Python news

OXiane

INSTITUT

make it clever

OXiane INSTITUT

Summary

- New Features in Python 3.9
  - New operators and basic methods
  - Types and annotations
  - Standard library improvements
  - Asynchronous programming
  - Other improvements
- New Features in Python 3.10
  - New keywords and syntax
  - Types and annotations
  - Asynchronous programming
  - Error handling improvements
- New Features in Python 3.11
  - Syntax improvements
  - Performance improvements
  - Exception handling
  - Asynchronous programming
- New Features in Python 3.12
  - Standard library improvements
  - New functions and enhancements
  - Asynchronous programming

Python 3.9

New operators and basic methods

1. The Union (`|`) Operator for Dictionaries

Before Python 3.9

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict_combined_before = dict1.copy()
dict_combined_before.update(dict2)
# Result: {'a': 1, 'b': 3, 'c': 4}
```

With Python 3.9

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict_combined_39 = dict1 | dict2
# Result: {'a': 1, 'b': 3, 'c': 4}
```

2. The Update (`|=`) Operator for Dictionaries

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict1.update(dict2)
# Result: {'a': 1, 'b': 3, 'c': 4}
```

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict1 |= dict2
# Result: {'a': 1, 'b': 3, 'c': 4}
```

New operators and basic methods

3. New String Methods: `removeprefix()` **and** `removesuffix()`

Before Python 3.9

```
string = "HelloWorld"
if string.startswith("Hello"):
    string_without_prefix = string[len("Hello"):]
# Result: 'World'

if string.endswith("World"):
    string_without_suffix = string[:-len("World")]
# Result: 'Hello'
```

In Python 3.9

```
string = "HelloWorld"
string_without_prefix = string.removeprefix("Hello")
# Result: 'World'

string_without_suffix = string.removesuffix("World")
# Result: 'Hello'
```

Types and annotations

1. Type Hinting for Built-in Collection Types

Before Python 3.9

```
from typing import List, Dict

def process_data(data: List[int]) -> Dict[str, int]:
    return {'sum': sum(data)}
```

With Python 3.9

```
def process_data(data: list[int]) -> dict[str, int]:
    return {'sum': sum(data)}
```

2. New `Annotated` Type

Before Python 3.9

```
def positive_number(number: int) -> None:
    """
    This function expects a positive integer.
    """
    print(number)
```

In Python 3.9

```
from typing import Annotated

def positive_number(number: Annotated[int, "positive"]) -> None:
    print(number)
```

Standard library improvements

1. New `zoneinfo` Module

Example before Python 3.9

```
import pytz
from datetime import datetime

timezone = pytz.timezone('America/New_York')
new_york_time = datetime.now(timezone)
print(new_york_time)
```

In Python 3.9

```
from zoneinfo import ZoneInfo
from datetime import datetime

new_york_time = datetime.now(ZoneInfo('America/New_York'))
print(new_york_time)
```

2. Updates to `math` and `statistics` Modules

```
import math

# Get the next floating-point value after 1.0
next_value = math.nextafter(1.0, 2.0)
print(next_value)

# Get the unit in the last place of 1.0
ulp_value = math.ulp(1.0)
print(ulp_value)
```

```
import statistics

data = [1, 2, 2, 3, 3, 4, 4, 4]
modes = statistics.multimode(data)
print(modes)  # Output: [2, 3, 4]
```

Asynchronous programming

`asyncio.to_thread` Function

The `asyncio.to_thread` function was introduced in Python 3.9 to facilitate running IO-bound functions in a separate thread. This makes it easier to integrate synchronous code with asynchronous code, enhancing performance and readability.

Before Python 3.9

```
import asyncio
import time
from concurrent.futures import ThreadPoolExecutor

def blocking_io():
    print("start blocking_io")
    time.sleep(2)  # Simulating a blocking I/O operation
    print("end blocking_io")

async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        await loop.run_in_executor(pool, blocking_io)

asyncio.run(main())
```

With Python 3.9

```
import asyncio
import time

def blocking_io():
    print("start blocking_io")
    time.sleep(2)  # Simulating a blocking I/O operation
    print("end blocking_io")

async def main():
    await asyncio.to_thread(blocking_io)

asyncio.run(main())
```

Asynchronous programming

Key Differences and Benefits:

- **Simplicity**: `asyncio.to_thread` eliminates the need for manually creating and managing a `ThreadPoolExecutor`.
- **Readability**: The code is more concise and easier to understand.
- **Efficiency**: It integrates more seamlessly with the asyncio event loop, reducing boilerplate code and potential for errors.

