# Python internals: how callables work

March 23rd, 2012 at 10:53 am

*[The Python version described in this article is 3.x, more specifically - the 3.3 alpha release of CPython.]*

The concept of a *callable* is fundamental in Python. When thinking about what can be "called", the immediately obvious answer is functions. Whether it's user defined functions (written by you), or builtin functions (most probably implemented in C inside the CPython interpreter), functions were meant to be called, right?

Well, there are also methods, but they're not very interesting because they're just special functions that are bound to objects. What else can be called? You may, or may not be familiar with the ability to call *objects*, as long as they belong to classes that define the `__call__` magic method. So objects can act as functions. And thinking about this a bit further, classes are callable too. After all, here's how we create new objects:

```
class Joe:
  ... [contents of class]

joe = Joe()
```

Here we "call" `Joe` to create a new instance. So classes can act as functions as well!

It turns out that all these concepts are nicely united in the CPython implementation. Everything in Python is an object, and that includes every entity described in the previous paragraphs (user & builtin functions, methods, objects, classes). All these calls are served by a single mechanism. This mechanism is elegant and not that difficult to understand, so it's worth knowing about. But let's start at the beginning.

## Compiling calls

CPython executes our program in two major steps:

1. The Python source code is compiled to bytecode.
2. A VM executes that bytecode, using a toolbox of built-in objects and modules to help it do its job.

In this section I'll provide a quick overview of how the first step applies to making calls. I won't get too deep since these details are not the really interesting part I want to focus on in the article. If you want to learn more about the flow Python source undergoes in the compiler, read this.

Briefly, the Python compiler identifies everything followed by `(arguments...)` inside an expression as a call [1]. The AST node for this is `Call`. The compiler emits code for `Call` in the `compiler_call` function in `Python/compile.c`. In most cases, the `CALL_FUNCTION` bytecode instruction is going to be emitted. There are some variations I'm going to ignore for the purpose of the article. For example, if the call has "star args" – `func(a, b, *args)`, there's a special instruction for handling that – `CALL_FUNCTION_VAR`. It and other special instructions are just variations on the same theme.

## CALL_FUNCTION

So `CALL_FUNCTION` is the instruction we're going to focus on here. This is what it does:

> **CALL_FUNCTION(argc)**
>
> Calls a function. The low byte of *argc* indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters

are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack. Pops all function arguments, and the function itself off the stack, and pushes the return value.

CPython bytecode is evaluated by the the mammoth function PyEval_EvalFrameEx in Python/ceval.c. The function is scary but it's nothing more than a fancy dispatcher of opcodes. It reads instructions from the code object of the given frame and executes them. Here, for example, is the handler for CALL_FUNCTION (cleaned up a bit to remove tracing and timing macros):

```
TARGET(CALL_FUNCTION)
{
    PyObject **sp;
    sp = stack_pointer;
    x = call_function(&sp, oparg);
    stack_pointer = sp;
    PUSH(x);
    if (x != NULL)
        DISPATCH();
    break;
}
```

Not too bad – it's actually very readable. call_function does the actual call (we'll examine it in a bit), oparg is the numeric argument of the instruction, and stack_pointer points to the top of the stack [2]. The value returned by call_function is pushed back to the stack, and DISPATCH is just some macro magic to invoke the next instruction.

call_function is also in Python/ceval.c. It implements the actual functionality of the instruction. At 80 lines it's not very long, but long enough so I won't paste it wholly here. Instead I'll explain the flow in general and paste small snippets where relevant; you're welcome to follow along with the code open in your favorite editor.

## Any call is just an object call

The most important first step in understanding how calls work in Python is to ignore most of what call_function does. Yes, I mean it. The vast majority of the code in this function deals with optimizations for various common cases. It can be removed without hurting the correctness of the interpreter, only its performance. If we ignore all optimizations for the time being, all call_function does is decode the amount of arguments and amount of keyword arguments from the single argument of CALL_FUNCTION and forwards it to do_call. We'll get back to the optimizations later since they are interesting, but for the time being, let's see what the core flow is.

do_call loads the arguments from the stack into PyObject objects (a tuple for the positional arguments, a dict for the keyword arguments), does a bit of tracing and optimization of its own, but eventually calls PyObject_Call.

