Python behind the scenes #5: how variables are implemented in CPython

Published: Sat 14 November 2020
By *Victor Skvortsov*
tags: Python behind the scenes Python CPython

Consider a simple assignment statement in Python:

```
a = b
```

The meaning of this statement may seem trivial. What we do here is take the value of the name `b` and assign it to the name `a`, but do we really? This is an ambiguous explanation that gives rise to a lot of questions:

- What does it mean for a name to be associated with a value? What is a value?
- What does CPython do to assign a value to a name? To get the value?
- Are all variables implemented in the same way?

Today we'll answer these questions and understand how variables, so crucial aspect of a programming language, are implemented in CPython.

Note: In this post I'm referring to CPython 3.9. Some implementation details will certainly change as CPython evolves. I'll try to keep track of important changes and add update notes.

Start of the investigation

Where should we start our investigation? We know from the previous parts that to run Python code, CPython compiles it to bytecode, so let's start by looking at the bytecode to which `a = b` compiles:

```
$ echo 'a = b' | python -m dis

  1           0 LOAD_NAME                0 (b)
              2 STORE_NAME               1 (a)
...
```

Last time we learned that the CPython VM operates using the value stack. A typical bytecode instruction pops values from the stack, does something with them and pushes the result of the computation back onto the stack. The `LOAD_NAME` and `STORE_NAME` instructions are typical in that respect. Here's what they do in our example:

- `LOAD_NAME` gets the value of the name `b` and pushes it onto the stack.
- `STORE_NAME` pops the value from the stack and associates the name `a` with that value.

Last time we also learned that all opcodes are implemented in a giant `switch` statement in `Python/ceval.c`, so we can see how the `LOAD_NAME` and `STORE_NAME` opcodes work by studying the corresponding cases of that `switch`. Let's start with the `STORE_NAME` opcode since we need to associate a name with some value before we can get the value of that name. Here's the `case` block that executes the `STORE_NAME` opcode:

```
case TARGET(STORE_NAME): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *v = POP();
    PyObject *ns = f->f_locals;
    int err;
    if (ns == NULL) {
        _PyErr_Format(tstate, PyExc_SystemError,
                      "no locals found when storing %R", name);
        Py_DECREF(v);
        goto error;
    }
    if (PyDict_CheckExact(ns))
        err = PyDict_SetItem(ns, name, v);
    else
        err = PyObject_SetItem(ns, name, v);
    Py_DECREF(v);
    if (err != 0)
        goto error;
    DISPATCH();
}
```

Let's analyze what it does:

1. The names are strings. They are stored in a code object in a tuple called `co_names`. The `names` variable is just a shorthand for `co_names`. The argument of the `STORE_NAME` instruction is not a name but an index used to look up the name in `co_names`. The first thing the VM does is get the name, which it's going to assign a value to, from `co_names`.
2. The VM pops the value from the stack.
3. Values of variables are stored in a frame object. The `f_locals` field of a frame object is a mapping from the names of local variables to their values. The VM associates a name `name` with a value `v` by setting `f_locals[name] = v`.

We learn from this two crucial facts:

- Python variables are names mapped to values.
- Values of names are references to Python objects.

