dis — Disassembler for Python bytecode

Source code: Lib/dis.py

The dis module supports the analysis of CPython bytecode by disassembling it. The CPython bytecode which this module takes as an input is defined in the file Include/opcode.h and used by the compiler and the interpreter.

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Changed in version 3.6: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Example: Given the function myfunc():

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of myfunc():

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

(The “2” is a line number).

Bytecode analysis

New in version 3.4.

The bytecode analysis API allows pieces of Python code to be wrapped in a Bytecode object that provides easy access to details of the compiled code.

class dis.**Bytecode**(*x*, ***, *first_line=None*, *current_offset=None*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by compile()).

This is a convenience wrapper around many of the functions listed below, most notably get_instructions(), as iterating over a Bytecode instance yields the bytecode operations as Instruction instances.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means `dis()` will display a “current instruction” marker against the specified opcode.

classmethod **from_traceback**(*tb*)

Construct a `Bytecode` instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()

Return a formatted view of the bytecode operations (the same as printed by `dis.dis()`, but returned as a multi-line string).

info()

Return a formatted multi-line string with detailed information about the code object, like `code_info()`.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

Example:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

Analysis functions

The `dis` module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn’t useful:

dis.**code_info**(*x*)

Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

New in version 3.2.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

`dis.`**`show_code`**(*x*, *, *file=None*)

Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

New in version 3.2.

Changed in version 3.4: Added *file* parameter.

`dis.`**`dis`**(*x=None*, *, *file=None*, *depth=None*)

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). Strings are first compiled to code objects with the `compile()` built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.7: Implemented recursive disassembling and added *depth* parameter.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

`dis.`**`distb`**(*tb=None*, *, *file=None*)

Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

`dis.`**`disassemble`**(*code*, *lasti=-1*, ***, *file=None*)
`dis.`**`disco`**(*code*, *lasti=-1*, ***, *file=None*)

Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

`dis.`**`get_instructions`**(*x*, ***, *first_line=None*)

Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of `Instruction` named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

New in version 3.4.

`dis.`**`findlinestarts`**(*code*)

This generator function uses the `co_firstlineno` and `co_lnotab` attributes of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs. See Objects/lnotab_notes.txt for the `co_lnotab` format and how to decode it.

Changed in version 3.6: Line numbers can be decreasing. Before, they were always increasing.

`dis.`**`findlabels`**(*code*)

Detect all offsets in the code object *code* which are jump targets, and return a list of these offsets.

`dis.`**`stack_effect`**(*opcode*, *oparg=None*, ***, *jump=None*)

Compute the stack effect of *opcode* with argument *oparg*.

If the code has a jump target and *jump* is `True`, `stack_effect()` will return the stack effect of jumping. If *jump* is `False`, it will return the stack effect of not jumping. And if *jump* is `None` (default), it will return the maximal stack effect of both cases.

New in version 3.4.

Changed in version 3.8: Added *jump* parameter.

Python Bytecode Instructions

The `get_instructions()` function and `Bytecode` class provide details of bytecode instructions as `Instruction` instances:

class `dis.`**`Instruction`**

Details for a bytecode operation

`opcode`

numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the Opcode collections.

`opname`

human readable name for operation

`arg`

numeric argument to operation (if any), otherwise `None`

`argval`

resolved arg value (if known), otherwise same as arg

`argrepr`

human readable description of operation argument

`offset`

start index of operation within bytecode sequence

`starts_line`

line started by this opcode (if any), otherwise `None`

`is_jump_target`

`True` if other code jumps to here, otherwise `False`

New in version 3.4.

The Python compiler currently generates the following bytecode instructions.

General instructions

`NOP`

Do nothing code. Used as a placeholder by the bytecode optimizer.

`POP_TOP`

Removes the top-of-stack (TOS) item.

`ROT_TWO`

Swaps the two top-most stack items.

`ROT_THREE`

Lifts second and third stack item one position up, moves top down to position three.

`ROT_FOUR`

Lifts second, third and forth stack items one position up, moves top down to position four.

New in version 3.8.

`DUP_TOP`

Duplicates the reference on top of the stack.

New in version 3.2.

`DUP_TOP_TWO`

Duplicates the two references on top of the stack, leaving them in the same order.

New in version 3.2.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

`UNARY_POSITIVE`

Implements `TOS = +TOS`.

