Published in Towards Data Science

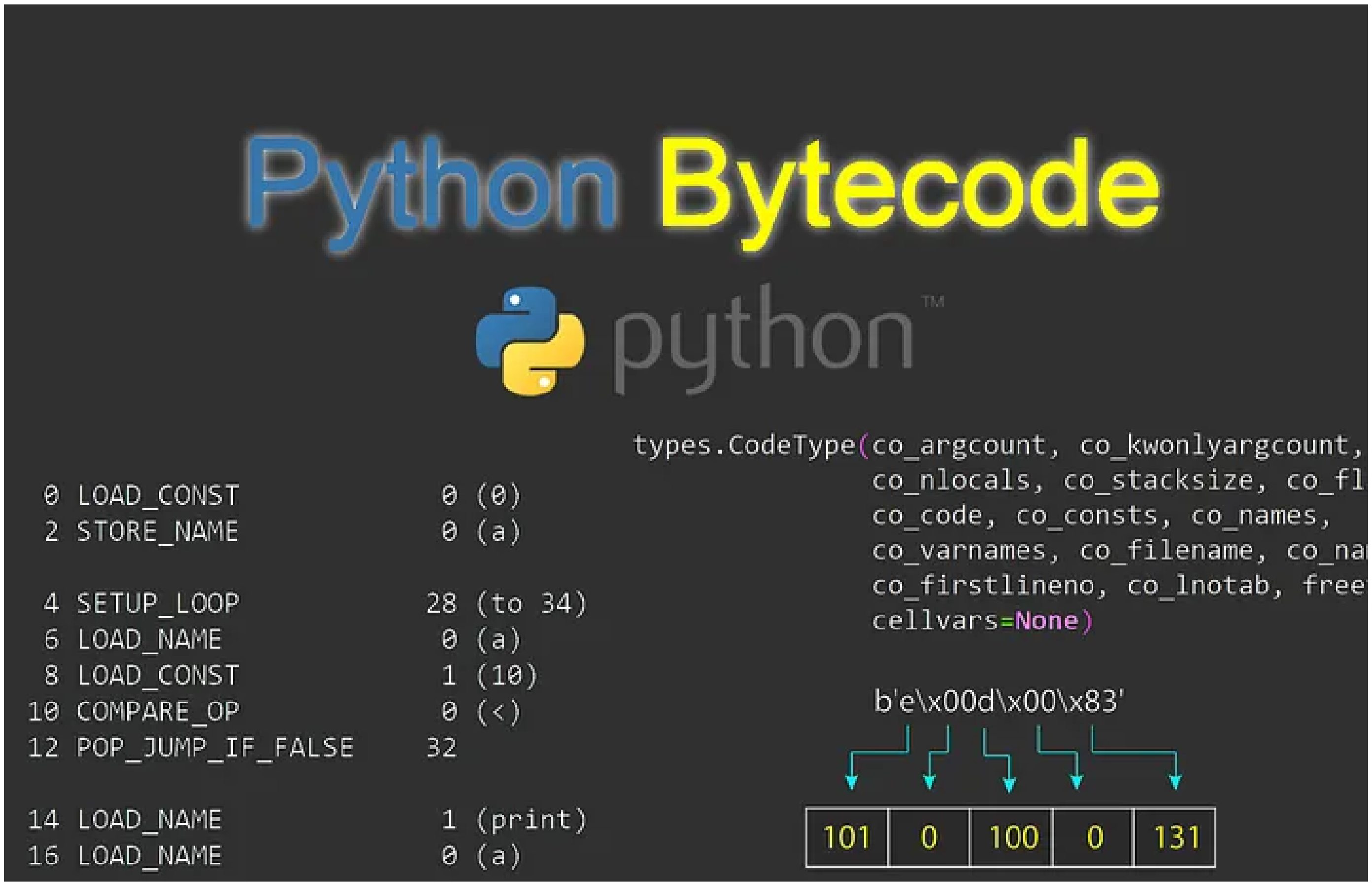
Reza Bagheri Follow

Mar 6, 2020 · 44 min read · · Listen

Save

Understanding Python Bytecode

Learn about disassembling Python bytecode



```
 0 LOAD_CONST            0 (0)
 2 STORE_NAME            0 (a)

 4 SETUP_LOOP           28 (to 34)
 6 LOAD_NAME             0 (a)
 8 LOAD_CONST            1 (10)
10 COMPARE_OP            0 (<)
12 POP_JUMP_IF_FALSE    32

14 LOAD_NAME             1 (print)
16 LOAD_NAME             0 (a)
```

```
types.CodeType(co_argcount, co_kwonlyargcount,
               co_nlocals, co_stacksize, co_fl
               co_code, co_consts, co_names,
               co_varnames, co_filename, co_na
               co_firstlineno, co_lnotab, free
               cellvars=None)
```

```
b'e\x00d\x00\x83'
```

101	0	100	0	131

The source code of a programming language can be executed using an interpreter or a compiler. In a compiled language, a compiler will translate the source code directly into binary machine code. This machine code is specific to that target machine since each machine can have a different operating system and hardware. After compilation, the target machine will directly run the machine code.

In an interpreted language, the source code is not directly run by the target machine. There is another program called the interpreter that reads and executes the source code directly. The interpreter, which is specific to the target machine, translates each statement of the source code into machine code and runs it.

Python is usually called an interpreted language, however, it combines compiling and interpreting. When we execute a source code (a file with a `.py` extension), Python first compiles it into a bytecode. The bytecode is a low-level platform-independent representation of your source code, however, it is not the binary machine code and cannot be run by the target machine directly. In fact, it is a set of instructions for a virtual machine which is called the Python Virtual Machine (PVM).

After compilation, the bytecode is sent for execution to the PVM. The PVM is an interpreter that runs the bytecode and is part of the Python system. The bytecode is platform-independent, but PVM is specific to the target machine. The default implementation of the Python programming language is CPython which is written in the C programming language. CPython compiles the python source code into the bytecode, and this bytecode is then executed by the CPython virtual machine.

Generating bytecode files

In Python, the bytecode is stored in a `.pyc` file. In Python 3, the bytecode files are stored in a folder named `__pycache__`. This folder is automatically created when you try to import another file that you created:

```
import file_name
```

However, it will not be created if we don’t import another file in the source code. In that case, we can still manually create it. To compile the individual files `file_1.py` to `file_n.py` from the command line, we can write:

```
python -m compileall file_1.py ... file_n.py
```

All the generated `pyc` files will be stored in the `__pycache__` folder. If you provide no file names after `compileall`, it will compile all the python source code files in the current folder.

We can also use the `compile()` function to compile a string that contains the Python source code. The syntax of this function is:

`compile(`*`source`*`, `*`filename`*`, `*`mode`*`, `*`flag`*`, `*`dont_inherit`*`, `*`optimize`*`)`

We only focus on the first three arguments which are required (the others are optional). `source` is the source code to compile which can be a String, a Bytes object, or an AST object. `filename` is the name of the file that the source code comes from. If the source code does not come from a file, you can write whatever you like or leave an empty string. `mode` can be:

`'exec'` : accepts Python source code in any form (any number of statements or blocks). It compiles them into a bytecode that finally returns `None`

`'eval'` : accepts a single expression and compiles it into a bytecode that finally returns the value of that expression

`'single'` : only accepts a single statement (or multiple statements separated by `;`). If the last statement is an expression, then the resulting bytecode prints the `repr()` of the value of that expression to the standard output.

For example, to compile some Python statements we can write:

```
s='''
a=5
a+=1
print(a)
```

```
'''
compile(s, "", "exec")
```

or equivalently write:

```
compile("a=5 \na+=1 \nprint(a)", "", "exec")
```

To evaluate an expression we can write:

```
compile("a+7", "", "eval")
```

This mode gives an error if you don't have an expression:

```
# This does not work:
compile("a=a+1", "", "eval")
```

Here `a=a+1` is not an expression and does not return anything, so we cannot use the `eval` mode. However, we can use the `single` mode to compile it:

```
compile("a=a+1", "", "single")
```

But what is returned by `compile`? When you run the `compile` function, Python returns:

```
<code object <module> at 0x000001A1DED95540, file "", line 1>
```

So what the `compile` function is returning is a *code object* (the address after `at` can be different on your machine).

Code object

The `compile()` function returns a Python code object. Everything in Python is an object. For example we you define an integer variable, its value is stored in an `int` object and you can easily check its type using the `type()` function:

```
a = 5
type(a)  # Output is: int
```

In a similar way, the bytecode generated by the compile function is stored in the `code` object.

```
c = compile("a=a+1", "", "single")
type(c)  # Output is: code
```

The code object contains not only the bytecode but also some other information necessary for the CPython to run the bytecode (they will be discussed later). A code object can be executed or evaluated by passing it to the `exec()` or `eval()` function. So we can write:

```
exec(compile("print(5)", "", "single"))  # Output is: 5
```

When you define a function in Python, it creates a code object for it and you can access it using the `__code__` attribute. For example, we can write:

```
def f(n):
    return n
f.__code__
```

And the output will be:

```
<code object f at 0x000001A1E093E660, file "<ipython-input-61-
```

```
88c7683062d9>", line 1>
```

Like any other objects the code object has some attributes, and to get the bytecode stored in a code object, you can use its `co_code` attribute:

```
c = compile("print(5)", "", "single")
c.co_code
```

The output is:

```
b'e\x00d\x00\x83\x01F\x00d\x01S\x00'
```

The result is a *bytes literal* which is prefixed with `b'`. It is an immutable sequence of bytes and has a type of `bytes`. Each byte can have a decimal value of 0 to 255. So a bytes literal is an immutable sequence of integers between 0 to 255. Each byte can be shown by an ASCII character whose character code is the same as the byte value or it can be shown by a leading `\x` followed by two characters. The leading `\x` escape means that the next two characters are interpreted as hex digits for the character code. For example:

```
print(c.co_code[0])
chr(c.co_code[0])
```

gives:

```
101
'e'
```

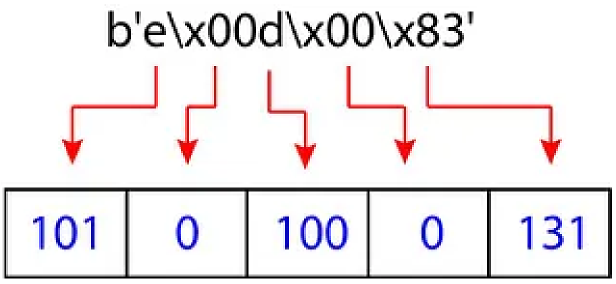
since the first element has the decimal value of 101 and can be shown with the character `e` whose ASCII character code is 101. Or:

```
print(c.co_code[4])
chr(c.co_code[4])
```

gives:

```
131
'\x83'
```

since the 4th element has the decimal value of 131. The hexadecimal value of 131 is 83. So this byte can be shown with a character whose character code is `\x83`.



These sequences of bytes can be interpreted by CPython, but they are not human-friendly. So we need to understand how these bytes are mapped to the actual instructions that will be executed by CPython. In the next section, we are going to disassemble the byte code into some human-friendly instruction to see how the bytecode is executed by CPython.