PyObject_Call is a super-important function. It's also available to extensions in the Python C API. Here it is, in all its glory:

```
PyObject *
PyObject_Call(PyObject *func, PyObject *arg, PyObject *kw)
{
    ternaryfunc call;

    if ((call = func->ob_type->tp_call) != NULL) {
        PyObject *result;
        if (Py_EnterRecursiveCall(" while calling a Python object"))
            return NULL;
        result = (*call)(func, arg, kw);
        Py_LeaveRecursiveCall();
        if (result == NULL && !PyErr_Occurred())
            PyErr_SetString(
                PyExc_SystemError,
                "NULL result without error in PyObject_Call");
        return result;
    }
    PyErr_Format(PyExc_TypeError, "'%.200s' object is not callable",
                 func->ob_type->tp_name);
    return NULL;
}
```

Deep recursion protection and error handling aside [3], PyObject_Call extracts the tp_call attribute [4] of the object's type and calls it. This is possible since tp_call holds a function pointer.

Let it sink for a moment. *This is it*. Ignoring all kinds of wonderful optimizations, this is what *all calls in Python* boil down to:

- Everything in Python is an object [5].

- Every object has a type; the type of an object dictates the stuff that can be done to/with the object.
- When an object is called, its type's `tp_call` attribute is called.

As a user of Python, your only direct interaction with `tp_call` is when you want your objects to be callable. If you define your class in Python, you have to implement the `__call__` method for this purpose. This method gets directly mapped to `tp_call` by CPython. If you define your class as a C extension, you have to assign `tp_call` in the type object of your class manually.

But recall that classes themselves are "called" to create new objects, so `tp_call` plays a role here as well. Even more fundamentally, when you define a class there is also a call involved – on the class's metaclass. This is an interesting topic and I'll cover it in a future article.

## Extra credit: Optimizations in CALL_FUNCTION

This part is optional, since the main point of the article was delivered in the previous section. That said, I think this material is interesting, since it provides examples of how some things you wouldn't usually think of as objects, actually *are* objects in Python.

As I mentioned earlier, we could just use `PyObject_Call` for every `CALL_FUNCTION` and be done with it. In reality, it makes sense to do some optimizations to cover common cases where that may be an overkill. `PyObject_Call` is a very generic function that needs all its arguments in special tuple and dictionary objects (for positional and keyword arguments, respectively). These arguments need to be taken from the stack and arranged in the containers `PyObject_Call` expects. In some common cases we can avoid a lot of this overhead, and this is what the optimizations in `call_function` are about.

The first special case `call_function` addresses is:

```
/* Always dispatch PyCFunction first, because these are
   presumed to be the most frequent callable object.
*/
if (PyCFunction_Check(func) && nk == 0) {
```

This handles objects of type `builtin_function_or_method` (represented by the `PyCFunction` type in the C implementation). There are a lot of those in Python, as the comment above notes. All functions and methods implemented in C, whether in the CPython interpreter, or in C extensions, fall into this category. For example:

```
>>> type(chr)
<class 'builtin_function_or_method'>
>>> type("".split)
<class 'builtin_function_or_method'>
>>> from pickle import dump
>>> type(dump)
<class 'builtin_function_or_method'>
```

There's an additional condition in that `if` – that the amount of keyword arguments passed to the function is zero. This allows some important optimizations. If the function in question accepts no arguments (marked by the `METH_NOARGS` flag when the function is created) or just a single object argument (`METH_O` flag), `call_function` doesn't go through the usual argument packing and can call the underlying function pointer directly. To understand how this is possible, reading about `PyCFunction` and the `METH_` flags in this part of the documentation is highly recommended.

Next, there's some special handling for methods of classes written in Python:

```
else {
  if (PyMethod_Check(func) && PyMethod_GET_SELF(func) != NULL) {
```

`PyMethod` is the internal object used to represent bound methods. The special thing about methods is that they carry around a reference to the object they're bound to. `call_function` extracts this object and places it on the stack, in preparation for what comes next.