The logic for executing the `LOAD_NAME` opcode is a bit more complicated because the VM looks up the value of a name not only in `f_locals` but also in a few other places:

```
case TARGET(LOAD_NAME): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *locals = f->f_locals;
    PyObject *v;

    if (locals == NULL) {
        _PyErr_Format(tstate, PyExc_SystemError,
                      "no locals when loading %R", name);
        goto error;
    }

    // Look up the value in `f->f_locals`
    if (PyDict_CheckExact(locals)) {
        v = PyDict_GetItemWithError(locals, name);
        if (v != NULL) {
```

```
                Py_INCREF(v);
            }
            else if (_PyErr_Occurred(tstate)) {
                goto error;
            }
        }
        else {
            v = PyObject_GetItem(locals, name);
            if (v == NULL) {
                if (!_PyErr_ExceptionMatches(tstate, PyExc_KeyError))
                    goto error;
                _PyErr_Clear(tstate);
            }
        }

        // look up the value in `f->f_globals` and `f->f_builtins`
        if (v == NULL) {
            v = PyDict_GetItemWithError(f->f_globals, name);
            if (v != NULL) {
                Py_INCREF(v);
            }
            else if (_PyErr_Occurred(tstate)) {
                goto error;
            }
            else {
                if (PyDict_CheckExact(f->f_builtins)) {
                    v = PyDict_GetItemWithError(f->f_builtins, name);
                    if (v == NULL) {
                        if (!_PyErr_Occurred(tstate)) {
                            format_exc_check_arg(
                                    tstate, PyExc_NameError,
                                    NAME_ERROR_MSG, name);
                        }
                        goto error;
                    }
                    Py_INCREF(v);
                }
                else {
                    v = PyObject_GetItem(f->f_builtins, name);
                    if (v == NULL) {
                        if (_PyErr_ExceptionMatches(tstate, PyExc_KeyError)) {
                            format_exc_check_arg(
                                        tstate, PyExc_NameError,
                                        NAME_ERROR_MSG, name);
                        }
                        goto error;
                    }
                }
            }
        }
        PUSH(v);
        DISPATCH();
    }
```

This code translates to English as follows:

1. As for the `STORE_NAME` opcode, the VM first gets the name of a variable.
2. The VM looks up the value of the name in the mapping of local variables: `v = f_locals[name]`.
3. If the name is not in `f_locals`, the VM looks up the value in the dictionary of global variables `f_globals`. And if the name is not in `f_globals` either, the VM looks up the value in `f_builtins`. The `f_builtins` field of a frame object points to the dictionary of the `builtins` module, which contains built-in types, functions, exceptions and constants. If the name is not there, the VM gives up and sets the `NameError` exception.

4. If the VM finds the value, it pushes the value onto the stack.

The way the VM searches for a value has the following effects:

- We always have the names from the `builtin`'s dictionary, such as `int`, `next`, `ValueError` and `None`, at our disposal.
- If we use a built-in name for a local variable or a global variable, the new variable will shadow the built-in one.
- A local variable shadows the global variable with the same name.

Since all we need to be able to do with variables is to associate them with values and to get their values, you might think that the `STORE_NAME` and `LOAD_NAME` opcodes are sufficient to implement all variables in Python. This is not the case. Consider the example:

```
x = 1

def f(y, z):
    def _():
        return z

    return x + y + z
```

The function `f` has to load the values of variables `x`, `y` and `z` to add them and return the result. Note which opcodes the compiler produces to do that:

```
$ python -m dis global_fast_deref.py
...
  7          12 LOAD_GLOBAL              0 (x)
             14 LOAD_FAST                0 (y)
             16 BINARY_ADD
             18 LOAD_DEREF               0 (z)
             20 BINARY_ADD
             22 RETURN_VALUE
...
```

None of the opcodes are `LOAD_NAME`. The compiler produces the `LOAD_GLOBAL` opcode to load the value of `x`, the `LOAD_FAST` opcode to load the value of `y` and the `LOAD_DEREF` opcode to load the value of `z`. To see why the compiler produces different opcodes, we need to discuss two important concepts: namespaces and scopes.

Namespaces and scopes

A Python program consists of code blocks. A code block is a piece of code that the VM executes as a single unit. CPython distinguishes three types of code blocks:

- module
- function (comprehensions and lambdas are also functions)
- class definition.

The compiler creates a code object for every code block in a program. A code object is a structure that describes what a code block does. In particular, it contains the bytecode of a block. To execute a code object, CPython creates a state of execution for it called a frame object. Besides other things, a frame object contains name-value mappings such as `f_locals`, `f_globals` and `f_builtins`. These mappings are referred to as namespaces. Each code block introduces a namespace: its local namespace. The same name in a program may refer to different variables in different namespaces:

```
x = y = "I'm a variable in a global namespace"
```

```
def f():
    x = "I'm a local variable"
    print(x)
    print(y)

print(x)
print(y)
f()
```

```
$ python namespaces.py
I'm a variable in a global namespace
I'm a variable in a global namespace
I'm a local variable
I'm a variable in a global namespace
```

Another important notion is the notion of a scope. Here's what the Python documentation says about it:

> *A scope is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.*

We can think about a scope as a property of a name that tells where the value of that name is stored. The example of a scope is a local scope. The scope of a name is relative to a code block. The following example illustrates the point:

```
a = 1

def f():
    b = 3
    return a + b
```

Here, the name `a` refers to the same variable in both cases. From the function's perspective, it's a global variable, but from the module's perspective, it's both global and local. The variable `b` is local to the function `f`, but it doesn't exist at the module level at all.