Bytecode details

Before going into further details, it is important to note that the implementation detail of Bytecode usually changes between versions of Python. So what you see in this article may not be valid for all versions of Python. In fact, it includes the changes that happened in version 3.6, and some of the details may not be valid for older versions. The code in this article has been tested with Python 3.7.

The bytecode can be thought of as a series of instructions or a low-level program for the Python interpreter. After version 3.6, Python uses 2 bytes for each instruction. One byte is for the code of that instruction which is called an *opcode*, and one byte is reserved for its argument which is called the *oparg*. Each opcode has a human-friendly name which is called the *opname*. The bytecode instructions have a general format like this:

```
opcode oparg
opcode oparg
  .
  .
  .
```

We already have the opcodes in our bytecode, and we just need to map them to their corresponding opname. There is a module called `dis` which can help with that. In this module, there is a list called `opname` which stores all the opnames. The *i*-th element of this list gives the opname for an instruction whose opcode is equal to *i*.

Some instructions do not need an argument, so they ignore the byte after the opcode. The opcodes which have a value below a certain number ignore their argument. This value is stored in `dis.HAVE_ARGUMENT` and is currently equal to 90. So the opcodes >= `dis.HAVE_ARGUMENT` have an argument, and the opcodes < `dis.HAVE_ARGUMENT` ignore it.

For example, suppose that we have a short bytecode `b'd\x00Z\x00d\x01S\x00'` and we want to disassemble it. This bytecode represents a sequence of four bytes. We can easily show their decimal value:

```
bytecode = b'd\x00Z\x00d\x01S\x00'
for byte in bytecode:
    print(byte, end=' ')
```

The output will be:

```
100 0 90 0 100 1 83 0
```

The first two bytes of the bytecode is `100 0`. The first byte is the opcode. To get its opname we can write (`dis` should be imported first):

```
dis.opname[100]
```

and the result is `LOAD_CONST`. Since the opcode is bigger than `dis.HAVE_ARGUMENT`, it has an oparg which is the second byte `0`. So `100 0` translates into:

```
LOAD_CONST 0
```

The last two bytes in the bytecode are `83 0`. Again we write `dis.opname[83]` and the result is `RETURN_VALUE`. 83 is lower than 90 (`dis.HAVE_ARGUMENT`), so this opcode ignores the oparg, and `83 0` is disassembled into:

```
RETURN_VALUE
```

In addition, some of the instructions can have an argument too big to fit into the default one byte. There is a special opcode `144` to handle these instructions. Its opname is `EXTENDED_ARG`, and it is also stored in `dis.EXTENDED_ARG`. This opcode prefixes any opcode which has an argument bigger than one byte. For example, suppose that we have the opcode 131 (its opname is `CALL_FUNCTION`) and its oparg needs to be 260. So it should be:

```
CALL_FUNCTION 260
```

However, the maximum number that a byte can store is 255, and 260 does not fit into a byte. So this opcode is prefixed with `EXTENDED_ARG`:

```
EXTENDED_ARG 1
CALL_FUNCTION 4
```

When the interpreter executes `EXTENDED_ARG`, its oparg (which is 1) is left-shifted by eight bits and stored in a temporary variable. Let's call it `extended_arg` (do not confuse it with the opname `EXTENDED_ARG`):

```
extened_arg = 1 << 8  # same as 1 * 256
```

So the binary value `0b1` (the binary value of 1) is converted to `0b100000000`. This is like multiplying 1 by 256 in the decimal system and `extened_arg` will be equal to 256. Now we have two bytes in `extened_arg`. When the interpreter reaches to the next instruction, this two-byte value is added to its oparg (which is 4 here) using a bitwise `or`.

```
extened_arg = extened_arg | 4
# Same as extened_arg += 4
```

This is like adding the value of the oparg to `extened_arg`. So now we have:

```
extened_arg = 256 + 4 = 260
```

and this value will be used as the actual oparg of `CALL_FUNCTION`. So, in fact,

```
EXTENDED_ARG 1
CALL_FUNCTION 4
```

is interpreted as:

```
EXTENDED_ARG 1
CALL_FUNCTION 260
```

For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

Now we can focus on the oparg itself. What does it mean? Actually the meaning of each oparg depends on its opcode. As mentioned before, the code object stores some information other than the bytecode. This information can be accessed using the different attributes of the code object, and we need some of these attributes to decipher the meaning of each oparg. These attributes are: `co_consts` , `co_names` , `co_varnames` , `co_cellvars` and `co_freevars` .

Code object attributes

I am going to explain the meaning of these attributes using an example. Suppose that you have the code object of this source code:

```
# Listing 1
s = '''
a = 5
b = 'text'
def f(x):
    return x
f(5)
'''
c=compile(s, "", "exec")
```

Now we can check what is stored in each of these attributes:

1- `co_consts` : A tuple containing the literals used by the bytecode. Here `c.co_consts` returns:

```
(5, 'text', <code object f at 0x00000218C297EF60, file "", line 4>,
'f', None)
```

So the literals `5` and `'text'` and the name of the function `'f'` are all stored in this tuple. In addition, the body of the function `f` is stored in a separate code object and is treated like a literal which is also stored in this tuple. Remember that the `exec` mode in `compile()` generates a bytecode that finally returns `None`. This `None` value is also stored as a literal. In fact, if you compile an expression in `eval` mode like this:

```
s = "3 * a"
c1 = compile(s, "", "eval")
c1.co_consts   # Output is (3,)
```

`None` won't be included in the `co_consts` tuple anymore. The reason is that this expression returns its final value not `None`.

If you try to get the `co_const` for the object code of a function like:

```
def f(x):
    a = x * 2
    return a
f.__code__.co_consts
```

The result will be `(None, 2)`. In fact, the default return value for a function is `None`, and it is always added as a literal. As I explain later, for the sake of efficiency, Python does not check if you are always going to reach a `return` statement or not, so `None` is always added as the default return value.

2- `co_names`: A tuple containing the names used by the bytecode which can be global variables, functions, and classes or also attributes loaded from objects. For example for the object code in Listing 1, `c.co_names` gives:

```
('a', 'b', 'f')
```

3- `co_varnames`: A tuple containing the local names used by the bytecode (arguments first, then the local variables). If we try it for the object code of Listing 1, it gives an empty tuple. The reason is that the local names are defined inside functions, and the function inside Listing 1 is stored as a separate code object, so its local variables will not be included in this tuple. To access the local variables of a function, we should use this attribute for the code object of that function. So we first write this source code:

```
def f(x):
    z = 3
    t = 5
    def g(y):
        return t*x + y
    return g
a = 5
b = 1
h = f(a)
```

Now `f.__code__` gives the code object of `f`, and `f.__code__.co_varnames` gives:

```
('x', 'z', 'g')
```

Why `t` is not included? The reason is that `t` is not a local variable of `f`. It is a nonlocal variable since it is accessed by the closure `g` inside `f`. In fact, `x` is also a nonlocal variable, but since it is the function's argument, it is always included in this tuple. To learn more about closures and nonlocal variables you can refer to this article.

4- `co_cellvars`: A tuple containing the names of nonlocal variables. These are the local variables of a function accessed by its inner functions. So `f.__code__.co_cellvars` gives:

```
('t', 'x')
```

5- `co_freevars`: A tuple containing the names of free variables. Free variables are the local variables of an outer function which are accessed by its inner function. So this attribute should be used with the code object of the closure `h`. Now `h.__code__.co_freevars` gives the same result:

```
('t', 'x')
```

Now that we are familiar with these attributes, we can go back to the opargs. The meaning of each oparg depends on its opcode. We have different categories of opcodes, and for each category, the oparg has a different meaning. In the `dis` module, there are some lists that give the opcodes for each category:

1- `dis.hasconst` : This list is equal to [100]. So only the opcode 100 (its opname is LOAD_CONST) is in the category of `hasconst` . The oparg of this opcode gives the index of an element in the `co_consts` tuple. For example in the bytecode of Listing 1, if we have:

```
LOAD_CONST  1
```

then the oparg is the element of `co_consts` whose index is 1. So we should replace `1` with `co_consts[1]` which is equal to `'text'` . So the instruction will be interpreted as:

```
LOAD_CONST  'text'
```

Similarly, there are some other lists in the `dis` module that define the other categories for the opcodes:

2- `dis.hasname` : The oparg for the opcodes in this list, is the index of an element in `co_names`

3- `dis.haslocal` : The oparg for the opcodes in this list, is the index of an element in `co_varnames`

4- `dis.hasfree` : The oparg for the opcodes in this list, is the index of an element in `co_cellvars + co_freevars`

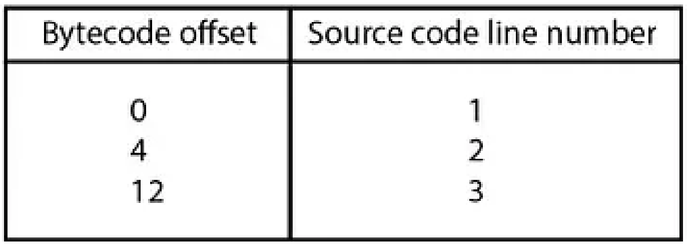
5- `dis.hascompare` : The oparg for the opcode in this list, is the index of an element of the tuple `dis.cmp_op` . This tuple contains the comparison and membership operators like `<` or `==`

6- `dis.hasjrel` : The oparg for the opcodes in this list, should be replaced with `offset + 2 + oparg` where `offset` is the index of the byte in the bytecode sequence which represents the opcode.

The code object has one more important attribute that should be discussed here. It is called `co_lnotab` which stores the line number information of the bytecode. This is an array of signed bytes stored in a bytes literal and is used to map the bytecode offsets to the source code line numbers. Let me explain it by an example. Suppose that your source code has only three lines and it has been compiled into a bytecode which has 24 bytes:

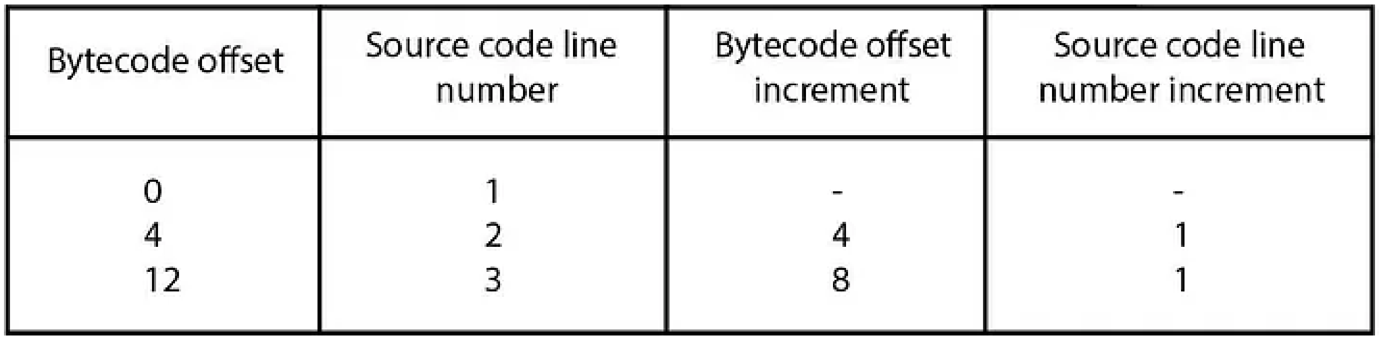
```
1          0 LOAD_CONST               0
           2 STORE_NAME               0

2          4 LOAD_NAME                0
           6 LOAD_CONST               1
           8 INPLACE_ADD
          10 STORE_NAME               0

3         12 LOAD_NAME                1
          14 LOAD_NAME                0
          16 CALL_FUNCTION            1
          18 POP_TOP
          20 LOAD_CONST               2
          22 RETURN_VALUE
```