Here's the rest of the call code (after it in `call_object` there's only some stack cleanup):

```
if (PyFunction_Check(func))
    x = fast_function(func, pp_stack, n, na, nk);
else
    x = do_call(func, pp_stack, na, nk);
```

`do_call` we've already met – it implements the most generic form of calling. However, there's one more optimization – if `func` is a `PyFunction` (an object used internally to represent functions defined in Python code), a separate path is taken – `fast_function`.

To understand what `fast_function` does, it's important to first consider what happens when a Python function is executed.

Simply put, its code object is evaluated (with `PyEval_EvalCodeEx` itself). This code expects its arguments to be on the stack. Therefore, in most cases there's no point packing the arguments into containers and unpacking them again. With some care, they can just be left on the stack and a lot of precious CPU cycles can be spared.

Everything else falls back to `do_call`. This, by the way, includes `PyCFunction` objects that *do* have keyword arguments. A curious aspect of this fact is that it's somewhat more efficient to not pass keyword arguments to C functions that either accept them or are fine with just positional arguments. For example [6]:

```
$ ~/test/python_src/33/python -m timeit -s's="a;b;c;d;e"' 's.split(";")'
1000000 loops, best of 3: 0.3 usec per loop
$ ~/test/python_src/33/python -m timeit -s's="a;b;c;d;e"' 's.split(sep=";")'
1000000 loops, best of 3: 0.469 usec per loop
```

This is a big difference, but the input is very small. For larger strings the difference is almost invisible:

```
$ ~/test/python_src/33/python -m timeit -s's="a;b;c;d;e"*1000' 's.split(";")'
10000 loops, best of 3: 98.4 usec per loop
$ ~/test/python_src/33/python -m timeit -s's="a;b;c;d;e"*1000' 's.split(sep=";")'
10000 loops, best of 3: 98.7 usec per loop
```

## Summary

The aim of this article was to discuss what it means to be callable in Python, approaching this concept from the lowest possible level – the implementation details of the CPython virtual machine. Personally, I find this implementation very elegant, since it unifies several concepts into a single one. As the extra credit section showed, Python entities we don't usually think of as objects – functions and methods – actually are objects and can also be handled in the same uniform manner. As I promised, future article(s) will dive deeper into the meaning of `tp_call` for creating new Python objects and classes.

[1] This is an intentional simplification – `()` serve other roles like class definitions (for listing base classes), function definitions (for listing arguments), decorators, etc – these are not in expressions. I'm also ignoring generator expressions on purpose.
[2] The CPython VM is a stack machine.
[3] `Py_EnterRecursiveCall` is needed where C code may end up calling Python code, to allow CPython keep track of its recursion level and bail out when it's too deep. Note that functions written in C don't have to abide by this recursion limit. This is why `do_call` special-cases `PyCFunction` before calling `PyObject_Call`.
[4] By "attribute" here I mean a structure field (sometimes also called "slot" in the documentation). If you're completely unfamiliar with the way Python C extensions are defined, go over this page.
[5] When I say that *everything* is an object – I mean it. You may think of objects as instances of classes you defined. However, deep down on the C level, CPython creates and juggles a lot of objects on your behalf. Types (classes), builtins, functions, modules – all these are represented by objects.
[6] This example will only run on Python 3.3, since the `sep` keyword argument to `split` is new in this version. In prior versions of Python `split` only accepted positional arguments.

Related posts:

1. Python internals: Symbol tables, part 1
2. Python internals: Symbol tables, part 2
3. Python internals: adding a new statement to Python
4. Python internals: Working with Python ASTs
5. Ruby as both a functional and an OO language

This entry was posted on Friday, March 23rd, 2012 at 10:53 and is filed under Articles, Python internals. You can follow any responses to this entry through the RSS 2.0 feed. You can skip to the end and leave a response. Pinging is currently not allowed.

## Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Write the number 4 here (required)

To post code with preserved formatting, enclose it in `backticks` (even multiple lines)

Submit Comment

Preview

A preview will appear here

---