`UNARY_NEGATIVE`

Implements `TOS = -TOS`.

`UNARY_NOT`

Implements `TOS = not TOS`.

`UNARY_INVERT`

Implements `TOS = ~TOS`.

`GET_ITER`

Implements `TOS = iter(TOS)`.

`GET_YIELD_FROM_ITER`

If `TOS` is a generator iterator or coroutine object it is left as is. Otherwise, implements `TOS = iter(TOS)`.

New in version 3.5.

Binary operations

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

`BINARY_POWER`

Implements `TOS = TOS1 ** TOS`.

`BINARY_MULTIPLY`

Implements `TOS = TOS1 * TOS`.

`BINARY_MATRIX_MULTIPLY`

Implements `TOS = TOS1 @ TOS`.

New in version 3.5.

`BINARY_FLOOR_DIVIDE`

Implements `TOS = TOS1 // TOS`.

`BINARY_TRUE_DIVIDE`

Implements `TOS = TOS1 / TOS`.

`BINARY_MODULO`

Implements `TOS = TOS1 % TOS`.

`BINARY_ADD`

Implements `TOS = TOS1 + TOS`.

`BINARY_SUBTRACT`

Implements `TOS = TOS1 - TOS`.

`BINARY_SUBSCR`

Implements `TOS = TOS1[TOS]`.

`BINARY_LSHIFT`

Implements `TOS = TOS1 << TOS`.

`BINARY_RSHIFT`

Implements `TOS = TOS1 >> TOS`.

`BINARY_AND`

Implements `TOS = TOS1 & TOS`.

`BINARY_XOR`

Implements `TOS = TOS1 ^ TOS`.

`BINARY_OR`

Implements `TOS = TOS1 | TOS`.

In-place operations

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

`INPLACE_POWER`

Implements in-place `TOS = TOS1 ** TOS`.

`INPLACE_MULTIPLY`

Implements in-place `TOS = TOS1 * TOS`.

`INPLACE_MATRIX_MULTIPLY`

Implements in-place `TOS = TOS1 @ TOS`.

New in version 3.5.

`INPLACE_FLOOR_DIVIDE`

Implements in-place `TOS = TOS1 // TOS`.

`INPLACE_TRUE_DIVIDE`

Implements in-place `TOS = TOS1 / TOS`.

`INPLACE_MODULO`

Implements in-place `TOS = TOS1 % TOS`.

INPLACE_ADD

Implements in-place `TOS = TOS1 + TOS`.

INPLACE_SUBTRACT

Implements in-place `TOS = TOS1 - TOS`.

INPLACE_LSHIFT

Implements in-place `TOS = TOS1 << TOS`.

INPLACE_RSHIFT

Implements in-place `TOS = TOS1 >> TOS`.

INPLACE_AND

Implements in-place `TOS = TOS1 & TOS`.

INPLACE_XOR

Implements in-place `TOS = TOS1 ^ TOS`.

INPLACE_OR

Implements in-place `TOS = TOS1 | TOS`.

STORE_SUBSCR

Implements `TOS1[TOS] = TOS2`.

DELETE_SUBSCR

Implements `del TOS1[TOS]`.

Coroutine opcodes

GET_AWAITABLE

Implements `TOS = get_awaitable(TOS)`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the CO_ITERABLE_COROUTINE flag, or resolves `o.__await__`.

New in version 3.5.

GET_AITER

Implements `TOS = TOS.__aiter__()`.

New in version 3.5.

Changed in version 3.7: Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT

Implements `PUSH(get_awaitable(TOS.__anext__()))`. See `GET_AWAITABLE` for details about `get_awaitable`

New in version 3.5.

END_ASYNC_FOR

Terminates an `async for` loop. Handles an exception raised when awaiting a next item. If TOS is `StopAsyncIteration` pop 7 values from the stack and restore the exception state using the second three of them. Otherwise re-raise the exception using the three values from the stack. An exception handler block is removed from the block stack.

New in version 3.8.

BEFORE_ASYNC_WITH

Resolves `__aenter__` and `__aexit__` from the object on top of the stack. Pushes `__aexit__` and result of `__aenter__()` to the stack.

New in version 3.5.

SETUP_ASYNC_WITH

Creates a new frame object.

New in version 3.5.

Miscellaneous opcodes

PRINT_EXPR

Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with `POP_TOP`.