Other improvements

1. New PEG Parser

Python 3.9 introduced a new PEG (Parsing Expression Grammar) parser, which replaced the previous LL(1) parser. This change allows for more flexible and powerful syntax parsing. The PEG parser is more capable of handling complex syntax rules and can improve the performance and maintainability of the language.

The benefits of the PEG parser might not be immediately visible in simple scripts, but it lays the groundwork for more advanced features and better language consistency in future versions of Python.

2. Parenthesized Context Managers

Before Python 3.9

```python
try:
    with open('file1.txt') as f1:
        with open('file2.txt') as f2:
            pass
except FileNotFoundError:
    pass
```

In Python 3.9

```python
try:
    with (open('file1.txt') as f1, open('file2.txt') as f2):
        pass
except FileNotFoundError:
    pass
```

Python 3.10

OXiane INSTITUT

Structural Pattern Matching

Python 3.10 introduces one of the most significant syntax additions in recent years—structural pattern matching, akin to switch-case statements found in other programming languages.

```python
def describe(value):
    match value:
        case {'type': 'fruit', 'name': str(name)}:
            return f"This is a {name}."
        case {'type': 'veggie', 'name': str(name)}:
            return f"This is a {name}, which is a vegetable."
        case _:
            return "Unknown item"
```

- This feature is perfect for handling different types of data with specific attributes, enhancing code readability and maintainability.

Structural Pattern Matching

- Example: Matching User Profiles

```python
def user_greeting(profile):
    match profile:
        case {'name': str(name), 'age': int(age)} if age >= 18:
            return f"Welcome, {name}! You are an adult."
        case {'name': str(name), 'age': int(age)}:
            return f"Hello, {name}! You are under 18."
        case {'name': str(name)}:
            return f"Welcome, {name}! Your age is not specified."
        case _:
            return "Unknown profile format"

# Example usages
print(user_greeting({'name': 'Alice', 'age': 22}))  # Outputs: Welcome, Alice! You are an adult.
print(user_greeting({'name': 'Bob', 'age': 17}))    # Outputs: Hello, Bob! You are under 18.
print(user_greeting({'name': 'Charlie'}))           # Outputs: Welcome, Charlie! Your age is not specified.
print(user_greeting({'username': 'Dave'}))          # Outputs: Unknown profile format
```

Type Union Operator

The new type union operator `|` simplifies the expression of type annotations.

```
def greet(name: str | None) -> str:
    return f"Hello, {name or 'guest'}"
```

- This operator streamlines type hints, making them cleaner and more intuitive, especially in complex functions.

Type Union Operator

Example 2

```python
def process_input(data: int | str) -> str:
    if isinstance(data, int):
        # If it's an integer, return its square as a string
        return f"The square of {data} is {data ** 2}."
    elif isinstance(data, str):
        # If it's a string, return it in uppercase
        return f"You entered the string: {data.upper()}."

# Example usages
print(process_input(10))  # Outputs: The square of 10 is 100.
print(process_input("hello"))  # Outputs: You entered the string: HELLO.
```

`connect_accepted_socket()` method

- Facilitates the integration of already accepted sockets into the asyncio event loop, simplifying the management of asynchronous network operations.

```
import asyncio
import socket
async def handle_connection(reader, writer):
    data = await reader.read(1024)
    # Process data
    writer.close()
async def main():
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.bind(('localhost', 8080))
    server_sock.listen()
    loop = asyncio.get_running_loop()
    while True:
        client_sock, addr = server_sock.accept()
        reader, writer = await asyncio.connect_accepted_socket(loop, client_sock)
        loop.create_task(handle_connection(reader, writer))
asyncio.run(main())
```

Error Handling Improvements

- Error messages now provide clearer guidance on the nature of syntax mistakes, significantly aiding in debugging efforts.
- Developers receive more intuitive feedback on errors, which includes improved messages for unclosed brackets or misplaced operators, thereby reducing confusion and speeding up the debugging process.

