Eli Bendersky, Python internals: Working with Python ASTs

November 28th, 2009 at 1:02 pm

Starting with Python 2.5, the Python compiler (the part that takes your source-code and translates it to Python VM code for the VM to execute) works as follows [1]:

1. Parse source code into a parse tree (`Parser/pgen.c`)
2. Transform parse tree into an Abstract Syntax Tree (`Python/ast.c`)
3. Transform AST into a Control Flow Graph (`Python/compile.c`)
4. Emit bytecode based on the Control Flow Graph (`Python/compile.c`)

Previously, the only place one could tap into the compilation process was to obtain the parse tree with the `parser` module. But parse trees are much less convenient to use than ASTs for code transformation and generation. This is why the addition of the `_ast` module in Python 2.5 was welcome – it became much simpler to play with ASTs created by Python and even modify them. Also, the python built-in `compile` function can now accept an AST object in addition to source code.

Python 2.6 then took another step forward, including the higher-level `ast` module in its standard library. `ast` is a convenient Python-written toolbox to aid working with `_ast` [2]. All in all we now have a very convenient framework for processing Python source code. A full Python-to-AST parser is included with the standard distribution – what more could we ask? This makes all kinds of language transformation tasks with Python very simple.

What follows are a few examples of cool things that can be done with the new `_ast` and `ast` modules.

Manually building ASTs

```
import ast

node = ast.Expression(ast.BinOp(
                ast.Str('xy'),
                ast.Mult(),
                ast.Num(3)))

fixed = ast.fix_missing_locations(node)

codeobj = compile(fixed, '<string>', 'eval')

print eval(codeobj)
```

Let's see what is going on here. First we manually create an AST node, using the AST node classes exported by `ast` [3]. Then the convenient `fix_missing_locations` function is called to patch the `lineno` and `col_offset` attributes of the node and its children.

Another useful function that can help is `ast.dump`. Here's a formatted dump of the node we've created:

```
Expression(

  body=BinOp(

          left=Str(s='xy'),

          op=Mult(),

          right=Num(n=3)))
```

The most useful single-place reference for the various AST nodes and their structure is `Parser/Python.asdl` in the source distribution.

Breaking compilation into pieces

Given some source code, we first parse it into an AST, and then compile this AST into a code object that can be evaluated:

```
import ast

source = '6 + 8'

node = ast.parse(source, mode='eval')

print eval(compile(node, '<string>', mode='eval'))
```

Again, `ast.dump` can be helpful to show the AST that was created:

```
Expression(

  body=BinOp(

          left=Num(n=6),

          op=Add(),

          right=Num(n=8)))
```

Simple visiting and transformation of ASTs

```
import ast

class MyVisitor(ast.NodeVisitor):

    def visit_Str(self, node):

        print 'Found string "%s"' % node.s

class MyTransformer(ast.NodeTransformer):

    def visit_Str(self, node):

        return ast.Str('str: ' + node.s)
```

```
node = ast.parse('''

favs = ['berry', 'apple']

name = 'peter'

for item in favs:

    print '%s likes %s' % (name, item)

''')

MyTransformer().visit(node)

MyVisitor().visit(node)
```

This prints:

```
Found string "str: berry"

Found string "str: apple"

Found string "str: peter"

Found string "str: %s likes %s"
```

The visitor class implements methods that are called for relevant AST nodes (for example `visit_Str` is called for `Str` nodes). The transformer is a bit more complex. It calls relevant methods for AST nodes and then replaces them with the returned value of the methods.

To prove that the transformed code is perfectly valid, we can just compile and execute it:

```
node = ast.fix_missing_locations(node)

exec compile(node, '<string>', 'exec')
```

As expected [4], this prints:

```
str: str: peter likes str: berry

str: str: peter likes str: apple
```

Reproducing Python source from AST nodes

Armin Ronacher [5] wrote a module named `codegen` that uses the facilities of `ast` to print back Python source from an AST. Here's how to show the source for the node we transformed in the previous example:

```
import codegen

print codegen.to_source(node)
```

And the result:

```
favs = ['str: berry', 'str: apple']

name = 'str: peter'
```

```
for item in favs:

    print 'str: %s likes %s' % (name, item)
```

Yep, looks right. `codegen` is very useful for debugging or tools that transform Python code and want to save the results [6]. Unfortunately, the version you get from Armin's website isn't suitable for the `ast` that made it into the standard library. A slightly patched version of `codegen` that works with the standard 2.6 library can be downloaded here.