Now we have a mapping from bytecode offsets to line numbers like this table:



Bytecode offset	Source code line number
0	1
4	2
12	3

The bytecode offset always starts at 0. The code object has an attribute named `co_firstlineno` which gives the line number for the offset zero. For this example `co_firstlineno` is equal to 1. Instead of storing the offset and line numbers literally, Python stores only the increments from one row to the next (excluding the first row). So the previous table turns into:



Bytecode offset	Source code line number	Bytecode offset increment	Source code line number increment
0	1	-	-
4	2	4	1
12	3	8	1

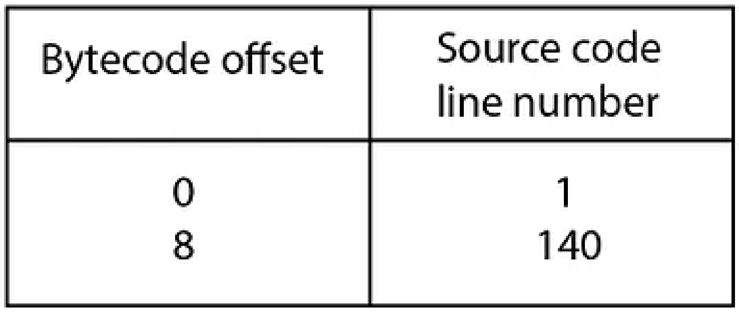
These two increment columns are zipped together in a sequence like this:

```
4 1 8 1
```

Each number is stored in a byte and the whole sequence is stored as a bytes literal in the `co_lnotab` of the code object. So if you check the value of `co_lnotab` you get:

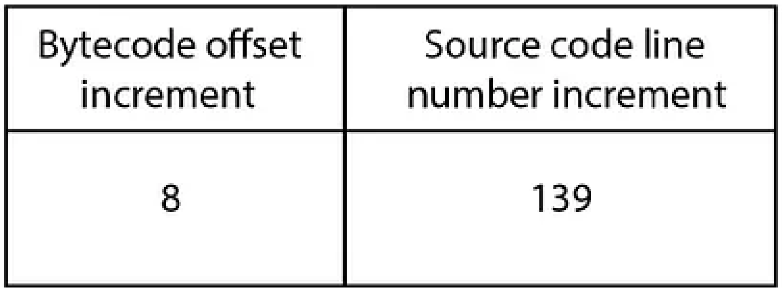
```
b'\x04\x01\x08\x01'
```

which is the bytes literal for the previous sequence. So by having the attributes `co_lnotab` and `co_firstlineno` you can retrieve the mapping from the bytecode offsets to the source code line numbers. `co_lnotab` is a sequence of signed bytes. So each signed byte in it can take a value from -128 to 127 (These values are still stored in a byte which takes 0 to 255. But a value between 128 and 255 is considered a negative number). A negative increment means that the line number is decreasing (this feature is used in optimizers). But what happens if the line increment is bigger than 127? In that case, the line increment will be split into 127 and some extra bytes and those extra bytes will be stored with a zero offset increment (if it is smaller than -128, it will be split into -128 and some extra bytes with a zero offset increment). For example, suppose that the bytecode offset versus the line number is like this:



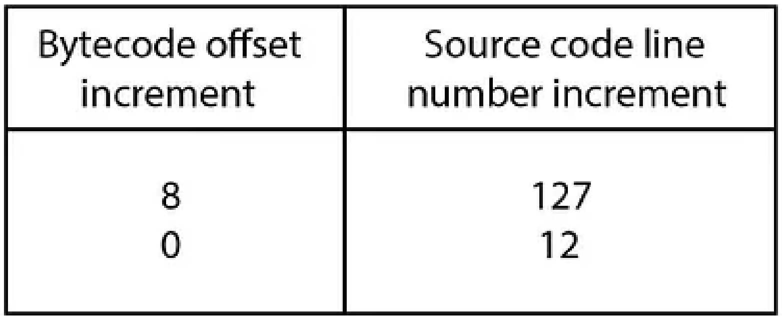
Bytecode offset	Source code line number
0 8	1 140

Then the offset increment versus the line number increment should be:



Bytecode offset increment	Source code line number increment
8	139

139 is equal to 127 + 12. So the previous row should be written as:



Bytecode offset increment	Source code line number increment
8 0	127 12

and should be stored as `8 127 0 12`. So the value of `co_lnotab` will be: `b'\x08\x7f\x00\x0c'`.

Disassembling the bytecode

Now that we are familiar with the bytecode structure, we can write a simple disassembler program. We first write a generator function to unpack each instruction and yield the offset, opcode, and oparg:

```python
def unpack_op(bytecode):
    extended_arg = 0
    for i in range(0, len(bytecode), 2):
        opcode = bytecode[i]
        if opcode >= dis.HAVE_ARGUMENT:
            oparg = bytecode[i+1] | extended_arg
            extended_arg = (oparg << 8) if opcode == dis.EXTENDED_ARG else 0
        else:
            oparg = None
        yield (i, opcode, oparg)
```

BC_Listing1.py hosted with ❤ by GitHub view raw

This function reads the next pair of bytes from the bytecode. The first byte is the opcode. By comparing this opcode with `dis.HAVE_ARGUMENT`, the function decides if it should take the second byte as the oparg or ignore it. The value of `extended_arg` will be added to oparg using the bitwise or (`|`). Initially, it is zero and has no effect on the oparg. If the opcode is equal to `dis.EXTENDED_ARG`, its oparg will be left-shifted by eight bits and stored in a temporary variable called `extended_arg`.

In the next iteration, this temporary variable will be added to the next oparg and adds one byte to it. This process continues if the next opcode is `dis.EXTENDED_ARG` again, and each time adds one byte to `extended_arg`. Finally when it reaches a different opcode, `extended_arg` will be added to its oparg and set back to zero.

The `find_linestarts` function returns a dictionary that contains the source code line number for each bytecode offset.

```python
def find_linestarts(codeobj):
    byte_increments = codeobj.co_lnotab[0::2]
    line_increments = codeobj.co_lnotab[1::2]
    byte = 0
    line = codeobj.co_firstlineno
    linestart_dict = {byte: line}
    for byte_incr, line_incr in zip(byte_increments,
                                    line_increments):
        byte += byte_incr
        if line_incr >= 0x80:
            line_incr -= 0x100
        line += line_incr
        linestart_dict[byte]=line
    return linestart_dict
```

BC_Listing2.py hosted with ❤ by GitHub view raw

It first divided the `co_lnotab` bytes literal into two sequences. One is the offset increments and the other is the line number increments. The line number for offset `0` is in `co_firstlineno`. The increments are added to these two numbers to get the bytecode offset and its corresponding line number. If the line number increment is equal or bigger than 128 (0x80), it will be considered a decrement.

The `get_argvalue` function returns the human-friendly meaning of each oparg. It first checks to which category the opcode belongs and then figures out what the oparg is referring to.

```python
def get_argvalue(offset, codeobj, opcode, oparg):
    constants= codeobj.co_consts
    varnames = codeobj.co_varnames
    names = codeobj.co_names
    cell_names = codeobj.co_cellvars + codeobj.co_freevars
    argval = None
    if opcode in dis.hasconst:
        if constants is not None:
            argval = constants[oparg]
            if type(argval)==str or argval==None:
                 argval = repr(argval)
    elif opcode in dis.hasname:
        if names is not None:
            argval = names[oparg]
    elif opcode in dis.hasjrel:
        argval = offset + 2 + oparg
        argval = "to " + repr(argval)
    elif opcode in dis.haslocal:
        if varnames is not None:
            argval = varnames[oparg]
    elif opcode in dis.hascompare:
        argval = dis.cmp_op[oparg]
    elif opcode in dis.hasfree:
        if cell_names is not None:
            argval = cell_names[oparg]
    return argval
```

BC_Listing3.py hosted with ❤ by GitHub view raw

The `findlabels` function finds all the offsets in the bytecode which are jump targets and returns a list of these offsets. The jump targets will be discussed in the next section.

```python
def findlabels(codeobj):
    bytecode = codeobj.co_code
    labels = []
    for offset, opcode, oparg in unpack_op(bytecode):
            if opcode in dis.hasjrel:
                label = offset + 2 + oparg
            elif opcode in dis.hasjabs:
                label = oparg
            else:
                continue
            if label not in labels:
                labels.append(label)
    return labels
```

BC_Listing4.py hosted with ❤ by GitHub view raw

Now we can use all these functions to disassemble the bytecode. The `dissassemble` function takes a code object and disassembles it:

It will first unpack the offset, opcode and oparg for each pair of bytes in the bytecode of the code object. Then it finds the corresponding source code line numbers, and checks if the offset is a jump target. Finally, it finds the opname and the meaning of the oparg and prints all the information. As mentioned before each function definition is stored in a separate code object. So at the end the function calls itself recursively to disassemble all the function definitions in the bytecode. Here is an example of using this function. Initially, we have this source code:

```
a=0
while a<10:
    print(a)
    a += 1
```

We first store it in a string and compile it to get the object code. Then we use the `disassemble` function to disassemble its bytecode:

```
s='''a=0
while a<10:
    print(a)
    a += 1
'''
c=compile(s, "", "exec")
disassemble(c)
```

The output is:

```
1         0 LOAD_CONST               0 (0)
          2 STORE_NAME               0 (a)

2         4 SETUP_LOOP              28 (to 34)
    >>    6 LOAD_NAME                0 (a)
          8 LOAD_CONST               1 (10)
         10 COMPARE_OP               0 (<)
         12 POP_JUMP_IF_FALSE       32

3        14 LOAD_NAME                1 (print)
         16 LOAD_NAME                0 (a)
         18 CALL_FUNCTION            1
         20 POP_TOP

4        22 LOAD_NAME                0 (a)
         24 LOAD_CONST               2 (1)
         26 INPLACE_ADD
         28 STORE_NAME               0 (a)
         30 JUMP_ABSOLUTE            6
    >>   32 POP_BLOCK
    >>   34 LOAD_CONST               3 (None)
         36 RETURN_VALUE
```

So 4 lines of source code are converted into 38 bytes of bytecode or 19 lines of bytecode. In the next section, I will explain the meaning of these instructions and how they will be interpreted by CPython.

The module `dis` has a function named `dis()` which can disassemble the code object similarly. In fact, the `disassmble` function in this article is a simplified version of `dis.dis` function. So instead of writing, `disassemble(c)` we could write `dis.dis(c)` to get a similar output.