Error Handling Improvements

Example 1: Improved Messages for Unclosed Brackets

```python
def calculate(input_list):
    result = [x*(2+x) for x in input_list
print(result)

# Python 3.10 error message:
# SyntaxError: '(' was never closed
```

Example 2: Misplaced Operators

```python
age = 18
if age = 18:
    print("You are eighteen.")

# Python 3.10 error message:
# SyntaxError: invalid syntax. Perhaps you meant '=='?
```

Performance Improvements

1. Optimization of Buffer Operations

Buffer operations are critical in Python, especially for applications that handle large amounts of data or perform I/O operations, such as data processing applications, network servers, or multimedia processing. Python 3.10 improves how buffer operations are handled, primarily through these mechanisms:

- **Reduced Overhead**: Buffer operations have been optimized to reduce the overhead involved in accessing and manipulating buffer data. This includes minimizing the number of temporary objects created during these operations, which in turn reduces the pressure on Python's garbage collector.
- **Enhanced Memory Management**: Changes in memory management related to buffer operations help in more efficient allocation and deallocation of memory. This is particularly beneficial for applications that frequently read and write large blocks of data, as it enhances throughput and reduces latency.

Performance Improvements

2. More Efficient Use of Allocator Caches

Python uses a memory allocation strategy that involves allocator caches. These caches help by reusing memory blocks for objects of similar sizes, which can significantly speed up memory allocation and deallocation cycles:

- **Improved Allocator Efficiency**: Python 3.10 includes enhancements in the way allocator caches are managed. By optimizing the allocator's cache mechanism, Python can manage memory more efficiently, which reduces fragmentation and improves cache hit rates.
- **Adaptive Strategies**: The improvements include more adaptive strategies for managing the allocator caches based on the usage patterns. This means that Python can adjust its memory allocation strategy dynamically based on the current workload, leading to better performance, especially in long-running applications.

The collective impact of these enhancements includes:

- **Faster Execution Times**
- **Reduced Memory Usage**
- **Increased Scalability**

Python 3.11

Performance Improvements

- The main implementation of Python is now on average 25% faster with Python 3.11 than with Python 3.10.
- Speed has improved by up to 60% in some scenarios
- Python is also and especially faster at startup.

With Python 3.6

```
#!/bin/bash
SECONDS=0
for i in {1..250}
do
   /usr/bin/time python3.6 -c "pass"
done
duration=$SECONDS
echo "$(($duration % 60)) seconds elapsed."
```

```
$ speed-test36.sh
0.03 real         0.02 user         0.00 sys
0.03 real         0.02 user         0.00 sys
0.03 real         0.02 user         0.00 sys
...
0.03 real         0.02 user         0.00 sys
0.02 real         0.02 user         0.00 sys
0.03 real         0.02 user         0.00 sys
8 seconds elapsed.
```

With Python 3.11

```
#!/bin/bash
SECONDS=0
for i in {1..250}
do
   /usr/bin/time python3.11 -c "pass"
done
duration=$SECONDS
echo "$(($duration % 60)) seconds elapsed."
```

```
$ speed-test311.sh
0.02 real         0.01 user         0.00 sys
0.01 real         0.01 user         0.00 sys
0.02 real         0.01 user         0.00 sys
...
0.02 real         0.01 user         0.00 sys
0.02 real         0.01 user         0.00 sys
0.02 real         0.01 user         0.00 sys
5 seconds elapsed.
```

Exception notes

- It is now possible to add notes to exceptions with the `add_note` method:

```
import requests

def exception_notes():
    try:
        r = requests.get('http://www.google.comx')
    except requests.exceptions.RequestException as e:
        e.add_note("Couldn't fetch Google...")
        raise

exception_notes()
```

Exception Groups

- Python 3.11 brings the possibility to group exceptions

```python
try:
    raise ExceptionGroup("Exception Group for multiple errors", [
        ValueError("This is a value error"),
        TypeError("This is a type error"),
        KeyError("This is a Key error"),
        AttributeError('This is an Attribute Error'),
        AttributeError('This is another Attribute Error')
    ])
except* AttributeError as err:
    raise err
except* (ValueError, TypeError) as err:
    raise err
except* KeyError as err:
    raise err
```