So why is this useful?

Many tools require parsing the source code of the language they operate upon. With Python, this task has been trivialized by the built-in methods to parse Python source into convenient ASTs. Since there's very little (if any) type checking done in a Python compiler, in classical terms we can say that a complete Python front-end is provided. This can be utilized in:

- IDEs for various "intellisense" needs
- Static code checking tools like pylint and pychecker
- Python code generators like pythoscope
- Alternative Python interpreters
- Compilers from Python to other languages

There are surely other uses I'm missing. If you're aware of a library/tool that uses `ast`, let me know.

[1] Taken from the excellent PEP 339. This PEP is well worth the read - it explains each of the 4 steps in details with useful pointers into the source code where more information can be obtained.

[2] `_ast` is implemented in `Python/Python-ast.[ch]` which can be obtained from the source distribution.

[3] Actually, they are exported by `_ast`, but `ast` does `from _ast import *`

[4] Why so many `str`:? It's not a mistake!

[5] The author of the `ast` module.

[6] For example, the pythoscope tool for auto generating unit-tests from code could probably benefit from `ast` and `codegen`. Currently it seems to be working on the level of Python parse trees instead.

Python internals: adding a new statement to Python

June 30th, 2010 at 7:18 pm

This article is an attempt to better understand how the front-end of Python works. Just reading documentation and source code may be a bit boring, so I'm taking a hands-on

approach here: I'm going to add an `until` statement to Python.

All the coding for this article was done against the cutting-edge Py3k branch in the Python Mercurial repository mirror.

The `until` statement

Some languages, like Ruby, have an `until` statement, which is the complement to `while` (`until num == 0` is equivalent to `while num != 0`). In Ruby, I can write:

```
num = 3

until num == 0 do

  puts num

  num -= 1

end
```

And it will print:

```
3

2

1
```

So, I want to add a similar capability to Python. That is, being able to write:

```
num = 3

until num == 0:

  print(num)

  num -= 1
```

A language-advocacy digression

This article doesn't attempt to suggest the addition of an `until` statement to Python. Although I think such a statement would make some code clearer, and this article displays how easy it is to add, I completely respect Python's philosophy of minimalism. All I'm trying to do here, really, is gain some insight into the inner workings of Python.

Modifying the grammar

Python uses a custom parser generator named `pgen`. This is a LL(1) parser that converts Python source code into a parse tree. The input to the parser generator is the file `Grammar/Grammar` [1]. This is a simple text file that specifies the grammar of Python.

Two modifications have to be made to the grammar file. The first is to add a definition for the `until` statement. I found where the `while` statement was defined (`while_stmt`), and added `until_stmt` below [2]:

```
compound_stmt: if_stmt | while_stmt | until_stmt | for_stmt | try_stmt | with_stmt |
funcdef | classdef | decorated

if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]

while_stmt: 'while' test ':' suite ['else' ':' suite]

until_stmt: 'until' test ':' suite
```

Note that I've decided to exclude the `else` clause from my definition of `until`, just to make it a little bit different (and because frankly I dislike the `else` clause of loops and don't think it fits well with the Zen of Python).

The second change is to modify the rule for `compound_stmt` to include `until_stmt`, as you can see in the snippet above. It's right after `while_stmt`, again.

When you run `make` after modifying `Grammar/Grammar`, notice that the `pgen` program is run to re-generate `Include/graminit.h` and `Python/graminit.c`, and then several files get re-compiled.

Modifying the AST generation code

After the Python parser has created a parse tree, this tree is converted into an AST, since ASTs are much simpler to work with in subsequent stages of the compilation process.

So, we're going to visit `Parser/Python.asdl` which defines the structure of Python's ASTs and add an AST node for our new `until` statement, again right below the `while`:

```
| While(expr test, stmt* body, stmt* orelse)

| Until(expr test, stmt* body)
```

If you now run `make`, notice that before compiling a bunch of files, `Parser/asdl_c.py` is run to generate C code from the AST definition file. This (like `Grammar/Grammar`) is another example of the Python source-code using a mini-language (in other words, a DSL) to simplify programming. Also note that since `Parser/asdl_c.py` is a Python script, this is a kind of bootstrapping - to build Python from scratch, Python already has to be available.

