

Hi folks, I'm Amethyst, an engineer at Meta working on the Python Language Foundation team. We build infrastructure and tools to support thousands of engineers, data scientists, ML researchers, and anyone else that uses Python to get their work done.



Today I'd like to talk about the newly release Python 3.13, and why I think it's kind of a big deal. It released just last week, and there's a bunch of little features, and one big feature, that can make your everyday Python experience that much better.

1. Free Threading
2. Improved REPL
3. Color!
4. @deprecated

 amyreese  amethyst@n7.gg

Of course the biggest headlining feature, that I'm sure many of you have heard about, is Free Threading, aka NoGIL, that can greatly improve the performance of multithreaded Python applications.

Beyond that, there's a new and improved interactive interpreter, or REPL; colored output in multiple places; and finally a first party decorator for marking functions and classes as deprecated.

@deprecated(...)

 amyreese  amethyst @ n7.gg

Let's work from the bottom up, and start with the new deprecation mechanism. Deprecations have long been a thing in upstream Python, and now there's finally an official way to mark your own classes and functions as deprecated. And it's not just for documentation — using deprecated elements will emit a warning with a mini traceback, just like in the standard library.

```
from warnings import deprecated

@deprecated("gimme one reason")
def stay():
    print("no thanks")

$ python3.13t -q
>>> from show import stay
>>> stay()
<python-input-1>:1: DeprecationWarning: gimme one reason
    stay()
no thanks
```

 amyreese  amethyst@n7.gg

So for example let's say we have an old or outdated function, and we want to discourage users from calling it. We can mark it as deprecated, and provide a warning message to users. Ideally we would put something helpful here, like what other function or system to use instead.

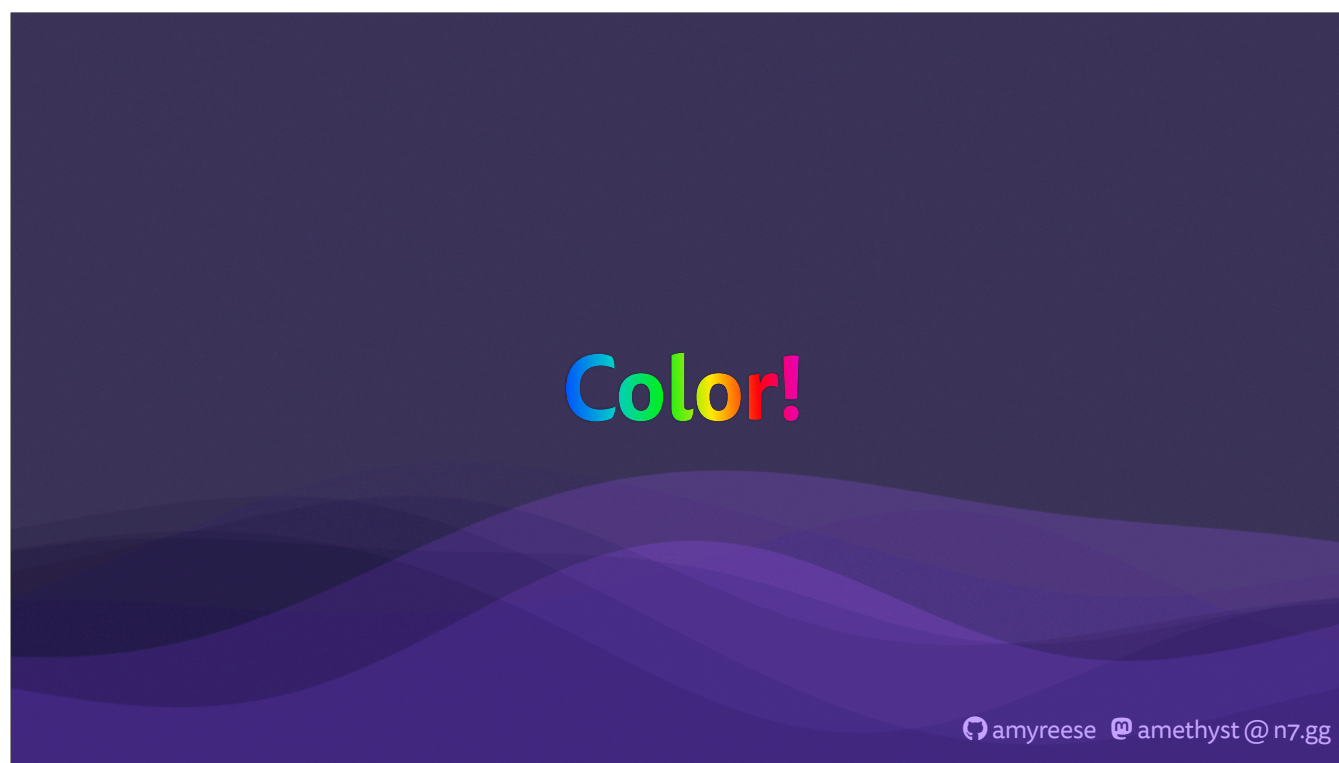
Then when we try to call that function, we see the Python runtime emits the DeprecationWarning, but the function still runs correctly.

```
@deprecated("The class is in another castle")
class Something:
    ...

$ python3.13t -q
>>> from show import Something
>>> obj = Something()
<python-input-1>:1: DeprecationWarning: The class is in another castle
    obj = Something()
```