Exception Groups

Example 2

```python
import asyncio
from concurrent.futures import ThreadPoolExecutor
from exceptions import ExceptionGroup

# Simulated tasks that might fail
async def fetch_data():
    if some_network_condition:
        raise ConnectionError("Failed to connect to the server")
    return "Data"

async def read_file():
    if some_file_condition:
        raise FileNotFoundError("File not found")
    return "File content"

async def process_data():
    if some_processing_condition:
        raise ValueError("Invalid data")
    return "Processed data"
```

Exception Groups

Example 2

```
async def main():
    tasks = [fetch_data(), read_file(), process_data()]
    results = []
    try:
        # Execute all tasks, gather might raise multiple exceptions
        results = await asyncio.gather(*tasks, return_exceptions=True)
        raise ExceptionGroup("Multiple task errors", [result for result in results if isinstance(result, BaseException)])
    except* ConnectionError as err:
        print(f"Network issue: {err}")
    except* FileNotFoundError as err:
        print(f"File issue: {err}")
    except* ValueError as err:
        print(f"Data processing issue: {err}")
    return results

# Running the async main function
asyncio.run(main())
```

Typing 'self'

- It is now possible to indicate that a method returns an instance of the class with the `Self` keyword available in the typing module.

```
from typing import Self

class CustomPath:
    def __init__(self, path: str):
        self.path = path

    # The concat method returns an instance of the class CustomPath
    def concat(self, other: str) -> Self:
        return CustomPath(f'{self.path}/{other}')

    def __str__(self):
        return self.path
```

More Precise Error Messages

- Error messages are now more precise, specifically indicating where an error is located in the traceback.

```python
def example1():
    d = {"uno": [1, [1, 2, 3], 3]}
    print(d["uno"][5][2])

def example2():
    a, b, c, d, e, f = 1, 2, 0, 4, 5, 6
    print(a / b / c / d / e / f)

def example3():
    a = None
    b = ""
    print(a.capitalize() + b.capitalize())
```

TOML Support

- This is now possible with the addition of the `tomllib` library, which allows reading .toml configuration files

config.toml

title = "TOML Example"
[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00
[database]
enabled = true
ports = [8000, 8001, 8002]
data = [["delta", "phi"], [3.14]]
temp_targets = { cpu = 79.5, case = 72.0 }

```python
import tomllib

with open("setup.toml", "rb") as f:
    data = tomllib.load(f)

print(data)
print(data['owner']['name'])
print(data['database']['ports'])
```

AsyncIO Task Groups

- The addition of the `TaskGroup` class allows for the creation of groups of asynchronous tasks

```python
import asyncio
import math

async def t1():
    print(int("hello"))
    await asyncio.sleep(2)

async def t2():
    print(math.sqrt(-10))
    await asyncio.sleep(1)

async def main():
    try:
        async with asyncio.TaskGroup() as tg:
            tg.create_task(t1())
            tg.create_task(t2())
    except* ValueError as e:
        print(e.exceptions)

if __name__ == '__main__':
    asyncio.run(main())
```

Standard Library Enhancements

1. The math module

- New functions added to the math module:

```
import math
print(math.cbrt(27))  # Outputs: 3.0000000000000004
print(math.exp2(6))   # Outputs: 64.0
```

2. Retrieving only folders with pathlib

- The `glob` method of the `Path` class in the `pathlib` module now allows you to directly indicate if you want to retrieve only the folders inside a directory.

```
from pathlib import Path

p = Path("/Users/u/python-311-new-features/standard_lib/paths_tests")
print("Files and folders")
dirs = p.glob("*")
for d in dirs:
    print(d)

print("Folders only")
dirs = p.glob("*/")
for d in dirs:
    print(d)
```

Standard Library Enhancements

StrEnum

It is now possible to use the `auto` function to automatically create string enumerations using the `StrEnum` class:

```
from enum import StrEnum, auto

class Color(StrEnum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()
```

Python 3.12

Syntactic Formalization of f-strings

- Expression components inside f-strings can now be any valid Python expression, including strings using the same quotation marks as the containing f-string, multiline expressions, comments, backslashes, and Unicode escape sequences.