While `Parser/asdl_c.py` generated the code to manage our newly defined AST node (into the files `Include/Python-ast.h` and `Python/Python-ast.c`), we still have to write the code that converts a relevant parse-tree node into it by hand. This is done in the file `Python/ast.c`. There, a function named `ast_for_stmt` converts parse tree nodes for statements into AST nodes. Again, guided by our old friend `while`, we jump right into the big `switch` for handling compound statements and add a clause for `until_stmt`:

```
case while_stmt:

    return ast_for_while_stmt(c, ch);

case until_stmt:

    return ast_for_until_stmt(c, ch);
```

Now we should implement `ast_for_until_stmt`. Here it is:

```
static stmt_ty

ast_for_until_stmt(struct compiling *c, const node *n)

{

    /* until_stmt: 'until' test ':' suite */

    REQ(n, until_stmt);

    if (NCH(n) == 4) {

        expr_ty expression;

        asdl_seq *suite_seq;

        expression = ast_for_expr(c, CHILD(n, 1));

        if (!expression)

            return NULL;

        suite_seq = ast_for_suite(c, CHILD(n, 3));

        if (!suite_seq)

            return NULL;

        return Until(expression, suite_seq, LINENO(n), n->n_col_offset, c->c_arena);

    }

    PyErr_Format(PyExc_SystemError,

                 "wrong number of tokens for 'until' statement: %d",

                 NCH(n));

    return NULL;

}
```

Again, this was coded while closely looking at the equivalent `ast_for_while_stmt`, with the difference that for `until` I've decided not to support the `else` clause. As expected, the AST is created recursively, using other AST creating functions like `ast_for_expr` for the condition expression and `ast_for_suite` for the body of the `until` statement. Finally, a new node named `Until` is returned.

Note that we access the parse-tree node `n` using some macros like `NCH` and `CHILD`. These are worth understanding - their code is in `Include/node.h`.

Digression: AST composition

I chose to create a new type of AST for the `until` statement, but actually this isn't necessary. I could've saved some work and implemented the new functionality using composition of

existing AST nodes, since:

```
until condition:

   # do stuff
```

Is functionally equivalent to:

```
while not condition:

  # do stuff
```

Instead of creating the `Until` node in `ast_for_until_stmt`, I could have created a `Not` node with an `While` node as a child. Since the AST compiler already knows how to handle these nodes, the next steps of the process could be skipped.

Compiling ASTs into bytecode

The next step is compiling the AST into Python bytecode. The compilation has an intermediate result which is a CFG (Control Flow Graph), but since the same code handles it I will ignore this detail for now and leave it for another article.

The code we will look at next is `Python/compile.c`. Following the lead of `while`, we find the function `compiler_visit_stmt`, which is responsible for compiling statements into bytecode. We add a clause for `Until`:

```
case While_kind:

    return compiler_while(c, s);

case Until_kind:

    return compiler_until(c, s);
```

If you wonder what `Until_kind` is, it's a constant (actually a value of the `_stmt_kind` enumeration) automatically generated from the AST definition file into `Include/Python-ast.h`. Anyway, we call `compiler_until` which, of course, still doesn't exist. I'll get to it an a moment.

If you're curious like me, you'll notice that `compiler_visit_stmt` is peculiar. No amount of `grep`-ping the source tree reveals where it is called. When this is the case, only one option remains – C macro-fu. Indeed, a short investigation leads us to the `VISIT` macro defined in `Python/compile.c`:

```
#define VISIT(C, TYPE, V) {\

    if (!compiler_visit_ ## TYPE((C), (V))) \

        return 0; \
```

It's used to invoke `compiler_visit_stmt` in `compiler_body`. Back to our business, however…

As promised, here's `compiler_until`:

```
static int
compiler_until(struct compiler *c, stmt_ty s)
{
    basicblock *loop, *end, *anchor = NULL;
    int constant = expr_constant(s->v.Until.test);
    if (constant == 1) {
        return 1;
    }
    loop = compiler_new_block(c);
    end = compiler_new_block(c);
    if (constant == -1) {
        anchor = compiler_new_block(c);
        if (anchor == NULL)
            return 0;
    }
    if (loop == NULL || end == NULL)
        return 0;
    ADDOP_JREL(c, SETUP_LOOP, end);
    compiler_use_next_block(c, loop);
    if (!compiler_push_fblock(c, LOOP, loop))
        return 0;
    if (constant == -1) {
        VISIT(c, expr, s->v.Until.test);
        ADDOP_JABS(c, POP_JUMP_IF_TRUE, anchor);
    }
    VISIT_SEQ(c, stmt, s->v.Until.body);
    ADDOP_JABS(c, JUMP_ABSOLUTE, loop);
    if (constant == -1) {
        compiler_use_next_block(c, anchor);
        ADDOP(c, POP_BLOCK);
```

```
    }

    compiler_pop_fblock(c, LOOP, loop);

    compiler_use_next_block(c, end);

    return 1;

}
```

