`!codeop` --- Compile Python code

codeop

Source code: `Lib/codeop.py`

The `!codeop` module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the `code` module. As a result, you probably don't want to use the module directly; if you want to include such a loop in your program you probably want to use the `code` module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print '`>>>`' or '`...`' next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The `!codeop` module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

compile_command(source, filename="<input>", symbol="single")

Tries to compile *source*, which should be a string of Python code and return a code object if *source* is valid Python code. In that case, the filename attribute of the code object will be *filename*, which defaults to `'<input>'`. Returns `None` if *source* is *not* valid Python code, but is a prefix of valid Python code.

If there is a problem with *source*, an exception will be raised. `SyntaxError` is raised if there is invalid Python syntax, and `OverflowError` or `ValueError` if there is an invalid literal.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of `statement` (`'exec'`) or as an `expression` (`'eval'`). Any other value will cause `ValueError` to be raised.

> [!NOTE] It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

Instances of this class have `~object.__call__` methods identical in signature to the built-in function `compile`, but with the difference that if the instance compiles program text containing a `__future__` statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

Instances of this class have `~object.__call__` methods identical in signature to `compile_command`; the difference is that if the instance compiles program text containing a `__future__` statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

□□□ #2
□ □□□ □□□ 2026-10-31 17:05:31 UTC
□ □□□ □□□ 2026-10-31 17:05:33 UTC