SET_ADD(*i*)

Calls `set.add(TOS1[-i], TOS)`. Used to implement set comprehensions.

LIST_APPEND(*i*)

Calls `list.append(TOS[-i], TOS)`. Used to implement list comprehensions.

MAP_ADD(*i*)

Calls `dict.__setitem__(TOS1[-i], TOS1, TOS)`. Used to implement dict comprehensions.

New in version 3.1.

Changed in version 3.8: Map value is TOS and map key is TOS1. Before, those were reversed.

For all of the `SET_ADD`, `LIST_APPEND` and `MAP_ADD` instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE

Returns with TOS to the caller of the function.

YIELD_VALUE

Pops TOS and yields it from a generator.

YIELD_FROM

Pops TOS and delegates to it as a subiterator from a generator.

New in version 3.3.

SETUP_ANNOTATIONS

Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains variable annotations statically.

New in version 3.6.

IMPORT_STAR

Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

POP_BLOCK

Removes one block from the block stack. Per frame, there is a stack of blocks, denoting `try` statements, and such.

POP_EXCEPT

Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

POP_FINALLY(*preserve_tos*)

Cleans up the value stack and the block stack. If *preserve_tos* is not `0` TOS first is popped from the stack and pushed on the stack after performing other stack operations:

- If TOS is `NULL` or an integer (pushed by `BEGIN_FINALLY` or `CALL_FINALLY`) it is popped from the stack.
- If TOS is an exception type (pushed when an exception has been raised) 6 values are popped from the stack, the last three popped values are used to restore the exception state. An exception handler block is removed from the block stack.

It is similar to `END_FINALLY`, but doesn’t change the bytecode counter nor raise an exception. Used for implementing `break`, `continue` and `return` in the `finally` block.

New in version 3.8.

BEGIN_FINALLY

Pushes `NULL` onto the stack for using it in `END_FINALLY`, `POP_FINALLY`, `WITH_CLEANUP_START` and `WITH_CLEANUP_FINISH`. Starts the `finally` block.

New in version 3.8.

END_FINALLY

Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised or execution has to be continued depending on the value of TOS.

- If TOS is `NULL` (pushed by `BEGIN_FINALLY`) continue from the next instruction. TOS is popped.
- If TOS is an integer (pushed by `CALL_FINALLY`), sets the bytecode counter to TOS. TOS is popped.
- If TOS is an exception type (pushed when an exception has been raised) 6 values are popped from the stack, the first three popped values are used to re-raise the exception and the last three popped values are used to restore the exception state. An exception handler block is removed from the block stack.

LOAD_BUILD_CLASS

Pushes `builtins.__build_class__()` onto the stack. It is later called by `CALL_FUNCTION` to construct a class.

SETUP_WITH(*delta*)

This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by `WITH_CLEANUP_START`. Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the `__enter__()` method is pushed onto the stack. The next opcode will either ignore it (`POP_TOP`), or store it in (a) variable(s) (`STORE_FAST`, `STORE_NAME`, or `UNPACK_SEQUENCE`).

New in version 3.2.

WITH_CLEANUP_START

Starts cleaning up the stack when a `with` statement block exits.

At the top of the stack are either `NULL` (pushed by `BEGIN_FINALLY`) or 6 values pushed if an exception has been raised in the with block. Below is the context manager’s `__exit__()` or `__aexit__()` bound method.

If TOS is `NULL`, calls `SECOND(None, None, None)`, removes the function from the stack, leaving TOS, and pushes `None` to the stack. Otherwise calls `SEVENTH(TOP, SECOND, THIRD)`, shifts the bottom 3 values of the stack down, replaces the empty spot with `NULL` and pushes TOS. Finally pushes the result of the call.

WITH_CLEANUP_FINISH

Finishes cleaning up the stack when a `with` statement block exits.

TOS is result of `__exit__()` or `__aexit__()` function call pushed by `WITH_CLEANUP_START`. SECOND is `None` or an exception type (pushed when an exception has been raised).

Pops two values from the stack. If SECOND is not None and TOS is true unwinds the EXCEPT_HANDLER block which was created when the exception was caught and pushes `NULL` to the stack.

All of the following opcodes use their arguments.

STORE_NAME(*namei*)

Implements `name = TOS`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use `STORE_FAST` or `STORE_GLOBAL` if possible.

DELETE_NAME(*namei*)

Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE(*count*)

Unpacks TOS into *count* individual values, which are put onto the stack right-to-left.

