Opcode	Mnemonics	Operand length	Description	Source Code
00h	STOP_CODE	0	Indicates end-of-code to the compiler, not used by the interpreter.	`# it's use in assembly code results in:` `# SystemError: unknown opcode`
01h	POP_TOP	0	Removes the top-of-stack (TOS) item.	`(lambda:1)()`
02h	ROT_TWO	0	Swaps the two top-most stack items.	`a = 1; b = 2` `(a, b) = (b, a)`
03h	ROT_THREE	0	Lifts second and third stack item one position up, moves top down to position three.	`a = 1` `(a, a, a) = (a, a, a)`
04h	DUP_TOP	0	Duplicates the reference on top of the stack.	`{'a':1}`
05h	ROT_FOUR	0	Lifts second, third and forth stack item one position up, moves top down to position four.	`x = range(6)` `x[2:4] += 'abc'`
09h	NOP	0	Do nothing code. Used as a placeholder by the bytecode optimizer.	`# not generated by compiler or removed after optimizations`
0Ah	UNARY_POSITIVE	0	Implements TOS = +TOS.	`a = 1` `a = +a`
0Bh	UNARY_NEGATIVE	0	Implements TOS = -TOS.	`a = 1` `a = -a`

0Ch	UNARY_NOT	0	Implements TOS = not TOS.	`a = 1` `a = not a`
0Dh	UNARY_CONVERT	0	Implements TOS = `TOS`.	`a = 1` `a = `a``
0Fh	UNARY_INVERT	0	Implements TOS = ~TOS.	`a = 1` `a = ~a`
12h	LIST_APPEND	0	Calls list.append(TOS1, TOS). Used to implement list comprehensions.	`a = [i*i for i in (1,2)]`
13h	BINARY_POWER	0	Implements TOS = TOS1 ** TOS.	`a = 2` `a = a ** 2`
14h	BINARY_MULTIPLY	0	Implements TOS = TOS1 * TOS.	`a = 2` `a = a * 2`
15h	BINARY_DIVIDE	0	Implements TOS = TOS1 / TOS when from __future__ import division is not in effect.	`a = 2` `a = a / 2`
16h	BINARY_MODULO	0	Implements TOS = TOS1 % TOS.	`a = 2` `a = a % 2`
17h	BINARY_ADD	0	Implements TOS = TOS1 + TOS.	`a = 2` `a = a + 2`
18h	BINARY_SUBTRACT	0	Implements TOS = TOS1	`a = 2`

			- TOS.	a = a - 2
19h	BINARY_SUBSCR	0	Implements TOS = TOS1[TOS].	a = [1] a[0]
1Ah	BINARY_FLOOR_DIVIDE	0	Implements TOS = TOS1 // TOS.	a = 2 a = a // 2
1Bh	BINARY_TRUE_DIVIDE	0	Implements TOS = TOS1 / TOS when from __future__ import division is in effect.	from __future__ import division a = 2 a = a / 2
1Ch	INPLACE_FLOOR_DIVIDE	0	Implements in-place TOS = TOS1 // TOS.	a = 1 a //= 1
1Dh	INPLACE_TRUE_DIVIDE	0	Implements in-place TOS = TOS1 / TOS when from __future__ import division is in effect.	from __future__ import division a = 1 a /= 2
1Eh	SLICE	0	Implements TOS = TOS[:].	a = [1,2,3] a = a[:]
1Fh	SLICE+1	0	Implements TOS = TOS1[TOS:].	a = [1,2,3] a = a[1:]
20h	SLICE+2	0	Implements TOS = TOS1[:TOS].	a = [1,2,3] a = a[:2]

21h	SLICE+3	0	Implements TOS = TOS2[TOS1:TOS].	`a = [1,2,3]` `a = a[1:2]`
28h	STORE_SLICE	0	Implements TOS[:] = TOS1.	`a = [1,2,3]` `a[:] = [1,2,3]`
29h	STORE_SLICE+1	0	Implements TOS1[TOS:] = TOS2.	`a = [1,2,3]` `a[1:] = [2,3]`
2Ah	STORE_SLICE+2	0	Implements TOS1[:TOS] = TOS2.	`a = [1,2,3]` `a[:2] = [1,2]`
2Bh	STORE_SLICE+3	0	Implements TOS2[TOS1:TOS] = TOS3.	`a = [1,2,3]` `a[1:2] = [2]`
32h	DELETE_SLICE	0	Implements del TOS[:].	`a = [1,2,3]` `del a[:]`
33h	DELETE_SLICE+1	0	Implements del TOS1[TOS:].	`a = [1,2,3]` `del a[1:]`

