code — Interpreter base classes

Source code: Lib/code.py

The code module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class code.**InteractiveInterpreter**(*locals=None*)

This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional *locals* argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key '__name__' set to '__console__' and key '__doc__' set to None.

class code.**InteractiveConsole**(*locals=None*, *filename="<console>"*)

Closely emulate the behavior of the interactive Python interpreter. This class builds on InteractiveInterpreter and adds prompting using the familiar sys.ps1 and sys.ps2, and input buffering.

code.**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*)

Convenience function to run a read-eval-print loop. This creates a new instance of InteractiveConsole and sets *readfunc* to be used as the InteractiveConsole.raw_input() method, if provided. If *local* is provided, it is passed to the InteractiveConsole constructor for use as the default namespace for the interpreter loop. The interact() method of the instance is then run with *banner* and *exitmsg* passed as the banner and exit message to use, if provided. The console object is discarded after use.

Changed in version 3.6: Added *exitmsg* parameter.

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

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

source is the source string; *filename* is the optional filename from which source was read, defaulting to '<input>'; and *symbol* is the optional grammar start symbol, which should be either 'single' (the default) or 'eval'.

Returns a code object (the same as compile(source, filename, symbol)) if the command is complete and valid; None if the command is incomplete; raises SyntaxError if the

command is complete and contains a syntax error, or raises `OverflowError` or `ValueError` if the command contains an invalid literal.

Interactive Interpreter Objects

`InteractiveInterpreter.`**`runsource`**(*source*, *filename="<input>"*, *symbol="single"*)

Compile and run some source in the interpreter. Arguments are the same as for `compile_command()`; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One several things can happen:

- The input is incorrect; `compile_command()` raised an exception (`SyntaxError` or `OverflowError`). A syntax traceback will be printed by calling the `showsyntaxerror()` method. `runsource()` returns `False`.
- The input is incomplete, and more input is required; `compile_command()` returned `None`. `runsource()` returns `True`.
- The input is complete; `compile_command()` returned a code object. The code is executed by calling the `runcode()` (which also handles run-time exceptions, except for `SystemExit`). `runsource()` returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

`InteractiveInterpreter.`**`runcode`**(*code*)

Execute a code object. When an exception occurs, `showtraceback()` is called to display a traceback. All exceptions are caught except `SystemExit`, which is allowed to propagate.

A note about `KeyboardInterrupt`: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

`InteractiveInterpreter.`**`showsyntaxerror`**(*filename=None*)

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If *filename* is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses `'<string>'` when reading from a string. The output is written by the `write()` method.

`InteractiveInterpreter.`**`showtraceback`**()

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

Changed in version 3.5: The full chained traceback is displayed instead of just the primary traceback.

`InteractiveInterpreter.`**`write`**(*data*)

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.`**`interact`**(*banner=None*, *exitmsg=None*)

Closely emulate the interactive Python console. The optional *banner* argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

The optional *exitmsg* argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If *exitmsg* is not given or `None`, a default message is printed.

Changed in version 3.4: To suppress printing any banner, pass an empty string.

Changed in version 3.6: Print an exit message when exiting.

`InteractiveConsole.`**`push`**(*line*)

Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

`InteractiveConsole.`**`resetbuffer`**()

Remove any unhandled source text from the input buffer.

`InteractiveConsole.`**`raw_input`**(*prompt=""*)

Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, `EOFError` is raised. The base implementation reads from `sys.stdin`; a subclass may replace this with a different implementation.