

python's pre-declared constants are kinda weird

2026-08-01

python has 6 pre-declared "constants": True, False, None, `__debug__`, Ellipsis (or equivalently `...`), and `NotImplemented`. but they all behave slightly differently, for some reason.

True, False, and None

True, False, and None are keywords. they aren't identifiers, they're just straight up their own lexical tokens. which is really weird; nothing else is like this in python. usually stuff is resolved during regular name resolution, not in the lexer itself.

an interesting side effect of this is that expressions like `x.True` raise a `SyntaxError`. i'm curious as to what the rationale was for this decision (if there was one).

there's some more interesting stuff with these constants, but i'll get to it later, since it ties in with the other constants.

`__debug__`

`__debug__` is a boolean constant: it's normally True, but when running with `-O`, it's False. the idea is similar to how `assert` is disabled in non-debug builds: you can wrap code in `if __debug__` if the check would be too expensive in an "optimized" build, or something.

`__debug__` is really interesting though, because although it's a normal identifier (unlike True, False, and None), it's the only identifier in the language which can't be assigned to:

```
>>> __debug__ = 67
      File "", line 1
SyntaxError: cannot assign to __debug__
```

you can't even assign to it as an attribute:

```
>>> x.__debug__ = 67
      File "", line 1
SyntaxError: cannot assign to __debug__
```

again, no other identifier behaves like this. this is a true special case.

but because it's not a keyword, it behaves slightly differently to True, False, and None: `x.__debug__` raises `AttributeError` (rather than `SyntaxError`), since it's syntactically valid; it's just looking up an attribute which doesn't exist.

interestingly, there's also a special error message for attempting to delete `__debug__` (despite the fact that this would raise a `NameError` anyway if not for the special case), but this *doesn't* apply for deleting an attribute named `__debug__`:

```
>>> del __debug__
      File "", line 1
SyntaxError: cannot delete __debug__
>>> del x.__debug__
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'x' is not defined
```

if `x` were defined, an `AttributeError` would be raised instead. in either case, it's not a `SyntaxError` (unlike assignment), for some reason.

tangent: `SyntaxError` is a lie

speaking of errors: assigning to `__debug__` is one of only a few cases i'm aware of where a `SyntaxError` is raised despite something not actually being invalid syntax. here, you can confirm it yourself:

```
>>> assert (__debug__ := 67)
```

running that assert in a debug build raises a `SyntaxError`, but with `-O`, the assertion is never compiled, and so no exception is raised.

two other instances of this are using `yield` or `await` outside of a function:

```
>>> assert (yield)
>>> assert (await 67)
```

Ellipsis and NotImplemented

`Ellipsis` and `NotImplemented` are documented in the "constants" section of the reference, but unlike the other 4 constants, they aren't "real" constants. they're just normal builtins, so they can be shadowed by globals:

```
>>> NotImplemented = 67
>>> NotImplemented
67
```

again, i'm curious about the rationale here. why is it that these aren't special, but the other constants are?

overwriting constants

here's something interesting: despite being lexical tokens, `True`, `False`, and `None` also exist as normal builtins:

```
>>> import builtins
>>> getattr(builtins, 'True')
True
>>> getattr(builtins, 'False')
False
>>> getattr(builtins, 'None') is None
True
```

there's no way to directly access these without using `getattr`.

but here's where things get really interesting: `setattr` also works!

```
>>> setattr(builtins, 'True', 67)
>>> getattr(builtins, 'True')
67
```

however, this doesn't change the value when accessed with the lexical token:

```
>>> True
True
```

but `__debug__` has the same behavior!

```
>>> setattr(builtins, '__debug__', 67)
>>> builtins.__debug__
67
>>> __debug__
True
```

so `__debug__` can *sorta* be assigned to, but despite not being a lexical token, it's special cased just like `True`, `False`, and `None`: its value is unaffected by changes to the `builtins` module. so it really is a constant!

`Ellipsis` and `NotImplemented` are, once again, not actually constants:

```
>>> setattr(builtins, 'Ellipsis', 67)
>>> Ellipsis
67
```

this doesn't change the value of `...` though:

```
>>> ...
Ellipsis
```

so in some sense, ... is a real constant, but Ellipsis isn't. weird, right?