34h	DELETE_SLICE+2	0	Implements del TOS1[:TOS].	`a = [1,2,3]` `del a[:2]`
35h	DELETE_SLICE+3	0	Implements del TOS2[TOS1:TOS].	`a = [1,2,3]` `del a[1:2]`
37h	INPLACE_ADD	0	Implements in-place TOS = TOS1 + TOS.	`a = 1` `a += 1`
38h	INPLACE_SUBTRACT	0	Implements in-place TOS = TOS1 - TOS.	`a = 1` `a -= 1`
39h	INPLACE_MULTIPLY	0	Implements in-place TOS = TOS1 * TOS.	`a = 1` `a *= 1`
3Ah	INPLACE_DIVIDE	0	Implements in-place TOS = TOS1 / TOS when from __future__ import division is not in effect.	`a = 1` `a /= 1`
3Bh	INPLACE_MODULO	0	Implements in-place TOS = TOS1 % TOS.	`a = 1` `a %= 1`
3Ch	STORE_SUBSCR	0	Implements TOS1[TOS] = TOS2.	`a = []` `a[0] = 1`
3Dh	DELETE_SUBSCR	0	Implements del TOS1[TOS].	`a = [1]` `del a[0]`

3Eh	BINARY_LSHIFT	0	Implements TOS = TOS1 << TOS.	`a = 1` `a = a << 1`
3Fh	BINARY_RSHIFT	0	Implements TOS = TOS1 >> TOS.	`a = 1` `a = a >> 1`
40h	BINARY_AND	0	Implements TOS = TOS1 & TOS.	`a = 1` `a = a & 1`
41h	BINARY_XOR	0	Implements TOS = TOS1 ^ TOS.	`a = 1` `a = a ^ 1`
42h	BINARY_OR	0	Implements TOS = TOS1 | TOS.	`a = 1` `a = a | 1`
43h	INPLACE_POWER	0	Implements in-place TOS = TOS1 ** TOS.	`a = 1` `a **= 1`
44h	GET_ITER	0	Implements TOS = iter(TOS).	`for a in (1,2): pass`
46h	PRINT_EXPR	0	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.	`# used only in interactive mode`
47h	PRINT_ITEM	0	Prints TOS to the file-like object bound to sys.stdout. There is one such instruction for each item in the print statement.	`print "hello world!"`
48h	PRINT_NEWLINE	0	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`

			comma.	
49h	PRINT_ITEM_TO	0	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.	`import sys` `print >> sys.stdout, "hello world",`
4Ah	PRINT_NEWLINE_TO	0	Like PRINT_NEWLINE, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.	`import sys` `print >> sys.stdout`
4Bh	INPLACE_LSHIFT	0	Implements in-place TOS = TOS1 << TOS.	`a = 1` `a <<= 1`
4Ch	INPLACE_RSHIFT	0	Implements in-place TOS = TOS1 >> TOS.	`a = 1` `a >>= 1`
4Dh	INPLACE_AND	0	Implements in-place TOS = TOS1 & TOS.	`a = 1` `a &= 1`
4Eh	INPLACE_XOR	0	Implements in-place TOS = TOS1 ^ TOS.	`a = 1` `a ^= 1`
4Fh	INPLACE_OR	0	Implements in-place TOS = TOS1 | TOS.	`a = 1` `a |= 1`
50h	BREAK_LOOP	0	Terminates a loop due to a break statement.	`for a in (1,2): break`

51h	WITH_CLEANUP	0	???	`from __future__ import with_statement`
				
				`with open("1.txt") as f:`
				`  print f.read()`

52h	LOAD_LOCALS	0	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.	`class a: pass`
53h	RETURN_VALUE	0	Returns with TOS to the caller of the function.	`# empty file`
54h	IMPORT_STAR	0	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 *.	`from sys import *`
55h	EXEC_STMT	0	Implements exec TOS2,TOS1,TOS. The compiler fills missing optional parameters with None.	`exec("print 'hello world'", globals(), locals())`

56h	YIELD_VALUE	0	Pops TOS and yields it from a generator.	`def foo():`
				`  print 'hello'`
				`  yield 1`
				`  print 'world'`
				`  yield 2`
				
				`a = foo()`
				`print a.next()`
				`print a.next()`

57h	POP_BLOCK	0	Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.	`for a in (1,2): break`

58h	END_FINALLY	0	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.	`try:`
				`  a = 1`
				`except ValueError:`
				`  a = 2`
				`finally:`
				`  a = 3`

59h	BUILD_CLASS	0	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.	`class a: pass`
5Ah	STORE_NAME	2	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.	`a = 1`
5Bh	DELETE_NAME	2	Implements del name, where /namei/ is the index into co_names attribute of the code object.	`a = 1` `del a`
5Ch	UNPACK_SEQUENCE	2	Unpacks TOS into /count/ individual values, which are put onto the stack right-to-left	`(a, b) = "ab"`

			right-to-left.	
5Dh	FOR_ITER	2	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/.	`for i in (1,2): pass`
5Fh	STORE_ATTR	2	Implements TOS.name = TOS1, where /namei/ is the index of name in co_names.	`import sys` `sys.stderr = sys.stdout`
60h	DELETE_ATTR	2	Implements del TOS.name, using /namei/ as index into co_names.	`import sys` `del sys.stderr`
61h	STORE_GLOBAL	2	Works as STORE_NAME(/namei/), but stores the name as a global.	`global a` `a = 1`
62h	DELETE_GLOBAL	2	Works as DELETE_NAME(/namei/), but deletes a global name.	`global a` `del a`