Disassembling a pyc file

As mentioned before, when the source code is compiled, the bytecode is stored in a `pyc` file. This bytecode can be disassembled in a similar way. However, it is important to mention that the `pyc` file contains some metadata plus the code object in *marshal* format. The marshal format is used for Python's internal object serialization. The size of the metadata depends on the Python version, and for version 3.7 it is 16 bytes. So when you read the `pyc` file, first you should read the metadata, and then load the code object using the `marshal` module. For example, to disassemble a `pyc` file named `u1.cpython-37.pyc` in the `__pycache__` folder we can write:

Bytecode operations

So far we learned how to disassemble the bytecode instructions. We can now focus on the meaning of these instructions and how they are executed by CPython. CPython which is the default implementation of Python uses a *stack-based* virtual machine. So first we should get familiar with the stack.

Stack and heap

Stack is a data structure with a LIFO (Last In First Out) order. It has two principal operations:

- push: adds an element to the stack
- pop: removes the most recently added element

So the last element added or pushed to the stack is the first element to be removed or popped. The advantage of using stack to store data is that memory is managed for you. Reading from and writing to stack is very fast, however, the size of stack is limited.

Data in Python is represented as objects stored on a private heap. Accessing the data on heap is a bit slower compared to stack, however, the size of heap is only limited by the size of virtual memory. The elements of heap have no dependencies with each other and can be accessed randomly at any time. Everything in Python is an object and objects are always stored on the heap. It's only the reference (or the pointer) to the object that is stored in the stack.

CPython uses the *call stack* for running a Python program. When a function is called in Python, a new *frame* is pushed onto the call stack, and every time a function call returns, its frame is popped off. The module in which the program runs has the bottom-most frame which is called the global frame or the module frame.

Each frame has an *evaluation stack* where the execution of a Python function occurs. The function arguments and its local variables are pushed into this evaluation stack. CPython uses the evaluation stack to store the parameters required for any operations and also the result of those operations. Before starting that operation, all the required parameters are pushed onto the evaluation stack. Then the operation is started and it pops its parameters. When the operation is finished, it pushes the result back onto the evaluation stack.

All the objects are stored on the heap and the evaluation stack in the frames deals with references to them. So the references to these objects can be pushed onto the evaluation stack temporarily to be used for the later operations. Most of Python's bytecode instructions manipulate the evaluation stack in the current frame. In this article whenever we talk about the stack it means the evaluation stack in the current frame or the evaluation stack in the global frame if we are not in the scope of any functions.

Let me start with a simple example, and disassemble the bytecode of the following source code:

```
a=1
b=2
c=a+b
```

To do that we can write:

```
s='''a=1
b=2
c=a+b
'''
c=compile(s, "", "exec")
disassemble(c)
```

and we get:

```
1         0 LOAD_CONST               0 (1)
          2 STORE_NAME               0 (a)

2         4 LOAD_CONST               1 (2)
          6 STORE_NAME               1 (b)

3         8 LOAD_NAME                0 (a)
         10 LOAD_NAME                1 (b)
         12 BINARY_ADD
         14 STORE_NAME               2 (c)
         16 LOAD_CONST               2 (None)
         18 RETURN_VALUE
```

In addition, we can check some other attributes of the code object:

```
c.co_consts
# output is:  (1, 2, None)
c.co_names
# output is:  ('a', 'b', 'c')
```

Here the code is running in the module, so we are inside the global frame. The first instruction is `LOAD_CONST 0`. The instruction

`LOAD_CONST` *`consti`*

pushes the value of `co_consts[consti]` onto the stack. So we are pushing `co_consts[0]` (which is equal to `1`) onto the stack.

It is important to note that stack works with references to the objects. So whenever we say that an instruction pushes an object or the value of an object onto the stack, it means that a reference (or pointer) to that object is being pushed. The same thing happens when an object or its value is popped off the stack. Again its reference is popped. The interpreter knows how to retrieve or store the object's data using these references.

The instruction

`STORE_NAME` *`namei`*

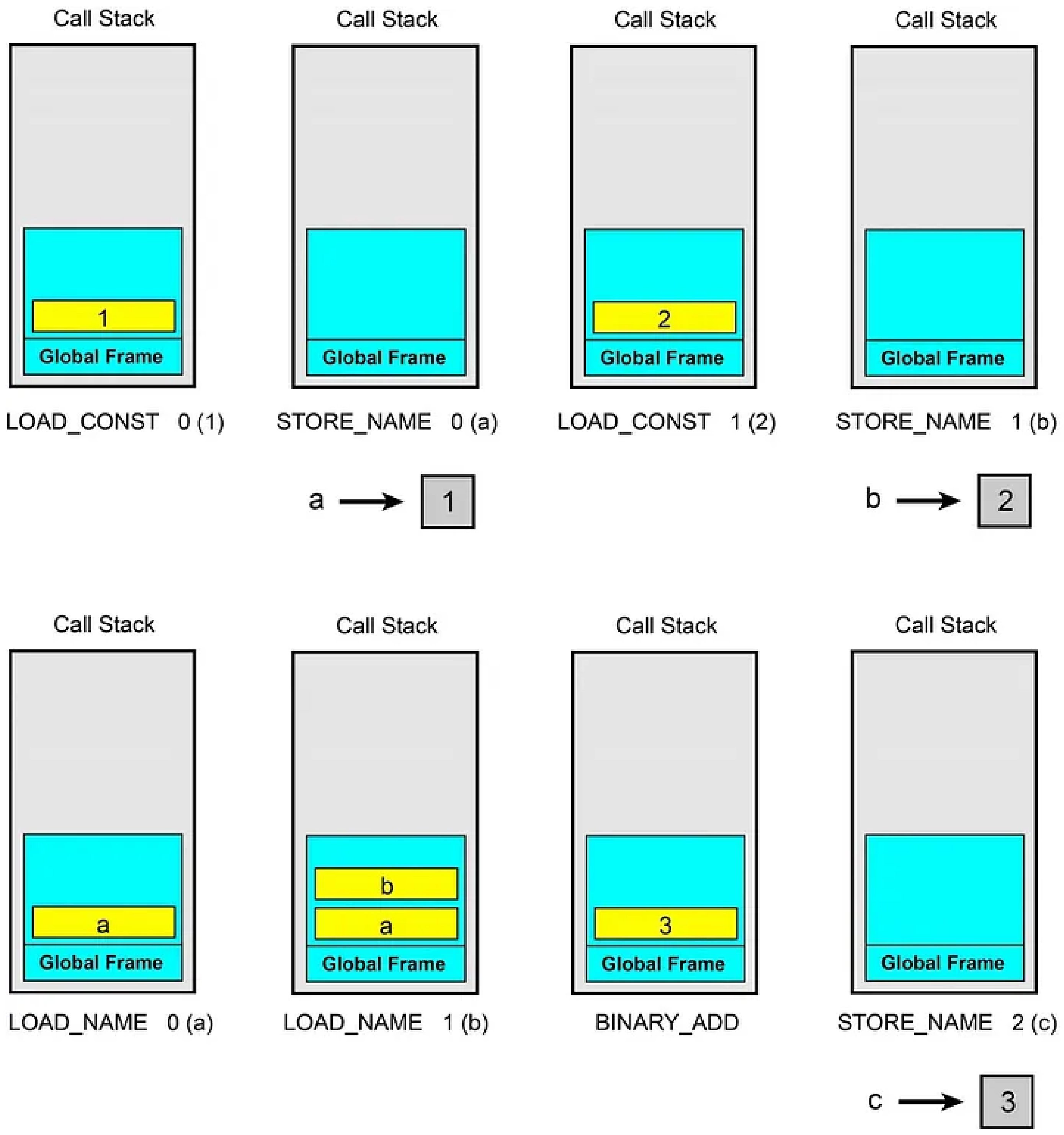
pops the top of the stack and stores it into an object whose reference is stored in `co_names[namei]` of the code object. So `STORE_NAME 0` pops the element on top of the stack (which is `1`) and stores it in an object. The reference to this object is `co_names[0]` which is `a`. These two instructions are the bytecode equivalent of `a=1` in the source code. `b=2` is converted similarly, and now the interpreter has created the objects `a` and `b`. The last line of the source code is `c=a+b`. The instruction

```
BINARY_ADD
```

pops the top two elements of the stack (`1` and `2`), adds them together and pushes the result (`3`) onto the stack. So now `3` is on top of the stack. After that `STORE_NAME 2` pops the top of the stack into the local object (referred by) `c` . Now remember that `compile` in `exec` mode compiles the source code into a bytecode that finally returns `None` . The instruction `LOAD_CONST 2` pushes `co_consts[2]=None` onto the stack, and the instruction

```
RETURN_VALUE
```

returns with the top of the stack to the caller of the function. Of course, here we are in the module scope and there is no caller function, so `None` is the final result which remains on top of the global stack. Figure 1 shows all the bytecode operations with offsets 0 to 14 (Again it should be noted that the references to the objects are pushed onto the stack, not the objects or their values. The figure does not show it explicitly).



Functions, global and local variables

Now let's see what happens if we also have a function. We are going to disassemble the bytecode of a source code which has a function:

```
#Listing 2
s='''a = 1
b = 2
```

```
def f(x):
    global b
    b = 3
    y = x + 1
    return y
f(4)
print(a)
'''
c=compile(s, "", "exec")
disassemble(c)
```

The output is:

```
1         0 LOAD_CONST               0 (1)
          2 STORE_NAME               0 (a)

2         4 LOAD_CONST               1 (2)
          6 STORE_GLOBAL             1 (b)

3         8 LOAD_CONST               2 (<code object f at
0x00000218C2E758A0, file "", line 3>)
         10 LOAD_CONST               3 ('f')
         12 MAKE_FUNCTION            0
         14 STORE_NAME               2 (f)

8        16 LOAD_NAME                2 (f)
         18 LOAD_CONST               4 (4)
         20 CALL_FUNCTION            1
         22 POP_TOP

9        24 LOAD_NAME                3 (print)
         26 LOAD_NAME                0 (a)
         28 CALL_FUNCTION            1
         30 POP_TOP
         32 LOAD_CONST               5 (None)
         34 RETURN_VALUE

Disassembly of<code object f at 0x00000218C2E758A0, file "", line 3>:

5         0 LOAD_CONST               1 (3)
          2 STORE_GLOBAL             0 (b)

6         4 LOAD_FAST                0 (x)
          6 LOAD_CONST               2 (1)
          8 BINARY_ADD
         10 STORE_FAST               1 (y)
```

```
  7        12 LOAD_FAST                1 (y)
           14 RETURN_VALUE
```

In addition, we can check some other attributes of the code object:

```
c.co_consts
# output is:  (1, 2, <code object f at 0x00000218C2E758A0, file "",
line 3>, 'f', 4, None)

c.co_names
# Output is: ('a', 'b', 'f', 'print')
```

In the first line (offsets 0 and 2) the constant `1` is first pushed into the evaluation stack of the global frame using `LOAD_CONST 0`. Then `STORE_NAME 0` pops it and stores it in an object.

In the second line, the constant `2` is pushed into the stack using `LOAD_CONST 1`. However, a different opname is used to assign it to the reference. The instruction

```
STORE_GLOBAL namei
```

pops the top of the stack and stores it into an object whose reference is stored in `co_names[namei]`. So `2` is stored in the object referred by `b`. This is considered a global variable. But why was this instruction not used for `a`? The reason is that `a` is a global variable inside the function `f`. If a variable is defined at the module scope and no functions access it, it will be stored and loaded using `STORE_NAME` and `LOAD_NAME`. At the module scope, there is no distinction between global and local variables.