The variable is considered to be local to a code block if it's bound in that code block. An assignment statement like `a = 1` binds the name `a` to `1`. An assignment statement, though, is not the only way to bind a name. The Python documentation lists a few more:

> *The following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, `for` loop header, or after as in a `with` statement or `except` clause. The `import` statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.*

Because any binding of a name makes the compiler think that the name is local, the following code raises an exception:

```
a = 1

def f():
    a += 1
    return a

print(f())
```

```
$ python unbound_local.py
...
    a += 1
UnboundLocalError: local variable 'a' referenced before assignment
```

The `a += 1` statement is a form of assignment, so the compiler thinks that `a` is local. To perform the operation, the VM tries to load the value of `a`, fails and sets the exception. To tell the compiler that `a` is global despite the assignment, we can use the `global` statement:

```
a = 1

def f():
    global a
    a += 1
    print(a)

f()
```

```
$ python global_stmt.py
2
```

Similarly, we can use the `nonlocal` statement to tell the compiler that a name bound in an enclosed (nested) function refers to a variable in an enclosing function:

```
a = "I'm not used"

def f():
    def g():
        nonlocal a
        a += 1
        print(a)
    a = 2
    g()

f()
```

```
$ python nonlocal_stmt.py
3
```

This is the work of the compiler to analyze the usage of names within a code block, take statements like `global` and `nonlocal` into account and produce the right opcodes to load and store values. In general, which opcode the compiler produces for a name depends on the scope of that name and on the type of the code block that is currently being compiled. The VM executes different opcodes differently. All of that is done to make Python variables work the way they do.

CPython uses four pairs of load/store opcodes and one more load opcode in total:

- `LOAD_FAST` and `STORE_FAST`
- `LOAD_DEREF` and `STORE_DEREF`
- `LOAD_GLOBAL` and `STORE_GLOBAL`
- `LOAD_NAME` and `STORE_NAME`; and
- `LOAD_CLASSDEREF`.

Let's figure out what they do and why CPython needs all of them.

LOAD_FAST and STORE_FAST

The compiler produces the `LOAD_FAST` and `STORE_FAST` opcodes for variables local to a function. Here's an example:

```
def f(x):
    y = x
    return y


$ python -m dis fast_variables.py
...
  2           0 LOAD_FAST                0 (x)
              2 STORE_FAST               1 (y)

  3           4 LOAD_FAST                1 (y)
              6 RETURN_VALUE
```

The `y` variable is local to `f` because it's bound in `f` by the assignment. The `x` variable is local to `f` because it's bound in `f` as its parameter.

Let's look at the code that executes the `STORE_FAST` opcode:

```
case TARGET(STORE_FAST): {
    PREDICTED(STORE_FAST);
    PyObject *value = POP();
    SETLOCAL(oparg, value);
    FAST_DISPATCH();
}
```

`SETLOCAL()` is a macro that essentially expands to `fastlocals[oparg] = value`. The `fastlocals` variable is just a shorthand for the `f_localsplus` field of a frame object. This field is an array of pointers to Python objects. It stores values of local variables, cell variables, free variables and the value stack. Last time we learned that the `f_localsplus` array is used to store the value stack. In the next section of this post we'll see how it's used to store values of cell and free variables. For now, we're interested in the first part of the array that's used for local variables.

We've seen that in the case of the `STORE_NAME` opcode, the VM first gets the name from `co_names` and then maps that name to the value on top of the stack. It uses `f_locals` as a name-value mapping, which is usually a dictionary. In the case of the `STORE_FAST` opcode, the VM doesn't need to get the name. The number of local variables can be calculated statically by the compiler, so the VM can use an array to store their values. Each local variable can the be associated with an index of that array. To map a name to a value, the VM simply stores the value in the corresponding index.

