code

- Code Objects
- Code Object Flags
- Extra information

object; code, code object

Code Objects

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn't yet been bound into a function.

> The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

> This is an instance of `PyTypeObject` representing the Python `code object <code-objects>`.

> Return true if *co* is a `code object <code-objects>`. This function always succeeds.

> Return the number of `free (closure) variables <closure variable>` in a code object.

>

Return the position of the first `free (closure) variable <closure variable>` in a code object.

3.13

Renamed from `PyCode_GetFirstFree` as part of `unstable-c-api`. The old name is deprecated, but will remain available until the signature changes again.

> Return a new code object. If you need a dummy code object to create a frame, use `PyCode_NewEmpty` instead.
>
> Since the definition of the bytecode changes often, calling `PyUnstable_Code_New` directly can bind you to a precise Python version.
>
> The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.
>
> 3.11 Added `qualname` and `exceptiontable` parameters.
>
> single: PyCode_New (C function)
>
> 3.12
>
> Renamed from `PyCode_New` as part of `unstable-c-api`. The old name is deprecated, but will remain available until the signature changes again.

> Similar to `PyUnstable_Code_New`, but with an extra "posonlyargcount" for positional-only arguments. The same caveats that apply to `PyUnstable_Code_New` also apply to this function.
>
> 3.8 as `PyCode_NewWithPosOnlyArgs`
>
> 3.11 Added `qualname` and `exceptiontable` parameters.
>
> 3.12
>
> Renamed to `PyUnstable_Code_NewWithPosOnlyArgs`. The old name is deprecated, but will remain available until the signature changes again.

> Return a new empty code object with the specified filename, function name, and first line number. The resulting code object will raise an `Exception` if executed.

> Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. This may be `O(N)` in the number of instructions in the code object, so it is not performant for iterating over all the line numbers in a code object.
>
> If you just need the line number of a frame, use `PyFrame_GetLineNumber` instead.

> Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.
>
> Returns `1` if the function succeeds and 0 otherwise.
>
> 3.11

> Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a `PyBytesObject` representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.
>
> This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.
>
> 3.11

> Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables. On error, `NULL` is returned and an exception is raised.
>
> 3.11

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

3.11

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a `PyTupleObject` containing the names of the `free (closure) variables <closure variable>`. On error, `NULL` is returned and an exception is raised.

3.11

Register *callback* as a code object watcher for the current interpreter. Return an ID which may be passed to `PyCode_ClearWatcher`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

3.12

Clear watcher identified by *watcher_id* previously returned from `PyCode_AddWatcher` for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

3.12

Enumeration of possible code object watcher events: - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

3.12

Apply the macro *V* to each suffix of `PyCodeEvent` constants. Currently expands to `V(CREATE) V(DESTROY)`. The macro *V* must accept one event name and supply any

punctuation needed between expansions.

> Type of a code object watcher callback function.
>
> If *event* is `PY_CODE_EVENT_CREATE`, then the callback is invoked after *co* has been fully initialized. Otherwise, the callback is invoked before the destruction of *co* takes place, so the prior state of *co* can be inspected.
>
> If *event* is `PY_CODE_EVENT_DESTROY`, taking a reference in the callback to the about-to-be-destroyed code object will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.
>
> Users of this API should not rely on internal runtime implementation details. Such details may include, but are not limited to, the exact order and timing of creation and destruction of code objects. While changes in these details may result in differences observable by watchers (including whether a callback is invoked or not), it does not change the semantics of the Python code being executed.
>
> If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using `PyErr_WriteUnraisable`. Otherwise it should return `0`.
>
> There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.
>
> 3.12

> This is a function that does nothing.
>
> Prior to Python 3.10, this function would perform basic optimizations to a code object.
>
> 3.10 This function now does nothing.
>
> 3.13

Code Object Flags

Code objects contain a bit-field of flags, which can be retrieved as the `~codeobject.co_flags` Python attribute (for example using `PyObject_GetAttrString`), and set using a *flags* argument to `PyUnstable_Code_New` and similar functions.

Flags whose names start with `CO_FUTURE_` correspond to features normally selectable by `future statements <future>`. These flags can be used in `PyCompilerFlags.cf_flags`. Note that many `CO_FUTURE_` flags are mandatory in current versions of Python, and setting them has no effect.

The following flags are available. For their meaning, see the linked documentation of their Python equivalents.

Flag	Meaning
	`inspect.CO_OPTIMIZED`
	`inspect.CO_NEWLOCALS`
	`inspect.CO_VARARGS`
	`inspect.CO_VARKEYWORDS`
	`inspect.CO_NESTED`
	`inspect.CO_GENERATOR`
	`inspect.CO_COROUTINE`
	`inspect.CO_ITERABLE_COROUTINE`
	`inspect.CO_ASYNC_GENERATOR`
	`inspect.CO_HAS_DOCSTRING`
	`inspect.CO_METHOD`
	no effect (`__future__.division`)
	no effect (`__future__.absolute_import`)
	no effect (`__future__.with_statement`)
	no effect (`__future__.print_function`)
	no effect (`__future__.unicode_literals`)
	no effect (`__future__.generator_stop`)
	`__future__.annotations`

Extra information

To support low-level extensions to frame evaluation, such as external just-in-time compilers, it is possible to attach arbitrary extra data to code objects.

These functions are part of the unstable C API tier: this functionality is a CPython implementation detail, and the API may change without deprecation warnings.

> Return a new opaque index value used to adding data to code objects.
>
> You generally call this function once (per interpreter). Use the result with `PyUnstable_Code_GetExtra` and `PyUnstable_Code_SetExtra` to manipulate data on individual code objects.
>
> If *free* is not `NULL`: when a code object is deallocated, *free* will be called on non-`NULL` data stored under the new index. Use `Py_DecRef` when storing `PyObject`.
>
> 3.6 as `_PyEval_RequestCodeExtraIndex`
>
> 3.12
>
> Renamed to `PyUnstable_Eval_RequestCodeExtraIndex`. The old private name is deprecated, but will be available until the API changes.

> Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.
>
> If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.
>
> 3.6 as `_PyCode_GetExtra`
>
> 3.12
>
> Renamed to `PyUnstable_Code_GetExtra`. The old private name is deprecated, but will be available until the API changes.

> Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.
>
> 3.6 as `_PyCode_SetExtra`

> 3.12
>
> Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

□□□ #2
□ □□□ □□□ 2026-10-31 16:50:06 UTC
□ □□□ □□□ 2026-10-31 16:50:09 UTC