In the third line, the function `f` is defined. The body of the function is compiled in a separate code object named `<code object f at 0x00000218C2E758A0, file "", line 3>` and it is pushed onto the stack. Then a string object which is the name of this function `'f'` is pushed onto the stack (in fact references to them are pushed). The instruction

```
MAKE_FUNCTION argc
```

is used to create the function. It needs some parameters that should be pushed onto the stack. The name of the function should be on top of the stack and the function's code object should be below it. In this example, its oparg is zero, but it can have other values. For example, if the function definition had a keyword argument like:

```
def f(x=5):
    global b
    b = 3
    y = x + 1
    return y
```

Then the disassembled bytecode for line 2 would be:

```
2         4 LOAD_CONST               5 ((5,))
          6 LOAD_CONST               1 (<code object f at
0x00000218C2E75AE0, file "", line 2>)
          8 LOAD_CONST               2 ('f')
         10 MAKE_FUNCTION            1
```

An oparg of `1` for `MAKE_FUNCTION` indicates that the function has some keyword arguments, and a tuple containing the default values should be pushed onto the stack before the function's code object (here it is `(5,)`). After creating the function, `MAKE_FUNCTION` pushes the new function object onto the stack. Then at offset 14, `STORE_NAME 2` pops the function object and stores it as a function object referenced by `f`.

Now let's looks inside the code object of `f(x)` which starts at line 5. The statement `global a` does not convert into a separate instruction in the bytecode. It only guides the compiler that `a` should be treated as a global variable. So `STORE_GLOBAL 0` will be used to change its value. The instruction

```
LOAD_GLOBAL namei
```

pushes a reference to the object referred by `co_names[namei]` onto the stack. It is then stored in `b` using `STORE_GLOBAL 0`. The instruction

```
LOAD_FAST var_num
```

pushes a reference to the object whose reference is `co_varnames[var_num]` onto the stack. In the code object of function `f`, the attribute `co_varnames` contains:

```
('x', 'y')
```

So `LOAD_FAST 0` pushes `x` onto the stack. Then `1` is pushed onto the stack. `BINARY_ADD` pops `x` and `1`, adds them together and pushes the result onto the stack. The instruction

```
STORE_FAST var_num
```

pops the top of the stack and stores it into an object whose reference is stored in `co_varnames[var_num]`. So `STORE_FAST 1` pops the result and stores it in an object whose reference is `y`. `LOAD_FAST` and `STORE_FAST` are used with local variables of the functions. So they are not used at the module scope. On the other hand, `LOAD_GLOBAL` and `STORE_GLOBAL` are used for the global variables accessed inside functions. Finally, `LOAD_FAST 1` will push the value of `y` on top of the stack and `RETURN_VALUE` will return it to the caller of the function which is the module.

But how this function is called? If you look at the bytecode of line 8, first, `LOAD_NAME 2` pushes the function object whose reference is `f` onto the stack. `LOAD_CONST 4` pushes its argument (`4`) onto the stack. The instruction

```
CALL_FUNCTION argc
```

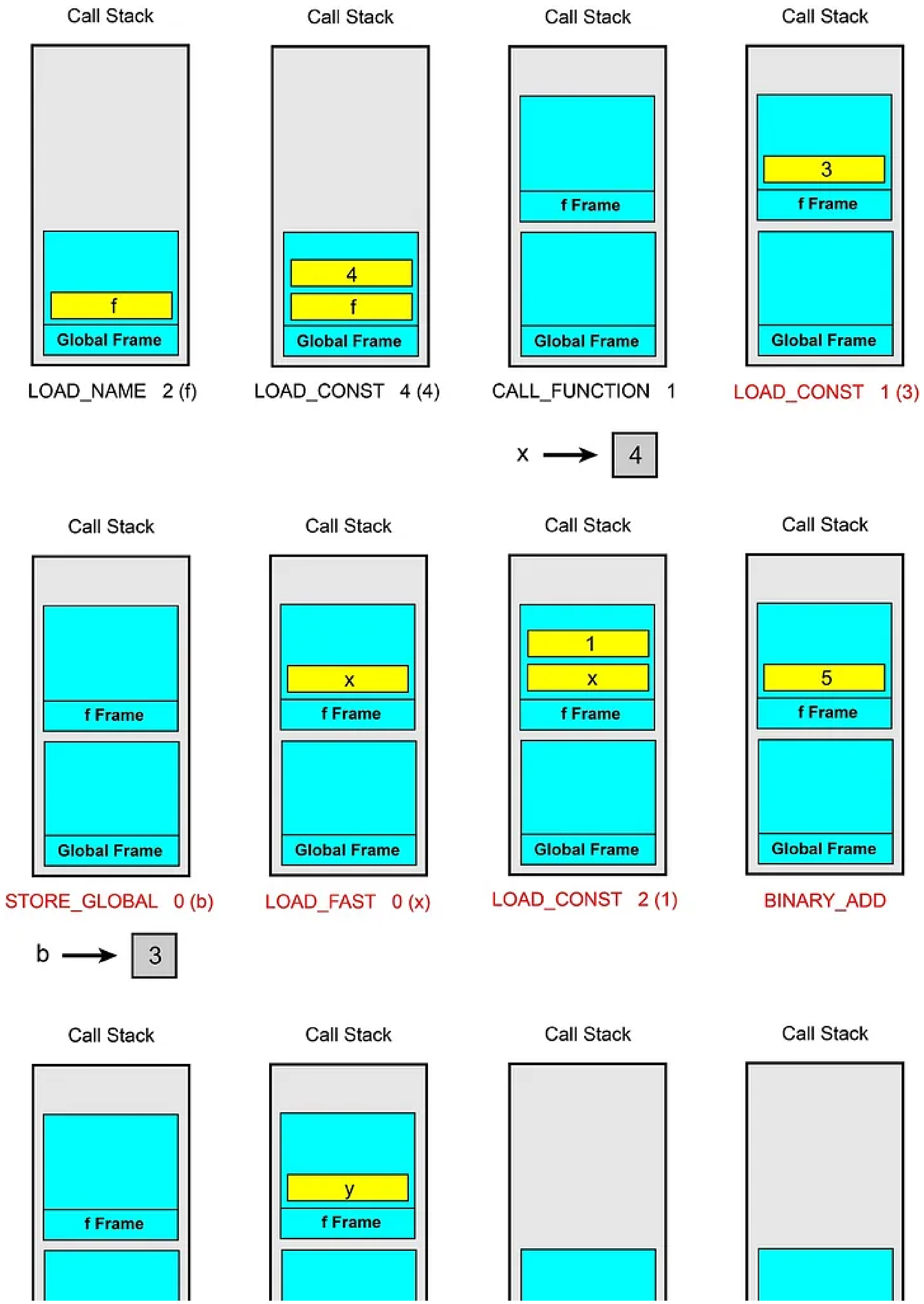
calls a callable object with positional arguments. Its oparg, *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is the function callable object to call.

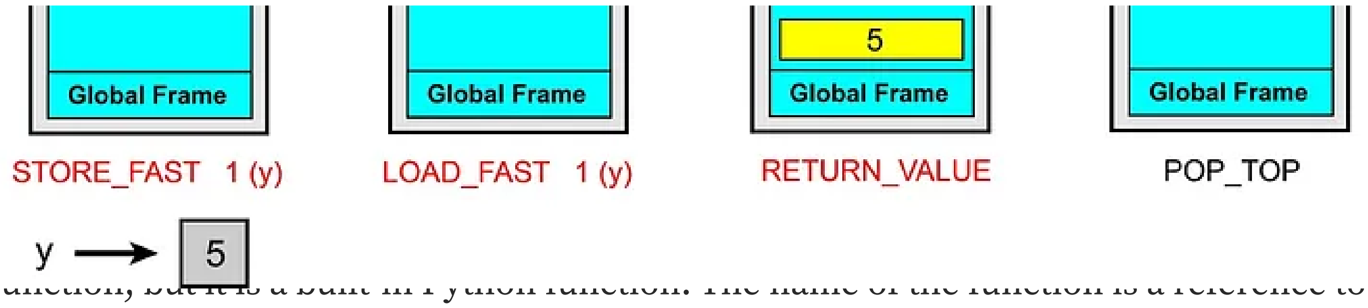
`CALL_FUNCTION` first pops all the arguments and the callable object off the stack. Then it will allocate a new frame on the call stack, populate the local variables for the function call, and execute the bytecode of the function inside that frame. Once that's done, the frame will be popped off the call stack, and in the previous frame, the return value of the function will be pushed on top of the evaluation stack. If there is no previous frame, it will be pushed on top of the evaluation stack of the global frame.

In our example, we only have one positional argument, so the instruction will be `CALL_FUNCTION 1`. After that, the instruction

```
POP_TOP
```

pops the item on top of the stack. That is because we do not need the returned value of the function anymore. Figure 2 shows all the bytecode operations with offsets 16 to 22. The bytecode instructions inside `f(x)` are shown in red.


Call Stack
f
Global Frame
LOAD_NAME 2 (f)
Call Stack
4
f
Global Frame
LOAD_CONST 4 (4)
Call Stack
f Frame
Global Frame
CALL_FUNCTION 1
x → 4
Call Stack
3
f Frame
Global Frame
LOAD_CONST 1 (3)
Call Stack
f Frame
Global Frame
STORE_GLOBAL 0 (b)
b → 3
Call Stack
x
f Frame
Global Frame
LOAD_FAST 0 (x)
Call Stack
1
x
f Frame
Global Frame
LOAD_CONST 2 (1)
Call Stack
5
f Frame
Global Frame
BINARY_ADD
Call Stack
f Frame
Call Stack
y
f Frame
Call Stack
Call Stack



function, but it is a built-in Python function. The name of the function is a reference to its callable object. So first it is pushed onto the stack and then its argument is pushed. Finally, it will be called using `CALL_FUNCTION`. `print` will return `None`, and the returned value will be popped off the stack after that.

Python uses its built-in functions to create data structures. For example, the following line:

```
a = [1,2,3]
```

will be converted to:

```
1         0 LOAD_CONST               0 (1)
          2 LOAD_CONST               1 (2)
          4 LOAD_CONST               2 (3)
          6 BUILD_LIST               3
          8 STORE_NAME               0 (a)
```

Initially, each element of the list is pushed onto the stack. Then the instruction

BUILD_LIST *count*

is called to create the list using the *count* items from the stack and pushes the resulting list object onto the stack. Finally, the object on the stack will be popped and stored on the heap and `a` will be its reference.