 amyreese  amethyst@n7.gg

Similarly, when we deprecate a class, then every time it gets instantiated or subclassed, we see that deprecation warning emitted. But that all sounds very business-y, so let's look at something more fun...



Color! That's right, Python 3.13 finally has colorized output in the CLI.



**tracebacks
doctests
interactive interpreter**

 amyreese  amethyst @ n7.gg

More specifically, it now uses color to help distinguish elements in exception tracebacks, error results from doctests, and in the interactive interpreter. You've already seen a bit of the color in the REPL, so let's take a closer look at these colors for tracebacks and doctests.


```
def abort():
    raise RuntimeError("ohno")

$ python3.13t -q
>>> from show import abort
>>> abort()
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    abort()
    ~~~~~^
  File "/Users/amethyst/scratch/py313/show.py", line 10, in abort
    raise RuntimeError("ohno")
RuntimeError: ohno
>>>
```

 amyreese  amethyst@n7.gg

So here we just define a simple function that raises an exception. When we run that function, we can see the colored traceback in our terminal, which helpfully highlights elements including the last function call before the error. And the colors work both in the REPL and when running a Python program directly from the CLI.

```
def half(value: int) -> int:
    """
    >>> half(4)
    2
    >>> half(37)
    18
    >>> half(939)
    42
    """
    return value // 2
```

 amyreese  amethyst@n7.gg

Now let's look at an example of doctests, where you can define test cases directly in your function's docstring, in a REPL style format of expressions and their expected results.

Notice in this case we have three tests for this function, with two truths and a lie.

```
$ python3.13t -m doctest show.py
*****
File "/Users/amethyst/scratch/py313/show.py", line 19, in show.half
Failed example:
    half(939)
Expected:
    42
Got:
    469
*****
1 item had failures:
  1 of 3 in show.half
***Test Failed*** 1 failure.
> [1]
```