The VM doesn't need to get the names of variables local to a function to load and store their values. Nevertheless, it stores these names in a function's code object in the `co_varnames` tuple. Why? Names are necessary for debugging and error messages. They are also used by the tools such as `dis` that reads `co_varnames` to display names in parentheses:

```
              2 STORE_FAST               1 (y)
```

CPython provides the `locals()` built-in function that returns the local namespace of the current code block in the form of a dictionary. The VM doesn't keep such a dictionary for functions but it can built one on the fly by mapping keys from `co_varnames` to values from `f_localsplus`.

The `LOAD_FAST` opcode simply pushes `f_localsplus[oparg]` onto the stack:

```
case TARGET(LOAD_FAST): {
    PyObject *value = GETLOCAL(oparg);
    if (value == NULL) {
        format_exc_check_arg(tstate, PyExc_UnboundLocalError,
                             UNBOUNDLOCAL_ERROR_MSG,
                             PyTuple_GetItem(co->co_varnames, oparg));
        goto error;
    }
    Py_INCREF(value);
```

```
        PUSH(value);
        FAST_DISPATCH();
    }
```

The `LOAD_FAST` and `STORE_FAST` opcodes exist only for performance reasons. They are called `*_FAST` because the VM uses an array for the mapping, which works faster than a dictionary. What's the speed gain? Let's measure the difference between `STORE_FAST` and `STORE_NAME`. The following piece of code stores the value of the variable `i` 100 million times:

```
for i in range(10**8):
    pass
```

If we place it in a module, the compiler produces the `STORE_NAME` opcode. If we place it in a function, the compiler produces the `STORE_FAST` opcode. Let's do both and compare the running times:

```
import time


# measure STORE_NAME
times = []
for _ in range(5):
    start = time.time()
    for i in range(10**8):
        pass
    times.append(time.time() - start)

print('STORE_NAME: ' + ' '.join(f'{elapsed:.3f}s' for elapsed in sorted(times)))


# measure STORE_FAST
def f():
    times = []
    for _ in range(5):
        start = time.time()
        for i in range(10**8):
            pass
        times.append(time.time() - start)

    print('STORE_FAST: ' + ' '.join(f'{elapsed:.3f}s' for elapsed in sorted(times)))

f()
```

```
$ python fast_vs_name.py
STORE_NAME: 4.536s 4.572s 4.650s 4.742s 4.855s
STORE_FAST: 2.597s 2.608s 2.625s 2.628s 2.645s
```

Another difference in the implementation of `STORE_NAME` and `STORE_FAST` could theoretically affect these results. The case block for the `STORE_FAST` opcode ends with the `FAST_DISPATCH()` macro, which means that the VM goes to the next instruction straight away after it executes the `STORE_FAST` instruction. The case block for the `STORE_NAME` opcode ends with the `DISPATCH()` macro, which means that the VM may possible go to the start of the evaluation loop. At the start of the evaluation loop the VM checks whether it has to suspend the bytecode execution, for example, to release the GIL or to handle the signals. I've replaced the `DISPATCH()` macro with `FAST_DISPATCH()` in the case block for `STORE_NAME`, recompiled CPython and got similar results. So, the difference in times should indeed be explained by:

- the extra step to get a name; and
- the fact that a dictionary is slower than an array.

LOAD_DEREF and STORE_DEREF

There is one case when the compiler doesn't produce the `LOAD_FAST` and `STORE_FAST` opcodes for variables local to a function. This happens when a variable is used within a nested function.

```
def f():
    b = 1
    def g():
        return b


$ python -m dis nested.py
...
Disassembly of <code object f at 0x1027c72f0, file "nested.py", line 1>:
  2           0 LOAD_CONST               1 (1)
              2 STORE_DEREF              0 (b)

  3           4 LOAD_CLOSURE             0 (b)
              6 BUILD_TUPLE              1
              8 LOAD_CONST               2 (<code object g at 0x1027c7240, file "nested.py", line 3>)
             10 LOAD_CONST               3 ('f.<locals>.g')
             12 MAKE_FUNCTION            8 (closure)
             14 STORE_FAST               0 (g)
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE

Disassembly of <code object g at 0x1027c7240, file "nested.py", line 3>:
  4           0 LOAD_DEREF               0 (b)
              2 RETURN_VALUE
```

The compiler produces the `LOAD_DEREF` and `STORE_DEREF` opcodes for cell and free variables. A cell variable is a local variable referenced in a nested function. In our example, `b` is a cell variable of the function `f`, because it's referenced by `g`. A free variable is a cell variable from the perspective of a nested function. It's a variable not bound in a nested function but bound in the enclosing function or a variable declared `nonlocal`. In our example, `b` is a free variable of the function `g`, because it's not bound in `g` but bound in `f`.