UNPACK_EX(*counts*)

Implements assignment with a starred target: Unpacks an iterable in TOS into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it. The resulting values are put onto the stack right-to-left.

STORE_ATTR(*namei*)

Implements `TOS.name = TOS1`, where *namei* is the index of name in `co_names`.

DELETE_ATTR(*namei*)

Implements `del TOS.name`, using *namei* as index into `co_names`.

STORE_GLOBAL(*namei*)

Works as `STORE_NAME`, but stores the name as a global.

DELETE_GLOBAL(*namei*)

Works as `DELETE_NAME`, but deletes a global name.

LOAD_CONST(*consti*)

Pushes `co_consts[consti]` onto the stack.

LOAD_NAME(*namei*)

Pushes the value associated with `co_names[namei]` onto the stack.

BUILD_TUPLE(*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST(*count*)

Works as `BUILD_TUPLE`, but creates a list.

BUILD_SET(*count*)

Works as `BUILD_TUPLE`, but creates a set.

BUILD_MAP(*count*)

Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., TOS3: TOS2, TOS1: TOS}`.

Changed in version 3.5: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP(*count*)

The version of `BUILD_MAP` specialized for constant keys. *count* values are consumed from the stack. The top element on the stack contains a tuple of keys.

New in version 3.6.

BUILD_STRING(*count*)

Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

New in version 3.6.

BUILD_TUPLE_UNPACK(*count*)

Pops *count* iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays `(*x, *y, *z)`.

New in version 3.5.

BUILD_TUPLE_UNPACK_WITH_CALL(*count*)

This is similar to `BUILD_TUPLE_UNPACK`, but is used for `f(*x, *y, *z)` call syntax. The stack item at position `count + 1` should be the corresponding callable `f`.

New in version 3.6.

BUILD_LIST_UNPACK(*count*)

This is similar to `BUILD_TUPLE_UNPACK`, but pushes a list instead of tuple. Implements iterable unpacking in list displays `[*x, *y, *z]`.

New in version 3.5.

BUILD_SET_UNPACK(*count*)

This is similar to BUILD_TUPLE_UNPACK, but pushes a set instead of tuple. Implements iterable unpacking in set displays {*x, *y, *z}.

New in version 3.5.

BUILD_MAP_UNPACK(*count*)

Pops *count* mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays {**x, **y, **z}.

New in version 3.5.

BUILD_MAP_UNPACK_WITH_CALL(*count*)

This is similar to BUILD_MAP_UNPACK, but is used for f(**x, **y, **z) call syntax. The stack item at position count + 2 should be the corresponding callable f.

New in version 3.5.

Changed in version 3.6: The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

LOAD_ATTR(*namei*)

Replaces TOS with getattr(TOS, co_names[namei]).

COMPARE_OP(*opname*)

Performs a Boolean operation. The operation name can be found in cmp_op[opname].

IMPORT_NAME(*namei*)

Imports the module co_names[namei]. TOS and TOS1 are popped and provide the *fromlist* and *level* arguments of __import__(). The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent STORE_FAST instruction modifies the namespace.

IMPORT_FROM(*namei*)

Loads the attribute co_names[namei] from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a STORE_FAST instruction.

JUMP_FORWARD(*delta*)

Increments bytecode counter by *delta*.

POP_JUMP_IF_TRUE(*target*)

If TOS is true, sets the bytecode counter to *target*. TOS is popped.

New in version 3.1.

POP_JUMP_IF_FALSE(*target*)

If TOS is false, sets the bytecode counter to *target*. TOS is popped.

New in version 3.1.

`JUMP_IF_TRUE_OR_POP`(*target*)

If TOS is true, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

New in version 3.1.

`JUMP_IF_FALSE_OR_POP`(*target*)

If TOS is false, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

New in version 3.1.

`JUMP_ABSOLUTE`(*target*)

Set bytecode counter to *target*.

`FOR_ITER`(*delta*)

TOS is an iterator. Call its `__next__()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by *delta*.

`LOAD_GLOBAL`(*namei*)

Loads the global named `co_names[namei]` onto the stack.

`SETUP_FINALLY`(*delta*)

Pushes a try block from a try-finally or try-except clause onto the block stack. *delta* points to the finally block or the first except block.

