Automatically inline Python function calls

August 01, 2013

experiments

Edit: Code is here on GitHub

Calling functions in Python can be expensive. Consider this example: there are two statements that are being timed, the first one calls a function that returns an integer while the second one calls a function that returns the result of a second function call which returns an integer.

```
tom@toms ~$ python -m timeit -n 10000000  -s "def get_n(): return
10000000 loops, best of 3: 0.145 usec per loop
tom@toms ~$ time python -m timeit -n 10000000 -s "get_n = lambda:
10000000 loops, best of 3: 0.335 usec per loop
```

The additional function call doubled the program execution time, despite not effecting the output of the function in any way. This got me thinking, how hard would it be to create a Python module that would inline functions, removing the calling overhead from certain functions you specify?

As it turns out, not that hard. **Note**: This is simply an experiment to see what's possible, don't even think about using this in real Python code (there are some serious limitations explained at the end). Check this out:

```
from inliner import inline

@inline
def add_stuff(x, y):
    return x + y
```

```
def call_func_args(num):
    return add_stuff(1, num)

import dis
dis.dis(call_func_args)
# Prints:
# 0 LOAD_CONST               1 (1)
# 3 LOAD_FAST                0 (num)
# 6 BINARY_ADD
# 7 RETURN_VALUE
```

The dis function prints out the bytecode operations for a Python function, which shows that the call_func_args function has been modified so that the add_stuff() call never takes place and instead the body of the add_stuff function has been inlined inside the call_func_args function. I've put the code on GitHub , have a look if you like. Below I will explain how it works, for those interested.

Diving in: Import hooks and the AST module

Python is an interpreted language, when you run a Python program the source code is parsed into an Abstract Syntax Tree which is then 'compiled' into bytecode. We need a way of modifying the AST of an imported module before it gets compiled, and as luck would have it Python provides powerful hooks into the import mechanism that allow you to write importers that grab code from the internet or restrict packages from being imported . Getting our claws into the import mechanism is as simple as this:

```
import sys, imp

class Loader(object):
    def __init__(self, module):
        self.module = module

    def load_module(self, fullname):
            return self.module

class Importer(object):
    def find_module(self, fullname, path):
        file, pathname, description = imp.find_module(
            fullname.split(".")[-1], path)
        module_contents = file.read()
        # We can now mess around with the module_contents.
        # and produce a module object
        return Loader(make_module(module_contents))

sys.meta_path.append(Importer())
```

Now whenever anything is imported our find_module() method will be called. This should return an object with a load_module() function, which returns the final module.

Modifying the AST

Python provides an AST module to modify Python AST trees. So inside our `find_module` function we can get the source code of the module we are importing, parse it into an AST representation and then modify it before compiling it. You can see this in action here .

First we need to find all functions that are wrapped by our inline decorator, which is pretty simple to do. The AST module provides a NodeVisitor and a NodeTransformer class you can subclass. For each different type of AST node a visit_NAME method will be called, which you can then choose to modify or pass along untouched. The InlineMethodLocator runs through all the function definition's in a tree and stores any that are wrapped by our inline decorator:

```
class InlineMethodLocator(ast.NodeVisitor):
    def __init__(self):
        self.functions = {}

    def visit_FunctionDef(self, node):
        if any(filter(lambda d: d.id == "inline", node.decorator_
            func_name = utils.getFunctionName(node)
            self.functions[func_name] = node
```

The next step after we have identified the functions we want to inline is to find where they are called, and then inline them. To do this we need to look for all Call nodes in our modules AST tree:

```
class FunctionInliner(ast.NodeTransformer):
    def __init__(self, functions_to_inline):
        self.inline_funcs = functions_to_inline

    def visit_Call(self, node):
        func = node.func
        func_name = utils.getFunctionName(func)
        if func_name in self.inline_funcs:
            func_to_inline = self.inline_funcs[func_name]
            transformer = transformers.getFunctionHandler(func_to
            if transformer is not None:
                node = transformer.inline(node, func_to_inline)

        return node
```

This visits all call objects and if we are calling a function we want to inline then we go grab a transformer object which will be responsible for the actual inlining. I've only written one transformer so far that works on simple functions (functions with 1 statement), but more can be added fairly easily. The simple function transformer simply returns the contents of the function and maps the functions values to the values of the calling function:

```
class SimpleFunctionHandler(BaseFunctionHandler):
    def inline(self, node, func_to_inline):
        # Its a simple function we have here. That means it is on
        # call with the inlined functions body
        body = func_to_inline.body[0]
        if isinstance(body, ast.Return):
            body = body.value

        return self.replace_params_with_objects(body, func_to_inl
```

Limitations

There are some serious limitations with this code:

1. **Inlined functions must have a unique name**: The AST provides us with no type information (as Python is dynamically typed), only the name of the function we are calling. That means without writing code that attempts to deduce the type of a class instance (no mean feat) then each function call must have a unique name.
2. **Only inlines functions in the same module**: To keep things simple only calls in the same module are inlined.
3. **Inlined class functions can't reference any double underscore attributes**: Accessing self.__attr is about as 'private' as you can get in Python. The attribute lookup is prefixed with the class name, which we can't easily detect while inlining.
4. **Everything will break**: Python is very dynamic, you may wish to replace functions at runtime. Obviously if the functions have been inlined then this won't have any effect.

Last modified September 21, 2019 (commit d7c8aae)