I have a confession to make: this code wasn't written based on a deep understanding of Python bytecode. Like the rest of the article, it was done in imitation of the kin `compiler_while` function. By reading it carefully, however, keeping in mind that the Python VM is stack-based, and glancing into the documentation of the `dis` module, which has a list of Python bytecodes with descriptions, it's possible to understand what's going on.

That's it, we're done... Aren't we?

After making all the changes and running `make`, we can run the newly compiled Python and try our new `until` statement:

```
>>> until num == 0:

...   print(num)

...   num -= 1

...

3

2

1
```

Voila, it works! Let's see the bytecode created for the new statement by using the `dis` module as follows:

```
import dis

def myfoo(num):

    until num == 0:

        print(num)

        num -= 1

dis.dis(myfoo)
```

Here's the result:

```
4           0 SETUP_LOOP              36 (to 39)

      >>    3 LOAD_FAST                0 (num)

            6 LOAD_CONST               1 (0)
```

```
             9 COMPARE_OP               2 (==)
            12 POP_JUMP_IF_TRUE        38
5           15 LOAD_NAME                0 (print)
            18 LOAD_FAST                0 (num)
            21 CALL_FUNCTION            1
            24 POP_TOP
6           25 LOAD_FAST                0 (num)
            28 LOAD_CONST               2 (1)
            31 INPLACE_SUBTRACT
            32 STORE_FAST               0 (num)
            35 JUMP_ABSOLUTE            3
      >>    38 POP_BLOCK
      >>    39 LOAD_CONST               0 (None)
            42 RETURN_VALUE
```

The most interesting operation is number 12: if the condition is true, we jump to after the loop. This is correct semantics for `until`. If the jump isn't executed, the loop body keeps running until it jumps back to the condition at operation 35.

Feeling good about my change, I then tried running the function (executing `myfoo(3)`) instead of showing its bytecode. The result was less than encouraging:

```
Traceback (most recent call last):
  File "zy.py", line 9, in <module>
    myfoo(3)
  File "zy.py", line 5, in myfoo
    print(num)
SystemError: no locals when loading 'print'
```

Whoa… this can't be good. So what went wrong?

The case of the missing symbol table

One of the steps the Python compiler performs when compiling the AST is create a symbol table for the code it compiles. The call to `PySymtable_Build` in `PyAST_Compile` calls into the symbol table module (`Python/symtable.c`), which walks the AST in a manner similar to the code generation functions. Having a symbol table for each scope helps the compiler figure out some key information, such as which variables are global and which are local to a scope.

To fix the problem, we have to modify the `symtable_visit_stmt` function in `Python/symtable.c`, adding code for handling `until` statements, after the similar code for `while` statements [3]:

```
case While_kind:
    VISIT(st, expr, s->v.While.test);
    VISIT_SEQ(st, stmt, s->v.While.body);
    if (s->v.While.orelse)
        VISIT_SEQ(st, stmt, s->v.While.orelse);
    break;
case Until_kind:
    VISIT(st, expr, s->v.Until.test);
    VISIT_SEQ(st, stmt, s->v.Until.body);
    break;
```

And now we really are done. Compiling the source after this change makes the execution of `myfoo(3)` work as expected.

Conclusion

In this article I've demonstrated how to add a new statement to Python. Albeit requiring quite a bit of tinkering in the code of the Python compiler, the change wasn't difficult to implement, because I used a similar and existing statement as a guideline.

The Python compiler is a sophisticated chunk of software, and I don't claim being an expert in it. However, I am really interested in the internals of Python, and particularly its front-end. Therefore, I found this exercise a very useful companion to theoretical study of the compiler's principles and source code. It will serve as a base for future articles that will get deeper into the compiler.

References

I used a few excellent references for the construction of this article. Here they are, in no particular order:

- PEP 339: Design of the CPython compiler – probably the most important and comprehensive piece of official documentation for the Python compiler. Being very short, it painfully displays the scarcity of good documentation of the internals of Python.
- "Python Compiler Internals" – an article by Thomas Lee
- "Python: Design and Implementation" – a presentation by Guido van Rossum
- Python (2.5) Virtual Machine, A guided tour – a presentation by Peter Tröger

[1] From here on, references to files in the Python source are given relatively to the root of the source tree, which is the directory where you run `configure` and `make` to build Python.