The values of cell and free variables are stored in the `f_localsplus` array after the values of normal local variables. The only difference is that `f_localsplus[index_of_cell_or_free_variable]` points not to the value directly but to a cell object containing the value:

```
typedef struct {
    PyObject_HEAD
    PyObject *ob_ref;       /* Content of the cell or NULL when empty */
} PyCellObject;
```

The `STORE_DEREF` opcode pops the value from the stack, gets the cell of the variable specified by `oparg` and assigns `ob_ref` of that cell to the popped value:

```
case TARGET(STORE_DEREF): {
    PyObject *v = POP();
    PyObject *cell = freevars[oparg]; // freevars = f->f_localsplus + co->co_nlocals
    PyObject *oldobj = PyCell_GET(cell);
    PyCell_SET(cell, v); // expands to ((PyCellObject *)(cell))->ob_ref = v
    Py_XDECREF(oldobj);
    DISPATCH();
}
```

The `LOAD_DEREF` opcode works by pushing the contents of a cell onto the stack:

```
case TARGET(LOAD_DEREF): {
    PyObject *cell = freevars[oparg];
```

```
        PyObject *value = PyCell_GET(cell);
        if (value == NULL) {
          format_exc_unbound(tstate, co, oparg);
          goto error;
        }
        Py_INCREF(value);
        PUSH(value);
        DISPATCH();
    }
```

What's the reason to store values in cells? This is done to connect a free variable with the corresponding cell variable. Their values are stored in different namespaces in different frame objects but in the same cell. The VM passes the cells of an enclosing function to the enclosed function when it creates the enclosed function. The `LOAD_CLOSURE` opcode pushes a cell onto the stack and the `MAKE_FUNCTION` opcode creates a function object with that cell for the corresponding free variable. Due to the cell mechanism, when an enclosing function reassigns a cell variable, an enclosed function sees the reassignment:

```
def f():
    def g():
        print(a)
    a = 'assigned'
    g()
    a = 'reassigned'
    g()

f()
```

```
$ python cell_reassign.py
assigned
reassigned
```

and vice versa:

```
def f():
    def g():
        nonlocal a
        a = 'reassigned'
    a = 'assigned'
    print(a)
    g()
    print(a)

f()
```

```
$ python free_reassign.py
assigned
reassigned
```

Do we really need the cell mechanism to implement such behavior? Couldn't we just use the enclosing namespace to load and store values of free variables? Yes, we could, but consider the following example:

```
def get_counter(start=0):
    def count():
        nonlocal c
        c += 1
        return c

    c = start - 1
    return count
```

```
count = get_counter()
print(count())
print(count())


$ python counter.py
0
1
```

Recall that when we call a function, CPython creates a frame object to execute it. This example shows that an enclosed function can outlive the frame object of an enclosing function. The benefit of the cell mechanism is that it allows to avoid keeping the frame object of an enclosing function and all its references in memory.

LOAD_GLOBAL and STORE_GLOBAL

The compiler produces the `LOAD_GLOBAL` and `STORE_GLOBAL` opcodes for global variables in functions. The variable is considered to be global in a function if it's declared `global` or if it's not bound within the function and any enclosing function (i.e. it's neither local nor free). Here's an example:

```
a = 1
d = 1

def f():
    b = 1
    def g():
        global d
        c = 1
        d = 1
        return a + b + c + d
```

The `c` variable is not global to `g` because it's local to `g`. The `b` variable is not global to `g` because it's free. The `a` variable is global to `g` because it's neither local nor free. And the `d` variable is global to `g` because it's explicitly declared `global`.