63h	DUP_TOPX	None	Duplicate /count/ items, keeping them in the same order. Due to implementation limits, count should be between 1 and 5 inclusive.	`a = 0`
				`b = [0]`
				`b[a] += 1`

64h	LOAD_CONST	2	Pushes "co_consts[/consti/]" onto the stack.	`a = 1`
65h	LOAD_NAME	2	Pushes the value associated with "co_names[/namei/]" onto the stack.	`a = 1` `a = a`
66h	BUILD_TUPLE	2	Creates a tuple consuming /count/ items from the stack, and pushes the resulting tuple onto the stack.	`a = 1;` `a = (a, a)`
67h	BUILD_LIST	2	Works as BUILD_TUPLE(/count/), but creates a list.	`[1,2,3]`

68h	BUILD_MAP	2	Pushes a new empty dictionary object onto the stack. The argument is ignored and set to /zero/ by the compiler.	`{"a":1,"b":2}`
69h	LOAD_ATTR	2	Replaces TOS with getattr(TOS, co_names[/namei/]).	`[].sort()`
6Ah	COMPARE_OP	2	Performs a Boolean operation. The operation name can be found in cmp_op[/opname/].	`a = 1 == 2`
6Bh	IMPORT_NAME	2	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 new`
6Ch	IMPORT_FROM	2	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.	`from dis import opmap`
6Eh	JUMP_FORWARD	2	Increments byte code counter by /delta/.	`if 1 == 2: pass` `else: pass`
6Fh	JUMP_IF_FALSE	2	If TOS is false, increment the byte code counter by /delta/. TOS is not changed.	`if 1 == 2: pass` `else: pass`
70h	JUMP_IF_TRUE	2	If TOS is true, increment the byte code counter by /delta/. TOS is left on the stack.	`if not(1 == 2): pass` `else: pass`
71h	JUMP_ABSOLUTE	2	Set byte code counter to /target/.	`for i in (1,2): pass`

74h	LOAD_GLOBAL	2	Loads the global named co_names[/namei/] onto the stack.	`global a`
				`a = 1`
				`a = a`

77h	CONTINUE_LOOP	2	Continues a loop due to a continue statement. /target/ is the address to jump to (which should be a FOR_ITER instruction).	`for x in (1,2):`
				`  try: continue`
				`  except: pass`

78h	SETUP_LOOP	2	Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of /delta/ bytes.	`while 0 > 1: pass`

79h	SETUP_EXCEPT	2	Pushes a try block from a try-except clause onto the block stack. /delta/ points to the first except block.	`try:`
				`  a = 1`
				`except ValueError:`
				`  a = 2`
				`finally:`
				`  a = 3`

7Ah	SETUP_FINALLY	2	Pushes a try block from a try-except clause onto the block stack. /delta/ points to the finally block.	`try:`
				`  a = 1`
				`except ValueError:`
				`  a = 2`
				`finally:`
				`  a = 3`

7Ch	LOAD_FAST	2	Pushes a reference to the local co_varnames[/var_num/] onto the stack.	`def f():`
				`  a = 1`
				`  a = a`

7Dh	STORE_FAST	2	Stores TOS into the local co_varnames[/var_num/].	`def f():`
				`  a = 1`
				`  a = a`

7Eh	DELETE_FAST	2	Deletes local co_varnames[/var_num/].	`def f():`
				`  a = 1`
				`  del a`

82h	RAISE_VARARGS	2	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.	`raise ValueError`
83h	CALL_FUNCTION	2	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.	`def f(): pass` `f()`
			Pushes a new function object on the stack. TOS	

84h	MAKE_FUNCTION	2	is the code associated with the function. The function object is defined to have /argc/ default parameters, which are found below TOS.	`def f(): pass`
85h	BUILD_SLICE	2	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.	`a = [1,2,3,4]` `b = a[::-1]`

86h	MAKE_CLOSURE	2	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.	`def f():`
				`  a = 1`
				`  def g():`
				`  return a + 1`
				`  return g()`
				
				`print f()`

87h	LOAD_CLOSURE	2	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)].	`def f():`
				`  a = 1`
				`  def g():`
				`  return a + 1`
				`  return g()`
				
				`print f()`

88h	LOAD_DEREF	2	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.	`def f():`
				`  a = 1`
				`  def g():`
				`  return a + 1`
				`  return g()`
				
				`print f()`

89h	STORE_DEREF	2	Stores TOS into the cell contained in slot /i/ of the cell and free variable storage.	`def f():`
				`  a = 1`
				`  def g():`
				`  return a + 1`
				`  return g()`
				
				`print f()`

8Ch	CALL_FUNCTION_VAR	2	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.	`def f(a,b): pass`
				`a = (1,2)`
				`f(*a)`

8Dh	CALL_FUNCTION_KW	2	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.	`def f(a,b): pass`
				`a = {"a":1,"b":2}`
				`f(**a)`

8Eh	CALL_FUNCTION_VAR_KW	2	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.	`def f(a,b,c): pass`
				`a = {"b":1,"c":2}`
				`b = (3,)`
				`f(*b, **a)`

8Fh	EXTENDED_ARG	2	Support for opargs more than 16 bits long.	`a=(0,1,2,3, ... ,65535)`