`CALL_FINALLY`(*delta*)

Pushes the address of the next instruction onto the stack and increments bytecode counter by *delta*. Used for calling the finally block as a “subroutine”.

New in version 3.8.

`LOAD_FAST`(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

`STORE_FAST`(*var_num*)

Stores TOS into the local `co_varnames[var_num]`.

`DELETE_FAST`(*var_num*)

Deletes local `co_varnames[var_num]`.

LOAD_CLOSURE(*i*)

Pushes a reference to the cell contained in slot *i* of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if *i* is less than the length of *co_cellvars*. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

LOAD_DEREF(*i*)

Loads the cell contained in slot *i* of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

LOAD_CLASSDEREF(*i*)

Much like `LOAD_DEREF` but first checks the locals dictionary before consulting the cell. This is used for loading free variables in class bodies.

New in version 3.4.

STORE_DEREF(*i*)

Stores TOS into the cell contained in slot *i* of the cell and free variable storage.

DELETE_DEREF(*i*)

Empties the cell contained in slot *i* of the cell and free variable storage. Used by the `del` statement.

New in version 3.2.

RAISE_VARARGS(*argc*)

Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise TOS` (raise exception instance or type at `TOS`)
- 2: `raise TOS1 from TOS` (raise exception instance or type at `TOS1` with `__cause__` set to `TOS`)

CALL_FUNCTION(*argc*)

Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. `CALL_FUNCTION` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Changed in version 3.6: This opcode is used only for calls with positional arguments.

CALL_FUNCTION_KW(*argc*)

Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. `CALL_FUNCTION_KW` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Changed in version 3.6: Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

`CALL_FUNCTION_EX`(*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Below that is an iterable object containing positional arguments and a callable object to call. `BUILD_MAP_UNPACK_WITH_CALL` and `BUILD_TUPLE_UNPACK_WITH_CALL` can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each “unpacked” and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

New in version 3.6.

`LOAD_METHOD`(*namei*)

Loads a method named `co_names[namei]` from TOS object. TOS is popped and method and TOS are pushed when interpreter can call unbound method directly. TOS will be used as the first argument (`self`) by `CALL_METHOD`. Otherwise, `NULL` and method is pushed (method is bound method or something else).

New in version 3.7.

`CALL_METHOD`(*argc*)

Calls a method. *argc* is number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with `LOAD_METHOD`. Positional arguments are on top of the stack. Below them, two items described in `LOAD_METHOD` on the stack. All of them are popped and return value is pushed.

New in version 3.7.

`MAKE_FUNCTION`(*argc*)

Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters’ default values
- `0x04` an annotation dictionary
- `0x08` a tuple containing cells for free variables, making a closure

- the code associated with the function (at TOS1)
- the qualified name of the function (at TOS)

`BUILD_SLICE`(*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the `slice()` built-in function for more information.

`EXTENDED_ARG`(*ext*)

Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

`FORMAT_VALUE`(*flags*)

Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows:

- `(flags & 0x03) == 0x00`: *value* is formatted as-is.
- `(flags & 0x03) == 0x01`: call `str()` on *value* before formatting it.
- `(flags & 0x03) == 0x02`: call `repr()` on *value* before formatting it.
- `(flags & 0x03) == 0x03`: call `ascii()` on *value* before formatting it.
- `(flags & 0x04) == 0x04`: pop *fmt_spec* from the stack and use it, else use an empty *fmt_spec*.

Formatting is performed using `PyObject_Format()`. The result is pushed on the stack.

New in version 3.6.

`HAVE_ARGUMENT`

This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

Changed in version 3.6: Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

`dis.`**`opname`**

Sequence of operation names, indexable using the bytecode.

`dis.`**`opmap`**

Dictionary mapping operation names to bytecodes.

`dis.`**`cmp_op`**

Sequence of all compare operation names.

`dis.`**`hasconst`**

Sequence of bytecodes that access a constant.

`dis.`**`hasfree`**

Sequence of bytecodes that access a free variable (note that ‘free’ in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes).

`dis.`**`hasname`**

Sequence of bytecodes that access an attribute by name.

`dis.`**`hasjrel`**

Sequence of bytecodes that have a relative jump target.

`dis.`**`hasjabs`**

Sequence of bytecodes that have an absolute jump target.

`dis.`**`haslocal`**

Sequence of bytecodes that access a local variable.

`dis.`**`hascompare`**

Sequence of bytecodes of Boolean operations.