Here's the implementation of the `STORE_GLOBAL` opcode:

```
case TARGET(STORE_GLOBAL): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *v = POP();
    int err;
    err = PyDict_SetItem(f->f_globals, name, v);
    Py_DECREF(v);
    if (err != 0)
        goto error;
    DISPATCH();
}
```

The `f_globals` field of a frame object is a dictionary that maps global names to their values. When CPython creates a frame object for a module, it assigns `f_globals` to the dictionary of the module. We can easily check this:

```
$ python -q
>>> import sys
>>> globals() is sys.modules['__main__'].__dict__
True
```

When the VM executes the `MAKE_FUNCTION` opcode to create a new function object, it assigns the `func_globals` field of that object to `f_globals` of the current frame object. When the function gets called, the VM creates a new frame object for it with `f_globals` set to `func_globals`.

The implementation of `LOAD_GLOBAL` is similar to that of `LOAD_NAME` with two exceptions:

- It doesn't look up values in `f_locals`.
- It uses cache to decrease the lookup time.

CPython caches the results in a code object in the `co_opcache` array. This array stores pointers to the `_PyOpcache` structs:

```
typedef struct {
    PyObject *ptr;  /* Cached pointer (borrowed reference) */
    uint64_t globals_ver;  /* ma_version of global dict */
    uint64_t builtins_ver; /* ma_version of builtin dict */
} _PyOpcache_LoadGlobal;

struct _PyOpcache {
    union {
        _PyOpcache_LoadGlobal lg;
    } u;
    char optimized;
};
```

The `ptr` field of the `_PyOpcache_LoadGlobal` struct points to the actual result of `LOAD_GLOBAL`. The cache is maintained per instruction number. Another array in a code object called `co_opcache_map` maps each instruction in the bytecode to its index minus one in `co_opcache`. If an instruction is not `LOAD_GLOBAL`, it maps the instruction to `0`, which means that the instruction is never cached. The size of the cache doesn't exceed 254. If the bytecode contains more than 254 `LOAD_GLOBAL` instructions, `co_opcache_map` maps extra instructions to `0` as well.

If the VM finds a value in the cache when it executes `LOAD_GLOBAL`, it makes sure that the `f_global` and `f_builtins` dictionaries haven't been modified since the last time the value was looked up. This is done by comparing `globals_ver` and `builtins_ver` with `ma_version_tag` of the dictionaries. The `ma_version_tag` field of a dictionary changes each time the dictionary is modified. See PEP 509 for more details.

If the VM doesn't find a value in the cache, it does a normal look up first in `f_globals` and then in `f_builtins`. If it eventually finds a value, it remembers current `ma_version_tag` of both dictionaries and pushes the value onto the stack.

LOAD_NAME and STORE_NAME (and LOAD_CLASSDEREF)

At this point you might wonder why CPython uses the `LOAD_NAME` and `STORE_NAME` opcodes at all. The compiler indeed doesn't produce these opcodes when it compiles functions. However, besides function, CPython has two other types of code blocks: modules and class definitions. We haven't talked about class definitions at all, so let's fix it.

First, it's crucial to understand that when we define a class, the VM executes its body. Here's what I mean:

```
class A:
    print('This code is executed')


$ python create_class.py
This code is executed
```

The compiler creates code objects for class definitions just as it creates code objects for modules and functions. What's interesting is that the compiler almost always produces the `LOAD_NAME` and `STORE_NAME` opcodes for variables within a class body. There are two rare exceptions to this rule: free variables and variables explicitly declared `global`.

The VM executes `*_NAME` opcodes and `*_FAST` opcodes differently. As a result, variables work differently in a class body than they do in a function:

```
x = 'global'

class C:
    print(x)
    x = 'local'
    print(x)


$ python class_local.py
global
local
```

On the first load, the VM loads the value of the `x` variable from `f_globals`. Then, it stores the new value in `f_locals` and, on the second load, loads it from there. If `C` was a function, we would get `UnboundLocalError: local variable 'x' referenced before assignment` when we call it, because the compiler would think that the `x` variable is local to `C`.

