18.10.1 Python Byte Code Instructions

The Python compiler currently generates the following byte code instructions.

STOP_CODE
: Indicates end-of-code to the compiler, not used by the interpreter.

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 item one position up, moves top down to position four.

DUP_TOP
: Duplicates the reference on top of the stack.

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_CONVERT
: Implements `TOS = `TOS``.

UNARY_INVERT
: Implements `TOS = ~TOS`.

GET_ITER
: Implements `TOS = iter(TOS)`.

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_DIVIDE
: Implements `TOS = TOS1 / TOS` when `from __future__ import division` is not in effect.

BINARY_FLOOR_DIVIDE
: Implements `TOS = TOS1 // TOS`.

BINARY_TRUE_DIVIDE
: Implements `TOS = TOS1 / TOS` when `from __future__ import division` is in effect.

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 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_DIVIDE

Implements in-place `TOS = TOS1 / TOS` when `from __future__ import division` is not in effect.

INPLACE_FLOOR_DIVIDE

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

INPLACE_TRUE_DIVIDE

Implements in-place `TOS = TOS1 / TOS` when `from __future__ import division` is in effect.

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`.

The slice opcodes take up to three parameters.

SLICE+0
Implements `TOS = TOS[:]`.
SLICE+1
Implements `TOS = TOS1[TOS:]`.
SLICE+2
Implements `TOS = TOS1[:TOS]`.
SLICE+3
Implements `TOS = TOS2[TOS1:TOS]`.

Slice assignment needs even an additional parameter. As any statement, they put nothing on the stack.

STORE_SLICE+0
Implements `TOS[:] = TOS1`.
STORE_SLICE+1
Implements `TOS1[TOS:] = TOS2`.
STORE_SLICE+2
Implements `TOS1[:TOS] = TOS2`.
STORE_SLICE+3
Implements `TOS2[TOS1:TOS] = TOS3`.
DELETE_SLICE+0
Implements `del TOS[:]`.
DELETE_SLICE+1
Implements `del TOS1[TOS:]`.
DELETE_SLICE+2
Implements `del TOS1[:TOS]`.
DELETE_SLICE+3
Implements `del TOS2[TOS1:TOS]`.
STORE_SUBSCR
Implements `TOS1[TOS] = TOS2`.
DELETE_SUBSCR
Implements `del TOS1[TOS]`.

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_STACK`.
PRINT_ITEM
Prints TOS to the file-like object bound to `sys.stdout`. There is one such instruction for each item in the `print` statement.
PRINT_ITEM_TO
Like `PRINT_ITEM`, but prints the item second from TOS to the file-like object at TOS. This is used by the extended print statement.
PRINT_NEWLINE
Prints a new line on `sys.stdout`. This is generated as the last operation of a `print` statement, unless the statement ends with a comma.
PRINT_NEWLINE_TO
Like `PRINT_NEWLINE`, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.
BREAK_LOOP

Terminates a loop due to a `break` statement.

CONTINUE_LOOP *target*

Continues a loop due to a `continue` statement. *target* is the address to jump to (which should be a `FOR_ITER` instruction).

LIST_APPEND

Calls `list.append(TOS1, TOS)`. Used to implement list comprehensions.

LOAD_LOCALS

Pushes a reference to the locals of the current scope on the stack. This is used in the code for a class definition: After the class body is evaluated, the locals are passed to the class definition.

RETURN_VALUE

Returns with TOS to the caller of the function.

YIELD_VALUE

Pops `TOS` and yields it from a generator.

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 *`.

EXEC_STMT

Implements `exec TOS2,TOS1,TOS`. The compiler fills missing optional parameters with `None`.

POP_BLOCK

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

END_FINALLY

Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

BUILD_CLASS

Creates a new class object. TOS is the methods dictionary, TOS1 the tuple of the names of the base classes, and TOS2 the class name.

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

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_LOCAL` 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.

DUP_TOPX *count*

Duplicate *count* items, keeping them in the same order. Due to implementation limits, *count* should be between 1 and 5 inclusive.

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_MAP *zero*

Pushes a new empty dictionary object onto the stack. The argument is ignored and set to zero by the compiler.

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*`]`. 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 byte code counter by *delta*.

JUMP_IF_TRUE *delta*

If TOS is true, increment the byte code counter by *delta*. TOS is left on the stack.

JUMP_IF_FALSE *delta*

If TOS is false, increment the byte code counter by *delta*. TOS is not changed.

JUMP_ABSOLUTE *target*

Set byte code 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_LOOP *delta*

Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

SETUP_EXCEPT *delta*

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

SETUP_FINALLY *delta*

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

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.

STORE_DEREF *i*

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

SET_LINENO *lineno*

This opcode is obsolete.

RAISE_VARARGS *argc*

Raises an exception. *argc* indicates the number of parameters to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION *argc*

Calls a function. The low byte of *argc* indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack.

MAKE_FUNCTION *argc*

Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have *argc* default parameters, which are found below TOS.

MAKE_CLOSURE *argc*

Creates a new function object, sets its *func_closure* slot, and pushes it on the stack. TOS is the code associated with the function. If the code object has N free variables, the next N items on the stack are the cells for these variables. The function also has *argc* default parameters, where are found before the cells.

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 two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

CALL_FUNCTION_VAR *argc*

Calls a function. *argc* is interpreted as in CALL_FUNCTION. The top element on the stack contains the variable argument list, followed by keyword and positional arguments.

CALL_FUNCTION_KW *argc*

Calls a function. *argc* is interpreted as in CALL_FUNCTION. The top element on the stack contains the keyword arguments dictionary, followed by explicit keyword and positional arguments.

CALL_FUNCTION_VAR_KW *argc*

Calls a function. *argc* is interpreted as in CALL_FUNCTION. The top element on the stack contains the keyword arguments dictionary, followed by the variable-arguments tuple, followed by explicit keyword and positional arguments.