EXTENDED_ARG

As mentioned before, some of the instructions can have an argument too big to fit into the default one byte, and they will be prefixed by the instruction `EXTENDED_ARG` . Here is an example. Suppose that we want to print 260 `*` characters. We could simply write `print('*' * 260)` . However, I will write something unusual instead:

```
s= 'print(' + '"*",' * 260 + ')'
c = compile(s, "", "exec")
disassemble(c)
```

Here `s` contains a `print` function which takes 260 arguments and each of them is a `*` character. Now look at the resulting disassembled bytecode:

```
1         0 LOAD_NAME                0 (print)
          2 LOAD_CONST               0 ('*')
          4 LOAD_CONST               0 ('*')
                 .                   .
                 .                   .
                 .                   .
        518 LOAD_CONST               0 ('*')
        520 LOAD_CONST               0 ('*')
        522 EXTENDED_ARG             1
        524 CALL_FUNCTION          260
        526 POP_TOP
        528 LOAD_CONST               1 (None)
        530 RETURN_VALUE
```

Here `print` is pushed onto the stack first. Then its 260 arguments are pushed. Then `CALL_FUNCTION` should call the function. But it needs the number of the arguments (of the target function) as its oparg. Here this number is 260 which is bigger than the maximum number that a byte can take. Remember that the oparg is only one byte. So the `CALL_FUNCTION` is prefixed by `EXTENDED_ARG` . The actual bytecode is:

```
        522 EXTENDED_ARG             1
        524 CALL_FUNCTION            4
```

As mentioned before the oparg of EXTENDED_ARG will be left-shifted by eight bits or simply multiplied by 256 and will be added to the oparg of the next opcode. So the oparg of `CALL_FUNCTION` will be interpreted to be `256+4 = 260` (please note that what the `disassemble` function shows is this interpreted oparg not the actual oparg in the bytecode).

Conditional statements and jumps

Consider the following source code which has an `if-else` statement:

```
s='''a = 1
if a>=0:
    b=a
else:
    b=-a
'''
c=compile(s, "", "exec")
disassemble(c)
```

The disassembled bytecode is:

```
1         0 LOAD_CONST               0 (1)
          2 STORE_NAME               0 (a)

2         4 LOAD_NAME                0 (a)
          6 LOAD_CONST               1 (0)
          8 COMPARE_OP               5 (>=)
         10 POP_JUMP_IF_FALSE       18

3        12 LOAD_NAME                0 (a)
         14 STORE_NAME               1 (b)
         16 JUMP_FORWARD             6 (to 24)

5   >>   18 LOAD_NAME                0 (a)
         20 UNARY_NEGATIVE
         22 STORE_NAME               1 (b)
    >>   24 LOAD_CONST               2 (None)
         26 RETURN_VALUE
```

We have a few new instructions here. In line 2, the object that `a` refers to is pushed onto the stack, and then literal `0` is pushed. The instruction

`COMPARE_OP` *`oparg`*

performs a Boolean operation. The operation name can be found in `cmp_op[oparg]`. The values of `cmp_op` are stored in a list named `dis.cmp_op`. The instruction first pops the top two elements of the stack. We call the first one `TOS1` and the second one `TOS2`. Then the boolean operation selected by *oparg* is performed on them `(TOS2 cmp_op[oparg] TOS1)`, and the result is pushed on top of the stack. In this example `TOS1=0` and `TOS2=value of a`. In addition, the *oparg* is `5` and `cmp_op[5]='≥'`. So `cmp_op` will test `a≥0` and stores the result (which is true or false) on top of the stack.

The instruction

`POP_JUMP_IF_FALSE` *`target`*

performs a conditional jump. First, it pops the top of the stack. If the element on top of the stack is false, it sets the bytecode counter to *target*. The bytecode counter shows the current bytecode offset which is being executed. So it jumps to the bytecode offset which is equal to target and the execution of bytecode continues from there. The offset 18 in the bytecode is a jump target, so there is a `>>` in front of that in the disassembled bytecode. The instruction

`JUMP_FORWARD` *`delta`*

increments the bytecode counter by *delta*. In the previous bytecode, the offset of this instruction is 16, and we know that each instruction takes 2 bytes. So when this instruction is finished, the bytecode counter is `16+2=18`. Here `delta=6`, and `18+6=24`, so it jumps to the offset `24`. The offset 24 is a jump target and it has a `>>` sign too.

Now we can see how the `if-else` statement is converted to the bytecode. The `cmp_op` checks if `a≥0` . If the result is false, `POP_JUMP_IF_FALSE` jumps to the offset 18 which is the start of `else` block. If it is true, the `if` block will be executed and then `JUMP_FORWARD` jumps to the offset 24 and does not execute the `else` block.

Now let's see a more complicated Boolean expression. Consider the following source code:

```
s='''a = 1
c = 3
if a>=0 and c==3:
    b=a
else:
    b=-a
'''
c=compile(s, "", "exec")
disassemble(c)
```

Here we have a logical `and` . The disassembled bytecode is:

```
1         0 LOAD_CONST                0 (1)
          2 STORE_NAME                0 (a)

2         4 LOAD_CONST                1 (3)
          6 STORE_NAME                1 (c)

3         8 LOAD_NAME                 0 (a)
         10 LOAD_CONST                2 (0)
         12 COMPARE_OP                5 (>=)
         14 POP_JUMP_IF_FALSE        30
         16 LOAD_NAME                 1 (c)
         18 LOAD_CONST                1 (3)
         20 COMPARE_OP                2 (==)
         22 POP_JUMP_IF_FALSE        30

4        24 LOAD_NAME                 0 (a)
         26 STORE_NAME                2 (b)
         28 JUMP_FORWARD              6 (to 36)

6   >>   30 LOAD_NAME                 0 (a)
         32 UNARY_NEGATIVE
         34 STORE_NAME                2 (b)
```

```
     >>   36 LOAD_CONST               3 (None)
          38 RETURN_VALUE
```

In Python `and` is a short-circuit operator. So when evaluating `X and Y`, it only evaluates `Y` if `X` is true. This can be easily seen in the bytecode. In line 3, first, the left operand of `and` is evaluated. If `(a≥0)` is false, it does not evaluate the second operand and jumps to the offset 30 to execute the `else` block. However, if it is true, the second operand `(b==3)` will be evaluated too.

Loops and block stack

As mentioned before, there is an evaluation stack inside each frame. In addition, in each frame, there is a *block stack*. It is used by CPython to keep track of certain types of control structures like the loops, `with` blocks and `try/except` blocks. When CPython wants to enter one of these structures a new item is pushed onto the block stack, and when CPython exits that structure, the item for that structure is popped off the block stack. Using the block stack CPython knows which structure is currently active. So when it reaches a `break` or `continue` statement, it knows which structures should be affected.

Let's see how loops are implemented in the bytecode. Consider the following code and its disassembled bytecode:

```
s='''for i in range(3):
    print(i)
'''
c=compile(s, "", "exec")
disassemble(c)

--------------------------------------------------------------------

1         0 SETUP_LOOP              24 (to 26)
          2 LOAD_NAME                0 (range)
          4 LOAD_CONST               0 (3)
          6 CALL_FUNCTION            1
          8 GET_ITER
    >>   10 FOR_ITER                12 (to 24)
         12 STORE_NAME               1 (i)

2        14 LOAD_NAME                2 (print)
         16 LOAD_NAME                1 (i)
```

```
             18 CALL_FUNCTION            1
             20 POP_TOP
             22 JUMP_ABSOLUTE           10
        >>   24 POP_BLOCK
        >>   26 LOAD_CONST               1 (None)
             28 RETURN_VALUE
```

The instruction

```
  SETUP_LOOP delta
```

is executed before the loop starts. This instruction pushes a new item (which is also called a block) onto the block stack. *delta* is added to the bytecode counter to determine the offset of the next instruction after the loop. Here the offset of `SET_LOOP` is `0`, so the bytecode counter is `0+2=2`. In addition, *delta* is `24`, so the offset of the next instruction after the loop is `2+24=26`. This offset is stored in the block that is pushed onto the block stack. In addition, the current number of items in the evaluation stack is stored in this block.

After that, the function `range(3)` should be executed. Its oparg (`3`) is pushed before the name of the function. The result is an *iterable*. Iterables can generate an *iterator* using the instruction:

```
  GET_ITER
```

It takes the iterable on top of the stack and pushes an iterator of that. The instruction:

```
  FOR_ITER delta
```

assumes that there is an iterator on top of the stack. It calls its `__next__()` method. If it yields a new value, this value is pushed on top of the stack (above the iterator). Inside the loop, the top of the stack is stored in `i` after that, and the `print` function is executed. Then the top of the stack which is the current value of the iterator is popped. After that, the instruction

JUMP_ABSOLUTE *target*

sets the bytecode counter to *target* and jumps to the *target* offset. So it jumps to offset 10 and runs `FOR_ITER` again to get the next value of the iterator. If the iterator indicates that there are no further elements available, the top of stack is popped, and the byte code counter is incremented by *delta*. Here `delta=12` , so after finishing the loop it jumps to offset 24. At offset 24, the instruction

POP_BLOCK

removes the current block from the top of the block stack. The offset of the next instruction after the loop is stored in the block (here it is 26). So the interpreter will jump to that offset and continue execution from there. Figure 3 shows the bytecode operations with offsets 0, 10, 24 and 26 as an example (In fact in Figures 1 and 2 we only showed the evaluation stack in each frame).

