested_f_string = f"""{f'''{f'{f"{1+1}"}'}'''}"""
print(nested_f_string)


# Python 3.12
nested_f_string = f"{f'{f'{f'{f'{1+1}'}'}'}'}"
print(nested_f_string)
```

Multi-line Expressions and Comments

Python 3.12 supports multi-line expressions and inline comments within f-strings, enhancing readability and maintainability.

```python
# Python 3.12

movie_list = f"""This is the movie list: {', '.join([
    'Inception',                # Mind-bending plot
    'Interstellar',             # Space exploration
    'The Matrix'                # Virtual reality
])}"""

print(movie_list)
```

Backslashes and Unicode Characters

The inclusion of backslashes and Unicode escape sequences in f-string expressions is now possible, broadening the scope for string formatting.

```
# Python 3.12

print(f"Separated by newlines: {'\\n'.join(songs)}")

print(f"Joined with a unicode character: {'\\N{BLACK HEART SUIT}'.join(songs)}")
```

Improved Error Messaging for f-strings

Enhanced parsing of f-strings leads to more precise error messages, aiding in quicker debugging and development.

```
>>> my_string = f"{x z y}" + f"{1 + 1}"
  File "<stdin>", line 1
    (x z y)
     ^^^
SyntaxError: f-string: invalid syntax. Perhaps you forgot a comma?


>>> my_string = f"{x z y}" + f"{1 + 1}"
  File "<stdin>", line 1
    my_string = f"{x z y}" + f"{1 + 1}"
                   ^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

itertools.batched

- New utility function in Python 3.12: itertools.batched.
- Splits an iterable into fixed-size batches for efficient processing.
- Ideal for handling large datasets or streaming data in chunks.

```python
from itertools import batched

# Using itertools.batched in Python 3.12
for batch in batched('ABCDEFG', 3):
   print(batch)

# Output:
# ('A', 'B', 'C')
# ('D', 'E', 'F')
# ('G',)
```

Performance

- Asyncio Performance Enhancements
- Boost in Inspect and Typing Modules
- Immortal Objects
- Unique Per Interpreter GIL
- Comprehension Inlining
- Other Enhancements

Asyncio Performance Enhancements

- 75% speed up in benchmarks
- Improved socket write performance
- Faster **asyncio.Task** creation
- C implementation of **asyncio.current_task()**

Boost in Inspect and Typing Modules

- **inspect.getattr_static()** 2x-6x faster
- **isinstance()** checks 2x-20x faster against protocols
- Slower **isinstance()** checks for protocols with 14+ members

Unique Per Interpreter GIL

- PEP 684 introduces a unique Global Interpreter Lock (GIL) for each sub-interpreter.
- Enables true parallel execution of Python code across multiple CPU cores.
- Available through the C-API in Python 3.12, with a Python API expected in version 3.13.
- Enhances multi-threaded performance by isolating sub-interpreters, each with its own GIL.

```c
// Python 3.12: Creating a new interpreter with its own GIL
PyInterpreterConfig config = {
   .check_multi_interp_extensions = 1,
   .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
   // Handle error
}
// New interpreter with its own GIL is now active
```

Comprehension Inlining

- Inlining for dictionary, list, and set comprehensions
- Up to 2x faster execution of comprehensions.
- Maintains variable isolation within comprehensions.

```python
# Before Python 3.12:
# Each execution created a new, single-use function object
result = [x**2 for x in range(10)]

# Python 3.12: Inlined comprehension example
# Faster execution, no separate function object
result = [x**2 for x in range(10)]

# Iterating over locals() workaround
keys = list(locals())
result = [k for k in keys]
```

Immortal objects

Introduce Immortal Objects, which allows objects to bypass reference counts, and related changes to the C-API.

```python
import sys

def main():
    count = sys.getrefcount(None)
    print(f"0b{count:b}")

if __name__ == "__main__":
    main()

# Output: 0b11111111111111111111111111111111
```

Buffer protocol

```python
import contextlib
import inspect


class MyBuffer:
    def __init__(self, data: str):
        self.data = bytearray(data, 'utf-8')
        self.view = None

    def __buffer__(self, flags: int) -> memoryview:
        if flags != inspect.BufferFlags.FULL_RO:
            raise TypeError("Only BufferFlags.FULL_RO supported")

        if self.view is not None:
            raise RuntimeError("Buffer already in use")

        self.view = memoryview(self.data)
        return self.view

    def __release_buffer__(self, view: memoryview) -> None:
        assert self.view is view  # guaranteed to be true
        self.view.release()
        self.view = None

    def extend(self, additional_data: str) -> None:
        if self.view is not None:
            raise RuntimeError("Cannot extend buffer while in use")
        # extend the buffer
        self.data.extend(bytearray(additional_data, 'utf-8'))
```

```python
buffer = MyBuffer("Hello")
with memoryview(buffer) as view:
    with contextlib.suppress(RuntimeError):
        # raises RuntimeError because the buffer is in use
        buffer.extend(" World")

# okay now because buffer is no longer in use
buffer.extend(" World")

with memoryview(buffer) as view:
    # should output "Hello World"
    print(view.tobytes().decode())
```

Other Enhancements

- Experimental support for BOLT optimizer
- 2-3x faster regex substitution
- Inlined comprehensions for dictionaries, lists, sets
- Enhanced **super()** method calls

Debugging

- Improved NameError Suggestions
- Syntax and ImportError Enhancements
- Low Impact Monitoring
- Tool Identifiers and Monitoring Events
- Advanced Event Monitoring Control
- Callback Functions and Event Handling

Improved NameError Suggestions

- Suggestions for missing standard library imports
- Instance attribute hints in NameError
- Enhanced import statement syntax errors

```
>>> class A:
...     def __init__(self):
...         self.blech = 1
...
...     def foo(self):
...         somethin = blech
...
>>> A().foo()
Traceback (most recent call last):
  File "<stdin>", line 1
    somethin = blech
               ^^^^^
NameError: name 'blech' is not defined. Did you mean: 'self.blech'?
```

Syntax and ImportError Enhancements

- Clearer SyntaxError for incorrect import syntax
- ImportError suggestions based on module contents

```
>>> import a.y.z from b.y.z
Traceback (most recent call last):
  File "<stdin>", line 1
    import a.y.z from b.y.z
    ^^^^^^^^^^^^^^^^^^^^^^^
 SyntaxError: Did you mean to use 'from ... import ...' instead?


>>> from collections import chainmap
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
 ImportError: cannot import name 'chainmap' from 'collections'. Did you mean: 'ChainMap'
```

Low Impact Monitoring

- New API for monitoring CPython execution events.
- Designed for profilers, debuggers, and monitoring tools.
- Supports a wide range of events with minimal overhead.
- Allows for near-zero overhead in debuggers and coverage tools.

```python
# Python 3.12
import sys

def my_callback(code, line_number):
   print(f"Executing line {line_number} in {code.co_filename}")

sys.monitoring.register_callback(sys.monitoring.DEBUGGER_ID,
sys.monitoring.events.LINE, my_callback)
sys.monitoring.set_events(sys.monitoring.DEBUGGER_ID,
sys.monitoring.events.LINE)
```

Resources & Links

- What's New In Python 3.12
- Python 3.12.2 Changelog
- Python 3.12 is here by James Murphy
- Python 3.12: New Features for You to Try by Geir Arne Hjelle