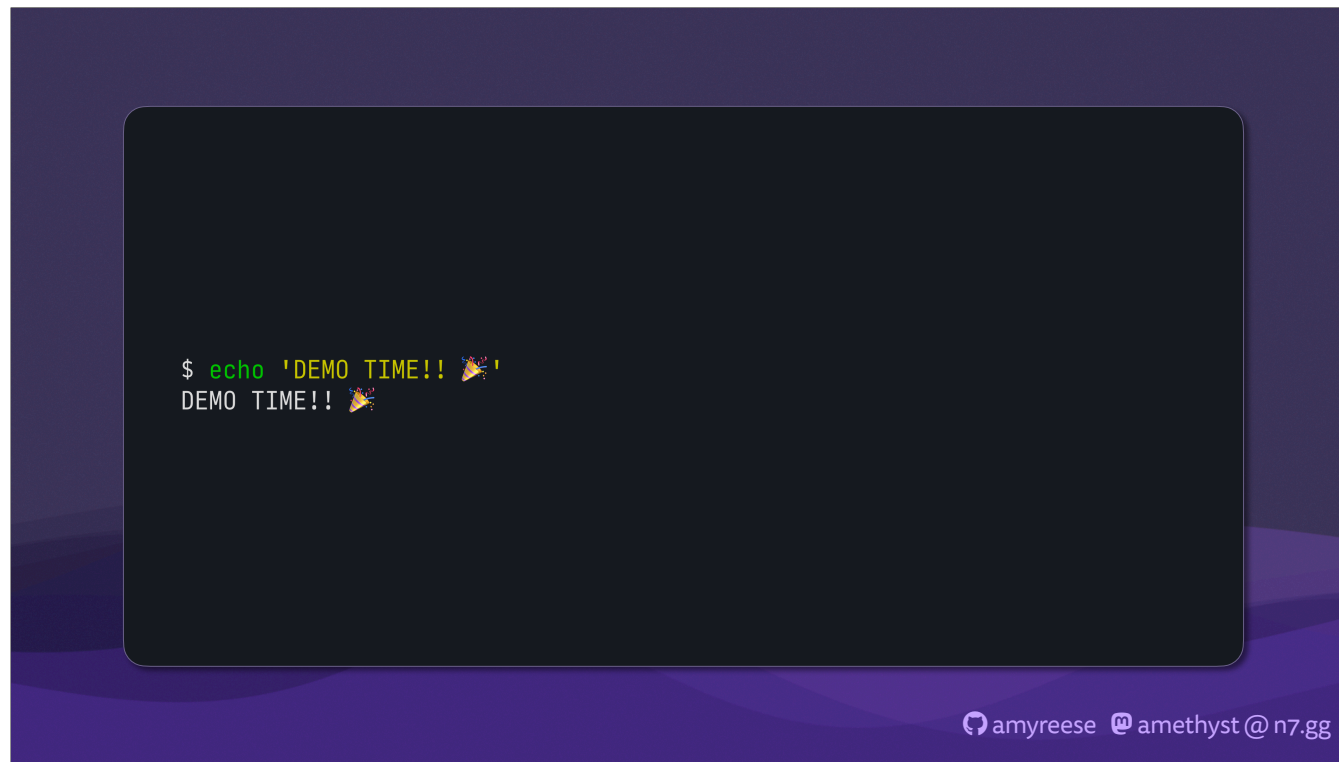
 amyreese  amethyst@n7.gg

When we run our doctests, we now see some colored output, better showing failed test cases and the final aggregate test results.

Improved REPL

 amyreese  amethyst@n7.gg

Now most of these new colorizations are also part of the new and improved REPL, or interactive interpreter. The new REPL is improved in a number of ways, and sort of bridges the gulf of functionality between the old REPL and alternatives like ipython. New features include tab-completion for variable names and object properties, better history and multiline input handling, and more.

A terminal window with a dark background and a purple gradient border. The prompt is a green dollar sign. The command is 'echo 'DEMO TIME!! 🚀''. The output is 'DEMO TIME!! 🚀'.

```
$ echo 'DEMO TIME!! 🚀'  
DEMO TIME!! 🚀
```

amyreese @amethyst@n7.gg

Since some of that isn't going to be obvious in screenshots, let's go jump over to a real REPL and give a quick live demo ...

Personally, I still prefer ipython as a more complete REPL, but this goes a long way to making the default REPL usable when that's the only option available.



Finally we get to the big prize of this release, free threading or NoGIL. This is an experimental feature for 3.13 that disables the GIL, or the global interpreter lock.

global interpreter lock

 amyreese  amethyst@n7.gg

For those unfamiliar, the standard Python runtime allows developers to write code without worrying about locks or synchronization on individual objects in memory, even when spawning hundreds or thousands of threads. It accomplishes this by using a single global lock in the Python runtime, and prevents more than one thread from executing Python code or accessing Python objects at one time. This makes individual threads faster by reducing the amount of time spent dealing with locks, at the cost of parallelism — you could have dozens of CPU cores and hundreds of threads, but with the GIL, Python itself would only ever be able to run a single thread at any given time, using only a single core.

