29.2. codeop — Compile Python code¶

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:

`codeop.compile_command`(*source*[, *filename*[, *symbol*]])¶

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) 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.

class `codeop.Compile`¶

Instances of this class have `__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.

class `codeop.CommandCompiler`¶

Instances of this class have `__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.

A note on version compatibility: the `Compile` and `CommandCompiler` are new in Python 2.2. If you want to enable the future-tracking features of 2.2 but also retain compatibility with 2.1 and earlier versions of Python you can either write

```
try:
   from codeop import CommandCompiler
   compile_command = CommandCompiler()
   del CommandCompiler
except ImportError:
   from codeop import compile_command
```

which is a low-impact change, but introduces possibly unwanted global state into your program, or you can write:

```
try:
   from codeop import CommandCompiler
except ImportError:
   def CommandCompiler():
       from codeop import compile_command
       return compile_command
```

and then call `CommandCompiler` every time you need a fresh compiler object.

Previous topic

29.1. code — Interpreter base classes

Next topic

30. Restricted Execution

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 29. Custom Python Interpreters »

Last updated on Feb 26, 2010. Created using Sphinx 0.6.3.