How do the namespaces of classes and functions interplay? When we place a function inside a class, which is a common practice to implement methods, the function doesn't see the names bound in the class' namespace:

```
class D:
    x = 1
    def method(self):
        print(x)

D().method()


$ python func_in_class.py
...
NameError: name 'x' is not defined
```

This is because the VM stores the value of `x` with `STORE_NAME` when it executes the class definition and tries to load it with `LOAD_GLOBAL` when it executes the function. However, when we place a class definition inside a function, the cell mechanism works as if we place a function inside a function:

```
def f():
    x = "I'm a cell variable"
    class B:
        print(x)

f()


$ python class_in_func.py
I'm a cell variable
```

There's a difference, though. The compiler produces the `LOAD_CLASSDEREF` opcode instead of `LOAD_DEREF` to load the value of `x`. The documentation of the `dis` module explains what `LOAD_CLASSDEREF` does:

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

Why does it check the locals dictionary first? In the case of a function, the compiler knows for sure if a variable is local or not. In the case of a class, the compiler cannot be sure. This is because CPython has metaclasses, and a metaclass may prepare a non-empty locals dictionary for a class by implementing the `__prepare__` method.

We can see now why the compiler produces the `LOAD_NAME` and `STORE_NAME` opcodes for class definitions but we also saw that it produces these opcodes for variables within the module's namespace, as in the `a = b` example. They work as

expected because module's `f_locals` and module's `f_globals` is the same thing:

```
$ python -q
>>> locals() is globals()
True
```

You might wonder why CPython doesn't use the `LOAD_GLOBAL` and `STORE_GLOBAL` opcodes in this case. Honestly, I don't know the exact reason, if there is any, but I have a guess. CPython provides the built-in `compile()`, `eval()` and `exec()` functions that can be used to dynamically compile and execute Python code. These functions use the `LOAD_NAME` and `STORE_NAME` opcodes within the top-level namespace. It makes perfect sense because it allows to execute code dynamically in a class body and get the same effect as if that code was written there:

```
a = 1

class A:
    b = 2
    exec('print(a + b)', globals(), locals())


$ python exec.py
3
```

CPython chose to always use the `LOAD_NAME` and `STORE_NAME` opcodes for modules. In this way, the bytecode the compiler produces when we run a module in a normal way is the same as when we execute the module with `exec()`.

How the compiler decides which opcode to produce

We learned in part 2 of this series that before the compiler creates a code object for a code block, it builds a symbol table for that block. A symbol table contains information about symbols (i.e. names) used within a code block including their scopes. The compiler decides which load/store opcode to produce for a given name based on its scope and the type of the code block that is currently being compiled. The algorithm can be summarized as follows:

1. Determine the scope of the variable:
    1. If the variable declared `global`, it's an explicit global variable.
    2. If the variable declared `nonlocal`, it's a free variable.
    3. If the variable is bound within the current code block, it's a local variable.
    4. If the variable is bound in the enclosing code block that is not a class definition, it's a free variable.
    5. Otherwise, it's a implicit global variable.
2. Update the scope:
    1. If the variable is local and and it's free in the enclosed code block, it's a cell variable.
3. Decide which opcode to produce:
    1. If the variable is a cell variable or a free variable, produce `*_DEREF` opcode; produce the `LOAD_CLASSDEREF` opcode to load the value if the current code block is a class definition.
    2. If the variable is a local variable and the current code block is a function, produce `*_FAST` opcode.
    3. If the variable is an explicit global variable or if it's an implicit global variable and the current code block is a function, produce `*_GLOBAL` opcode.
    4. Otherwise, produce `*_NAME` opcode.

You don't need to remember these rules. You can always read the source code. Check out `Python/symtable.c` to see how the compiler determines the scope of a variable, and `Python/compile.c` to see how it decides which opcode to produce.

Conclusion

The topic of Python variables is much more complicated than it may seem at first. A good portion of the Python documentation is related to variables, including a section on naming and binding and a section on scopes and namespaces. The top questions of the Python FAQ are about variables. I say nothing about questions on Stack Overflow. While the official resources give some idea why Python variables work the way they do, it's still hard to understand and remember all the rules. Fortunately, it's easier to understand how Python variables work by studying the source code of the Python implementation. And that's what we did today.

We've studied a group of opcodes that CPython uses to load and store values of variables. To understand how the VM executes other opcodes that actually compute something, we need to discuss the core of Python – Python object system. This is our plan for the next time.

If you have any questions, comments or suggestions, feel free to contact me at victor@tenthousandmeters.com