~~concurrency~~ parallelism

 amyreese  amethyst@n7.gg

By contrast, free threading disables the GIL, and replaces it with individual locks on Python objects. This makes an individual Python thread run slightly slower, but it allows Python to execute multiple threads simultaneously, potentially saturating dozens or hundreds of CPU cores just with multithreading, without needing multiprocessing or external worker queues. This transforms multithreaded code from concurrency to true parallelism, unlocking workloads in ways that asyncio and multiprocessing can't compete.



So how do we take advantage of free threading? Well for now, it first starts with installing a separate build of Python that has been specifically compiled with the GIL disabled. These free threading builds of Python can be installed side-by-side with the standard runtimes, and the free threading binaries are marked with a “t” suffix.


```
$ python3.13 -VV
Python 3.13.0 (main, Oct 12 2024, 23:07:28) [Clang 16.0.0 (clang-1600.0.26.3)]

$ python3.13t -VV
Python 3.13.0 experimental free-threading build (main, Oct 10 2024, 21:45:38) [Clang 16.0.0 (clang-1600.0.26.3)]
```

 amyreese  amethyst@n7.gg

On my machine, I have both builds installed, and we can see that python 3.13 “t” tells me that it’s the “experimental free-threading build”. This provides a good opportunity for us to compare the two builds in performance.

```
def spinner() -> int:
    values = []
    for k in range(10_000_000):
        values += [k // 37]
    return sum(values)
```

 amyreese  amethyst @ n7.gg

Let's look at a micro benchmark that's purely CPU-bound — we iterate through ten million integers, do some integer division, create a giant list of math results, and then sum up the list when we're done. This is intentionally a bit sloppy, creating new list objects on each iteration before throwing them away, so that we can exercise the process of creating, modifying, and discarding Python objects. This is also a self-contained, single threaded benchmark.

```
$ python3.13 -m timeit -s 'from speed import spinner' 'spinner()'
1 loop, best of 5: 767 msec per loop

$ python3.13t -m timeit -s 'from speed import spinner' 'spinner()'
1 loop, best of 5: 855 msec per loop
```

 amyreese  amethyst@n7.gg

Now we can use timeit to compare the performance of this benchmark on both builds, and on my M1 Mac Mini, I see a roughly 11% increase in runtime when running this on the free threading build. Not bad, but not great. But of course this is just a single thread. Let's make it multi-threaded.


```
from concurrent.futures import ThreadPoolExecutor, wait

def spinner() -> int:
    values = []
    for k in range(10_000_000):
        values += [k // 37]
    return sum(values)

def multi_spinner():
    pool = ThreadPoolExecutor()
    futures = [pool.submit(spinner) for _ in range(16)]
    results = wait(futures)
```

 amyreese  amethyst@n7.gg

We can take our previous benchmark, and just run it a bunch of times on a thread pool. We simply create a future for each time we want to run the benchmark, and wait for all of the futures to complete.

```
$ time python3.13 -m timeit \  
> -s 'from speed import multi_spinner' 'multi_spinner()' \  
1 loop, best of 5: 12.4 sec per loop \  
python3.13 -m timeit -s 'from speed import multi_spinner' 'multi_spinner' \  
()' 69.65s user 5.76s system 99% cpu 1:15.96 total \  
  
$ time python3.13t -m timeit \  
-s 'from speed import multi_spinner' 'multi_spinner()' \  
1 loop, best of 5: 6.36 sec per loop \  
python3.13t -m timeit -s 'from speed import multi_spinner' 'multi_spinne' \  
r()' 125.14s user 55.99s system 462% cpu 39.143 total
```

 amyreese  amethyst@n7.gg

Again we'll use timeit and compare the two runtimes. We'll also use zshell's `time` builtin, which will tell us average CPU utilization for each command. On my M1 Mac Mini, I only have four performance cores, and in this case, was getting thermally throttled by the benchmark, but we can still see that the free threaded version completed in half the time, and CPU utilization went from 99%, essentially using only a single core, to 462%, maxing out all four efficiency cores on my machine. Even with the slower single core performance, being able to actually utilize all four cores makes a *huge* difference for CPU-bound work loads, and with the right conditions, can scale to machines with dozens of cores or more.