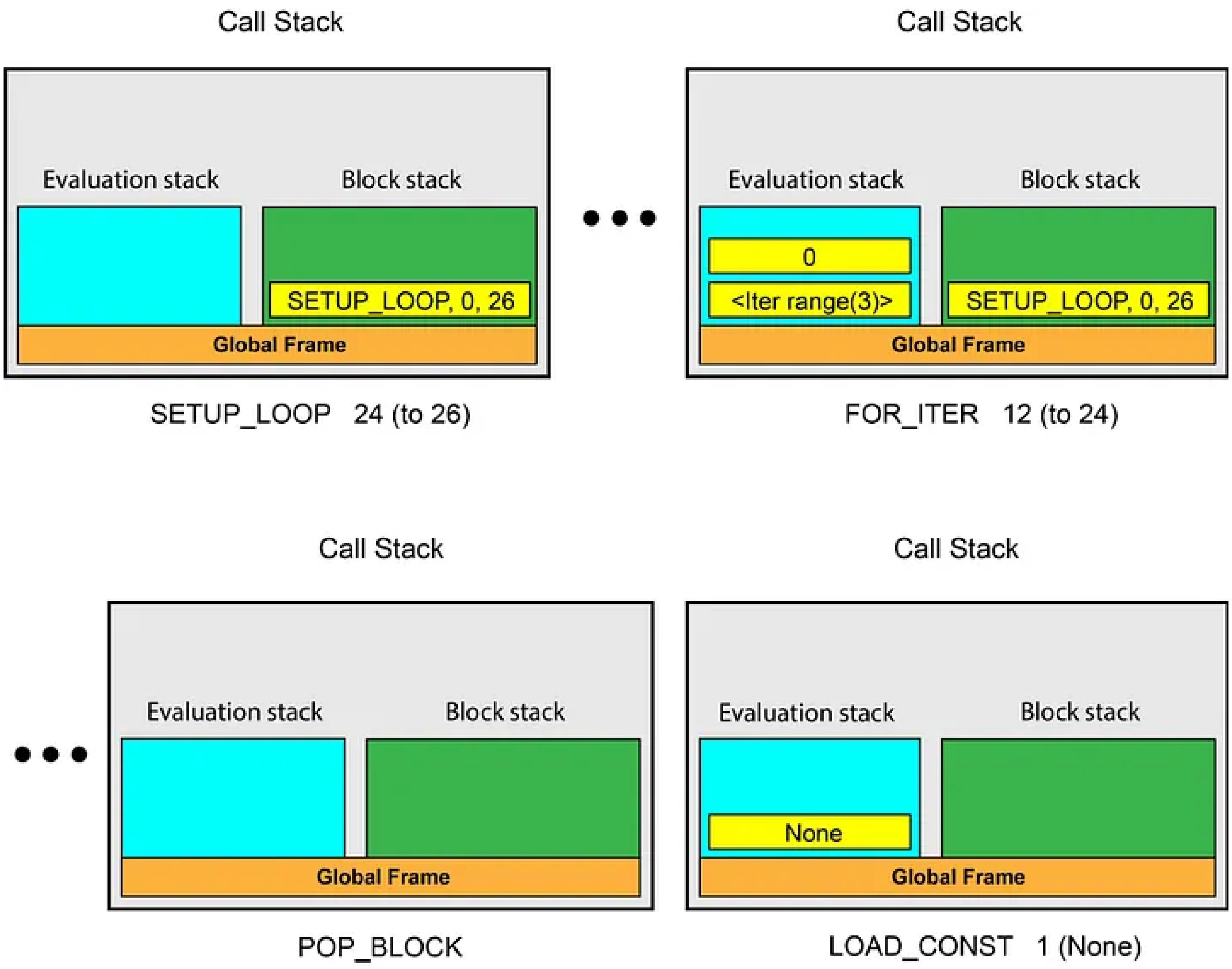


Figure 3

But what happens if we add a `break` statement to this loop? Consider the following source code and its disassembled bytecode:

```
s='''for i in range(3):
    break
    print(i)
'''
c=compile(s, "", "exec")
disassemble(c)

----------------------------------------------------------------------------

1          0 SETUP_LOOP              26 (to 28)
           2 LOAD_NAME                0 (range)
           4 LOAD_CONST               0 (3)
           6 CALL_FUNCTION            1
```

```
              8 GET_ITER
        >>   10 FOR_ITER                 14 (to 26)
             12 STORE_NAME                1 (i)

  2          14 BREAK_LOOP

  3          16 LOAD_NAME                 2 (print)
             18 LOAD_NAME                 1 (i)
             20 CALL_FUNCTION             1
             22 POP_TOP
             24 JUMP_ABSOLUTE            10
        >>   26 POP_BLOCK
        >>   28 LOAD_CONST                1 (None)
             30 RETURN_VALUE
```

We have only added a `break` statement to the previous loop. This statement is converted to

```
BREAK_LOOP
```

This opcode removes those extra items on the evaluation stack and pops the block from the top of the block stack. You should notice that the other instructions of the loop are still using the evaluation stack. So when the loop breaks, the items that belong to it should be popped off the evaluation stack. In this example, the iterator object is still on top of the stack. Remember that the block in the block stack stores the number of items that existed in the evaluation stack before starting the loop.

So by knowing that number, `BREAK_LOOP` pops those extra items off the evaluation stack. Then it jumps to the offset which is stored in the current block of the block stack (here it is 28). That is the offset of the next instruction after the loop. So the loop breaks and the execution is continued from there.

Creating the code object

The code object is an object of type `code` , and it is possible to create it dynamically. The module `types` can help with dynamic creation of new types, and the class `CodeType()` in this module returns a new code object:

```
types.CodeType(co_argcount, co_kwonlyargcount,
               co_nlocals, co_stacksize, co_flags,
               co_code, co_consts, co_names,
               co_varnames, co_filename, co_name,
               co_firstlineno, co_lnotab, freevars=None,
               cellvars=None)
```

The arguments form all the attributes of the code object. You are already familiar with some of these arguments (like `co_varnames` and `co_firstlineno`). `freevars` and `cellvars` are optional since they are used in closures and not all functions use them (Refer to this article for more information about them). The other attributes are explained using the following function as an example:

```
def f(a, b, *args, c, **kwargs):
    d=1
    def g():
        return 1
    g()
    return 1
```

`co_argcount` : If the code object is that of a function, the number of arguments it takes (not including keyword only arguments, `*` or `**` args). For function `f` it is `2`.

`co_kwonlyargcount` : If the code object is that of a function, number of keyword only arguments (not including `**` arg). For function `f` it is `1`.

`co_nlocals` : The number of local variables plus the name of functions defined in the code object (arguments are also considered local variables). In fact, it is the number of elements in `co_varnames` which is `('a', 'b', 'c', 'args', 'kwargs', 'd', 'g')`. So it is `7` for `f`.

`co_stacksize` : Shows the largest number of elements that will be pushed onto the evaluation stack by this code object. Remember that some opcodes need to push some elements onto the evaluation stack. This attribute shows the largest size that the stack will ever grow to from the bytecode operations. In this example it is `2`. Let me explain the reason for that. If you disassemble the bytecode of this function you get:

```
  2           0 LOAD_CONST               1 (1)
              2 STORE_FAST               5 (d)

  3           4 LOAD_CONST               2 (<code object g at 
  0x0000028A62AB1D20, file "<ipython-input-614-cb7dfbcc0072>", line 3>)
              6 LOAD_CONST               3 ('f.<locals>.g')
              8 MAKE_FUNCTION            0
             10 STORE_FAST               6 (g)

  5          12 LOAD_FAST                6 (g)
             14 CALL_FUNCTION            0
             16 POP_TOP

  6          18 LOAD_CONST               1 (1)
             20 RETURN_VALUE
```

In line 2, one element is pushed onto the stack using the `LOAD_CONST` and will be popped using `STORE_FAST` . Lines 5 and 6 similarly push one element onto the stack and pop it later. But in line 3, two elements are pushed onto the stack to define the inner function `g` : its code object and its name. So this is the maximum number of elements that will be pushed onto the evaluation stack by this code object, and it determines the stack size.

`co_flags` : An integer, with bits indicating things like whether the function accepts a variable number of arguments, whether the function is a generator, etc. In our example its value is `79` . The binary value of `79` is `0b1001111` . It uses a little-endian system in which the bytes are written from left to right in increasing significance. So the first bit is the first one on the right. You can refer to this link for the meaning of these bits. For example, the third bit from the right represents the `CO_VARARGS` flag. When it is `1` it means that the code object has a variable positional parameter (`*args` -like).

`co_filename` : A string, specifying the file in which the function is present. In this case, it is `'<ipython-input-59-960ced5b1120>'` since I was running the script in Jupyter notebook.

`co_name` : A name with which this code object was defined. Here it is the name of the function `'f'` .

Bytecode injection

Now that we are completely familiar with the code object, we can start changing its bytecode. It is important to note that the code object is immutable. So once created we cannot change it. Suppose that we want to change the bytecode of the following function:

```
def f(x, y):
    return x + y

c = f.__code__
```

Here we cannot change the bytecode of the code object of the function directly. Instead, we need to create a new code object and then assign it to this function. To do that we need a few more functions. The `disassemble` function can disassemble the bytecode into some human-friendly instructions. We can change them as we like, but then we need to assemble it back to the bytecode to assign it to a new code object. The output of `disassemble` is a formatted string which is easy to read, but difficult to change. So I will add a new function which can disassemble the bytecode into a list of instructions. It is very similar to `disassemble`, however, its output is a list.

```
def disassemble_to_list(c):
    code_list = []
    bytecode = c.co_code
    for offset, opcode, oparg in unpack_op(bytecode):
        argval = get_argvalue(offset, c, opcode, oparg)
        if argval is not None:
            if type(argval)==str:
                argval = argval.strip("\'")
            argval = None if argval=='None' else argval
            code_list.append([dis.opname[opcode], argval])
        else:
            if oparg is not None:
                code_list.append([dis.opname[opcode], oparg])
            else:
                code_list.append([dis.opname[opcode]])
    return code_list
```

BC_Listing7.py hosted with ❤ by GitHub view raw

We can try it on the previous function:

```
disassembled_bytecode = disassemble_to_list(c)
```

Now `disassembled_bytecode` is equal to:

```
[['LOAD_FAST', 'x'],
 ['LOAD_FAST', 'y'],
 ['BINARY_ADD'],
 ['RETURN_VALUE']]
```

We can now change the instructions of this list easily. But we also need to assemble it back to the bytecode:

```python
def get_oparg(offset, opcode, argval, constants, varnames, names, cell_names):
    oparg = argval
    if opcode in dis.hasconst:
        if constants is not None:
            oparg = constants.index(argval)
    elif opcode in dis.hasname:
        if names is not None:
            oparg = names.index(argval)
    elif opcode in dis.hasjrel:
        argval = int(argval.split()[1])
        oparg = argval - offset - 2
    elif opcode in dis.haslocal:
        if varnames is not None:
            oparg = varnames.index(argval)
    elif opcode in dis.hascompare:
        oparg = dis.cmp_op.index(argval)
    elif opcode in dis.hasfree:
        if cell_names is not None:
            oparg = cell_names.index(argval)
    return oparg
```

BC_Listing8.py hosted with ❤ by GitHub view raw

```python
def assemble(code_list, constants, varnames, names, cell_names):
    byte_list = []
    for i, instruction in enumerate(code_list):
        if len(instruction)==2:
            opname, argval = instruction
            opcode = dis.opname.index(opname)
            oparg = get_oparg(i*2, opcode, argval, constants, varnames, names, cell_names)
        else:
            opname = instruction[0]
            opcode = dis.opname.index(opname)
            oparg = 0
        byte_list += [opcode, oparg]
    return(bytes(byte_list))
```

BC_Listing9.py hosted with ❤ by GitHub view raw

The function `get_oparg` is like the inverse of `get_argvalue`. It takes an argvalue which is the human-friendly meaning of an oparg and returns the corresponding oparg. It needs the code object as its argument since the attributes of the code object like `co_consts` are necessary to convert the argvalue into the oparg.