Using the Same Quotation Marks

```
fruits = ['Apples', 'Pears', 'Bananas']
f"Here is your shopping list: {', '.join(fruits)}"
```

```
f"Here is your shopping list: {', '.join([
    'Apples',  # 3 apples
    'Pears',   # 2 pears
    'Bananas'  # 5 bananas
])}"
```

```
songs = ['Take me back to Eden', 'Alkaline', 'Ascensionism']
print(f"This is the playlist: {'\n'.join(songs)}")
# This is the playlist: Take me back to Eden
# Alkaline
# Ascensionism

print(f"This is the playlist: {'\N{BLACK HEART SUIT}'.join(songs)}")
# This is the playlist: Take me back to Eden♥Alkaline♥Ascensionism
```

More Precise Error Messages

- If you use a standard library module that has not been imported, the Python interpreter will explicitly indicate this:

```
>>> sys.version_info
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# NameError: name 'sys' is not defined. Did you forget to import 'sys'?
```

More Precise Error Messages

- If you forget to use 'self' in front of an attribute that exists in a class, Python will indicate that you probably forgot it:

```
class A:
    def __init__(self):
        self.blech = 1

    def foo(self):
        somethin = blech

>>> A().foo()
#   File "<stdin>", line 1
#     somethin = blech
#                ^^^^^
# NameError: name 'blech' is not defined. Did you mean: 'self.blech'?
```

More Precise Error Messages

- The last addition that can be practical for certain names whose exact syntax we forget, Python will be able to suggest elements present in the module in case of an import error:

```
>>> 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'?
```

Improved Type Annotations for **kwargs

- In Python, when you use **kwargs in the definition of a function, it means that the function can accept an indefinite number of arguments in the form of keywords (keywords).
- Currently, when you annotate **kwargs with a type T, it means that all keyword arguments must be of this type. For example, if you write a function like this:

```python
def foo(**kwargs: str) -> None: ...
# This means that all arguments provided to foo must be strings.
```

Improved Type Annotations for **kwargs

- PEP 589 introduced TypedDict, which allows specifying a dictionary whose keys are strings and values can be of different types.

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

```
def foo(**kwargs: Movie) -> None: ...
# This would mean that each argument provided to foo should itself be a dictionary with the keys name and year.
```

```
foo(arg1={"name": "Blade Runner", "year": 1982}, arg2={"name": "Harry Potter", "year": 2011})
```

Improved Type Annotations for **kwargs

- To avoid this confusion, a new approach is proposed: using Unpack.
- Using Unpack allows specifying that the arguments provided directly to the function must match the keys of the TypedDict.

```
def foo(**kwargs: Unpack[Movie]) -> None: ...
# Here, it is expected that foo be called with two keywords, name and year, rather than with a single keyword that would be a dictionary:
foo(name="Blade Runner", year=1982)
```

The override Decorator

- A new decorator typing.override() has been added to the typing module.
- It indicates to "type checkers" that the method is intended to replace a method in a class.

```python
from typing import override

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

class GoodChild(Base):
  @override  # Here, no error, the class correctly overrides the get_color method of Base.
  def get_color(self) -> str:
    return "yellow"

class BadChild(Base):
  @override  # Error: there is a typo in the method name
  def get_colour(self) -> str:
    return "red"
```

A GIL per Interpreter

- The GIL (Global Interpreter Lock) is a lock that the Python interpreter uses to ensure that only one thread runs in the interpreter at a time.
- This is one of the reasons why traditional Python programs do not fully utilize the capabilities of multicore processors for multithreaded execution.
- PEP 684 introduces a major novelty: instead of having a single GIL for the entire interpreter, we can now have a unique GIL for each sub-interpreter.

Faster Comprehensions

- Comprehensions (lists, dictionaries, and sets) are expressions commonly used in Python to generate collections concisely.
- Previously, every time a comprehension was executed, Python created an anonymous function (lambda functions) behind the scenes to carry it out.
- With this new proposal (PEP 709), this step is optimized: comprehensions are now "inlined" (or inlined), which means they are executed directly without creating a temporary function.
- This improves the performance of comprehensions, making them up to twice as fast.

```
squared_numbers = [x**2 for x in range(10)]
```

```
import time

# Measuring performance of a list comprehension
start_time = time.time()
squared_numbers = [x**2 for x in range(10000)]
end_time = time.time()
print("Execution time with traditional comprehension:", end_time - start_time)
```

```
 import time

# Measuring performance of a list comprehension
start_time = time.time()
squared_numbers = [x**2 for x in range(10000)]
end_time = time.time()
print("Execution time with traditional comprehension:", end_time - start_time)
```

Conclusion