But of course there's a catch, and as usual, it's related to the Python ecosystem. Packages and libraries really need to be built (or rebuilt) with free threading in mind. Popular packages with native extensions, like numpy or scipy that depend on C, C++, Rust, or even Fortran code, need to explicitly add and declare support for free threading Python. Importing native extensions that don't support free threading will result in Python turning the GIL back on for safety, eliminating the multithreaded performance benefits.

dead code

 amyreese  amethyst@n7.gg

And even after the big name packages add support, unfortunately there is a lot of dead code out there, some of which has managed to find its way into the dependencies of common work loads. Some of these packages may take weeks, months, or even years to get updated, if they ever support it at all. Unless or until all of your dependencies support free threading, it might be difficult realizing these performance gains without intentional efforts to minimize reliance on these libraries.



your code

 amyreese  @amethyst@n7.gg

You might actually find that the problem is actually in code that you care about, in projects that unintentionally depended on global locking to maintain consistent state, or in libraries that aren't thread safe and rely on global state in a way that breaks under multithreading. The good news is that this is under your control. You can find the edge cases; you can eliminate global state; you can make your code thread safe. And this is where I get to plug one of my own personal projects.

unittest-ft

 amyreese  amethyst @ n7.gg

Inspired by similar plugins for pytest, unittest-ft is a tool for running your unit tests concurrently or in parallel, using a thread pool similar to the benchmark examples from earlier.

```
$ unittest-ft
.S..XX.
-----
Ran 7 tests in 461.74ms

OK (skipped=1, expected failures=2)
```

 amyreese  amethyst @ n7.gg

Out of the box, unittest-ft looks and acts a lot like the standard unittest runner, but under the hood it's actually spawning and running these tests in parallel. This can help run your test suite faster on Python 3.13 with free threading, and it will even show you how much time it saved when doing so, but it can also help to find and expose parts of your applications or libraries that aren't thread safe, even on older versions of Python that you might still be using in production.

randomized stress test

 amyreese  amethyst@n7.gg

One of the ways it can do this is with randomized stress tests. Instead of running all of your tests once in a predictable fashion, unittest-ft can run all of your tests ten times, in a completely random order every time, while also spreading them across separate threads. This can uncover thread safety issues when running concurrent tests, as well as find test cases that were unintentionally dependent on previous tests running before them.


```
$ unittest-ft --randomize --stress-test  
.S...S....S..SXXXXXS....X..XXSX.SXSXXXX..XXS.X...X....XS.....  
-----  
Ran 70 tests in 1.218s (saved 10.178s)  
  
OK (skipped=10, expected failures=20)
```

 amyreese  amethyst @ n7.gg

So now if we re-run our test suite with a randomized stress-test, we can see 10 times as many tests ran, and thanks to free threading, we can see that by running this many tests in parallel, we saved 10 seconds over the comparable amount of time that would have been spent running them in serial. And since this stress test is passing, we now have better confidence that our library and its test suite is robust and capable of running correctly in a multithreaded environment. We're ready for the future that everyone has been waiting for.



pip install unittest-ft
github.com/amyreese/unittest-ft

 amyreese  amethyst @ n7.gg

Now if you'd like to see if your own projects are thread safe, you can install unittest-ft today and run a stress test, without rewriting any of your tests. It's still an early beta project, but it's built on top of the standard library unit testing framework, so it's highly unlikely to eat your homework. I hope you find it as useful as I have for speeding up your development workflow. Most importantly, being able to create tools like this is a perfect example of why I think ...



... Python 3.13 is kind of a big deal. Thank you!