The function `assemble` takes a code object and a disassembled bytecode list and assembles it back into the bytecode. It uses `dis.opname` to convert the opname to the opcode. Then it calls `get_oparg` to convert the argvalue to the oparg. Finally, it returns a bytes literal of the bytecode list. We can now use these new functions to change the bytecode of the previous function `f`. First, we change one of the instructions in `disassembled_bytecode`:

```
disassembled_bytecode[2] = ['BINARY_MULTIPLY']
```

The instruction

BINARY_MULTIPLY

pops the top two elements of the stack, multiplies them together and pushes the result onto the stack. Now we assemble the modified disassembled bytecode:

```
new_co_code= assemble(disassembled_bytecode, c.co_consts,
                      c.co_varnames, c.co_names,
                      c.co_cellvars+c.co_freevars)
```

After that we create a new code object:

```
import types
nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
                    c.co_nlocals, c.co_stacksize, c.co_flags,
                    new_co_code, c.co_consts, c.co_names,
                    c.co_varnames, c.co_filename, c.co_name,
                    c.co_firstlineno, c.co_lnotab,
                    c.co_freevars, c.co_cellvars)
f.__code__ = nc
```

We use all the attributes of `f` to create it and only replace the new bytecode (`new_co_code`). Then we assign the new code object to `f`. Now if we run `f` again, it does not add its arguments together. Instead, it will multiply them together:

```
f(2,5)  # Output is 10 not 7
```

Caution: The `types.CodeType` function has two optional arguments for `freevars` and `cellvars`, however, you should be careful when using them. As mentioned before the `co_cellvars` and `co_freevars` attributes of the code object are only used when the code object belongs to a function which has free variables or nonlocal variables. So the function should be a closure or a closure should have been defined inside it. For example, consider the following function:

```
def func(x):
    def g(y):
        return x + y
    return g
```

Now if check its code object:

```
c = func.__code__
c.co_cellvars  # Output is: ('x',)
```

In fact, this function has one nonlocal variable `x` since this variable is accessed by its inner functions. Now we can try recreating its code object using the same attributes:

```
nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
                    c.co_nlocals, c.co_stacksize, c.co_flags,
                    new_co_code, c.co_consts, c.co_names,
                    c.co_varnames, c.co_filename, c.co_name,
                    c.co_firstlineno, c.co_lnotab,
                    cellvars = c.co_cellvars,
                    freevars = c.co_freevars)
```

But if we check the same attribute of the new code object

```
nc.co_cellvars  Output is: ()
```

It turns out to be empty. So `types.CodeType` cannot create the same code object. If you try to assign this code object to a function and execute that function, you will get an error (this has been tested on Python 3.7.4).

Code optimization

Understanding the bytecode instructions can help us with the optimization of the source code. Consider the following source code:

```
setup1='''import math
mult = 2
def f():
    total = 0
    i = 1
    for i in range(1, 200):
        total += mult * math.log(i)
    return total
'''

setup2='''import math
def f():
    log = math.log
    mult = 2
    total = 0
    for i in range(1, 200):
        total += mult * log(i)
    return total
'''
```

Here we define a function `f()` to calculate a simple mathematical expression. It has been defined in two different ways. In `setup1`, we are using the global variable `mult` inside `f()` and directly use the `log()` function from `math` module. In `setup2`, `mult` is a local variable of `f()`. In addition, `math.log` is first stored in the local variable `log`. Now we can compare the performance of these function:

```
t1 = timeit.timeit(stmt="f()", setup=setup1, number=100000)
t2 = timeit.timeit(stmt="f()", setup=setup2, number=100000)
print("t1=", t1)
print("t2=", t2)
--------------------------------------------------------------------
t1= 3.8076129000110086
t2= 3.2230119000014383
```

You may get different numbers for `t1` and `t2` , but the bottom line is that `setup2` is faster than `setup1` . Now let's compare their bytecode to see why it is faster. We just look at the line 7 in the disassembled code of `setup1` and `setup2` . This is the bytecode for this line: `total += mult * log(i)` .

In `setup1` we have:

```
7         24 LOAD_FAST                 0 (total)
          26 LOAD_GLOBAL               1 (mult)
          28 LOAD_GLOBAL               2 (math)
          30 LOAD_METHOD               3 (log)
          32 LOAD_FAST                 1 (i)
          34 CALL_METHOD               1
          36 BINARY_MULTIPLY
          38 INPLACE_ADD
          40 STORE_FAST                0 (total)
          42 JUMP_ABSOLUTE            20
    >>    44 POP_BLOCK
```

But in `setup2` we get:

```
7         30 LOAD_FAST                 2 (total)
          32 LOAD_FAST                 1 (mult)
          34 LOAD_FAST                 0 (log)
          36 LOAD_FAST                 3 (i)
          38 CALL_FUNCTION             1
          40 BINARY_MULTIPLY
          42 INPLACE_ADD
          44 STORE_FAST                2 (total)
          46 JUMP_ABSOLUTE            26
    >>    48 POP_BLOCK
```

As you see in `setup1` both `mult` and `math` are loaded using `LOAG_GLOBAL`, but in `setup2`, `mult` and `log` are loaded using `LOAD_FAST`. So two `LOAD_GLOBAL` instructions have been replace with `LOAD_FAST`. The fact is that `LOAD_FAST` as its name suggests is much faster than `LOAD_GLOBAL`. We mentioned that the name of the global and local variables are stored in the `co_names`, and `co_varnames`. But how does the CPython interpreter find the values when executing the compiled code?

Local variables are stored in an array on each frame (which is not shown in the previous figures to make them simpler). We know that the name of local variables are stored in `co_varnames`. Their values will be stored with the same order in this array. So when the interpreter sees an instruction like `LOAD_FAST 1 (mult)`, it reads the element of that array at index `1`.

The global and builtins of the module are stored in a dictionary. We know that their names are stored in the `co_names`. So when the interpreter sees an instruction like `LOAD_GLOBAL 1 (mult)`, it first gets the name of that global variable from `co_names[1]`. Then it will look up this name in the dictionary to get its value. This is a much slower process compared to a simple array lookup for the local variables. As a result, `LOAD_FAST` is faster than `LOAD_GLOBAL`, and replacing `LOAD_GLOBAL` with `LOAD_FAST` can improve performance. It can be done by simply storing builtin and global variables into local variables or directly changing the bytecode instructions.

Example: Defining constants in Python

This example illustrates how to use the bytecode injection to change the behavior of functions. We are going to write a decorator which adds a *const* statement to Python. In some programming languages like C, C++, and JavaScript there is a *const* keyword. If a variable is declared as const using this keyword, then changing its value is illegal, and we cannot change the value of this variable in the source code anymore.

Python does not have a const statement, and I do not claim that it is really necessary to have such a keyword in Python. In addition, defining constants can be also done without using the bytecode injection. So this is just an example to show you how to put the bytecode injection into action. First, let me show how you can use it. The const keyword is provided using a function decorator named `const`. Once you decorate a function by `const` , you can declare the variable inside it as constants using the keyword `const.` (the `.` at the end is part of the keyword). Here is an example:

```
@const
def f(x):
    const. A=5
    return A*x
f(2)  # Output is: 10
```

The variable `A` inside `f` is now a constant. Now if you try to reassign this variable inside `f` , an exception will be raised:

```
@const
def f(x):
    const. A=5
    A = A + 1
    return A*x
-------------------------------------------------------------------------#
This raises an exception :
ConstError: 'A' is a constant and cannot be reassigned!
```

When a variable is declared as const., it should be assigned to its initial value, and it will be a local variable of that function.

Now let me show you how it has been implemented. Suppose that I define a function like this (without decoration):

```
def f(x):
    const. A=5
    A = A + 1
    return A*x
```

It will be compiled properly. But if you try executing this function, you get an error:

```
f(2)
--------------------------------------------------------------------------
```

```
NameError: name 'const' is not defined
```

Now let's take a look at the disassembled bytecode of this function:

```
2         0 LOAD_CONST               1 (5)
          2 LOAD_GLOBAL              0 (const)
          4 STORE_ATTR               1 (A)

3         6 LOAD_FAST                1 (A)
          8 LOAD_CONST               2 (1)
         10 BINARY_ADD
         12 STORE_FAST               1 (A)

4        14 LOAD_FAST                1 (A)
         16 LOAD_FAST                0 (x)
         18 BINARY_MULTIPLY
         20 RETURN_VALUE
```

When Python tries to compile the function, it takes `const` as a global variable since it has not been defined in the function. The variable `A` is considered to be an attribute of the global variable `A`. In fact, `const. A=1` is the same as `const.A=1` since Python ignores the whitespace between the dot operator and the name of the attribute. Of course, we really do not have a global variable named `A` in the source code. But Python will not check it at compile time. Only during execution it will turn out that the name `const` is not defined. So our source code will be accepted during compiling. But we need to change its bytecode before executing the code object of this function. We first need to create a function to change the bytecode:

This function receives the list of bytecode instructions generated by `assemble_to_list` as its argument. It has two lists named `constants` and `indices` which store the name of the variables declared as const and the offset at which they have been assigned for the first time. The first loop searches the list of bytecode instructions and finds all the `['LOAD_GLOBAL', 'const']` instructions. The name of the variable should be in the next instruction. In this example the next instruction is `['STORE_ATTR', 'A']`, and the name is `A`. This name and the offset of this instruction is stored in `constants` and `indices`. Now we need to get rid of the global variable `const` and its attribute and create a local variable named `A` instead. The instruction

```
NOP
```

is a 'Do nothing' code. When the interpreter reaches to `NOP`, it will ignore it. We cannot simply delete the opcode from the list of instructions since deleting one instruction reduces the offset of all the following instructions. Now if there are some jumps in the bytecode, their target offset should change too. So it is much easier to simply replace the unwanted instruction with `NOP`. Now we replace `['LOAD_GLOBAL', 'const']` with `NOP` and then replace `['STORE_ATTR', 'A']` with `['STORE_FAST', 'A']`. The final bytecode looks like this:

```
2          0 LOAD_CONST               1 (5)
           2 NOP
           4 STORE_FAST               1 (A)

3          6 LOAD_FAST                1 (A)
           8 LOAD_CONST               2 (1)
          10 BINARY_ADD
          12 STORE_FAST               1 (A)

4         14 LOAD_FAST                1 (A)
          16 LOAD_FAST                0 (x)
          18 BINARY_MULTIPLY
          20 RETURN_VALUE
```

Now line 2 is the equivalent of `a=2` in the source code, and executing this bytecode does not cause any run-time error. The loop also checks if the same variable is not declared as const twice. So if the variable declared as const already exists in the `constants` list, it will raise a custom exception. Now the only remaining thing is to

Open in app ↗ Sign up Sign In

Search Medium

reassignment of the constant variables. Any instruction like `['STORE_GLOBAL', 'A']` or `['STORE_FAST', 'A']` means that a reassignment is in the source code, so it will raise a custom exception to warn the user. The offset of the initial assignment of a const is required to make sure that the initial assignment is not considered as a reassignment.

As mentioned before, the bytecode should be changed before executing the code. So the function `add_const` needs to be called before calling the function `f`. For this reason, we place it inside a decorator. The decorator function `const` receives the target function `f` as its argument. It will first change the bytecode of `f` using `add_const` and then create a new code object with the modified bytecode. This code object will be assigned to `f`.

720 4

When we create the new code object, some of its attributes need to be modified. In the original function `const` is a global variable and `A` is an attribute, so both of them were added to the `co_names` tuple, and they should be removed from the `co_names` of the new code object. In addition, when an attribute like `A` is turned into a local variable, its name should be added to `co_varnames` tuple. The attribute `co_nlocals` gives the number of local variables (plus defined functions) and should be updated too. The other attributes remain the same. The decorator finally returns the target function with the new code object, and now the target function is ready for execution.

Understanding Python's bytecode allows you to get familiar with the low-level implementation of the Python compiler and virtual machine. If you know how your source code is converted to the bytecode, you can make better decisions about writing and optimizing your code. Bytecode injection is also a useful tool for code optimization and metaprogramming. I have only covered a small number of bytecode instructions in this article. You can refer to `dis` module's webpage to see the full list of Python's bytecode instructions. I hope that you have enjoyed reading this article. All the code listings of this article are available for download as a Jupyter notebook at: https://github.com/reza-bagheri/Understanding-Python-Bytecode

Python Bytecode Metaprogramming Disassembly Virtual Machine