[2] This demonstrates a common technique I use when modifying source code I'm not familiar with: work by similarity. This principle won't solve all your problems, but it can definitely ease the process. Since everything that has to be done for `while` also has to be done for `until`, it serves as a pretty good guideline.

[3] By the way, without this code there's a compiler warning for `Python/symtable.c`. The compiler notices that the `Until_kind` enumeration value isn't handled in the switch statement of `symtable_visit_stmt` and complains. It's always important to check for compiler warnings!

Python internals: Symbol tables, part 1

September 18th, 2010 at 8:03 am

Introduction

This article is the first in a short series in which I intend to explain how CPython [1] implements and uses symbol tables in its quest to compile Python source code into bytecode. In this part I will explain what a symbol table is and show how the general concepts apply to Python. In the second part I will delve into the implementation of symbol tables in the core of CPython.

So what is a symbol table?

As usual, it's hard to beat Wikipedia for a succinct definition:

In computer science, a symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.

Symbol tables are used by practically all compilers. They're especially important in statically typed languages where all variables have types and type checking by the compiler is an important part of the front-end.

Consider this C code:

```
int main()

{

    int aa, bb;

    bb = *aa;

    {

        int* aa;
```

```
        bb = *aa;

    }

    return 0;

}
```

There are two distinct assignments of the form `bb = *aa` here, but only the second one is legal. The compiler throws the following error when it sees the first one:

```
error: invalid type argument of ‘unary *’ (have ‘int’)
```

How does the compiler know that the `*` operator is given an argument of type `int`, which is invalid for this operator? The answer is: the symbol table. The compiler sees `*aa` and asks itself what the type of `aa` is. To answer this question it consults the symbol table it constructed earlier. The symbol table contains the declared types for all variables the compiler encountered in the code.

This example demonstrates another important concept – for most languages a single symbol table with information about all variables won't do. The second assignment is valid, because in the internal scope created by the curly braces `aa` is redefined to be of a pointer type. Thus, to correctly compile such code the C compiler has to keep a separate symbol table per scope [2].

A digression: "variables" in Python

So far I've been using the term "variable" liberally. Just to be on the safe side, let's clarify what is meant by variable in Python. Formally, Python doesn't really have variables in the sense C has. Rather, Python has symbolic names bound to objects:

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

bb = aa

aa[0] = 666
```

In this code, `aa` is a name bound to a list object. `bb` is a name bound to the same object. The third line modifies the list through `aa`, and if we print out `bb` we'll see the modified list as well.

Now, once this is understood, I will still use the term "variable" from time to time since it's occasionally convenient and everybody's used to it anyway.

Symbol tables for Python code

Alright, so symbol tables are very useful for type checking. But Python doesn't have compile-time type checking (duck typing FTW!), so what does CPython need symbol tables for?

The CPython compiler still has to resolve what kinds of variables are used in the code. Variables in Python can be local, global or even bound by a lexically enclosing scope. For

example:

```
def outer(aa):

    def inner():

        bb = 1

        return aa + bb + cc

    return inner
```

The function `inner` uses three variables: `aa`, `bb` and `cc`. They're all different from Python's point of view: `aa` is lexically bound in `outer`, `bb` is locally bound in `inner` itself, and `cc` is not bound anywhere in sight, so it's treated as global. The bytecode generated for `inner` shows clearly the different treatment of these variables:

```
5           0 LOAD_CONST               1 (1)
            3 STORE_FAST               0 (bb)

6           6 LOAD_DEREF               0 (aa)
            9 LOAD_FAST                0 (bb)
           12 BINARY_ADD
           13 LOAD_GLOBAL              0 (cc)
           16 BINARY_ADD
           17 RETURN_VALUE
```

As you can see, different opcodes are used for loading the variables onto the stack prior to applying `BINARY_ADD`. `LOAD_DEREF` is used for `aa`, `LOAD_FAST` for `bb` and `LOAD_GLOBAL` for `cc`.

At this point, there are three different directions we can pursue on our path to deeper understanding of Python:

5. Figure out the exact semantics of variables in Python – when are they local, when are they global and what exactly makes them lexically bound.
6. Understand how the CPython compiler knows the difference.
7. Learn about the different bytecode opcodes for these variables and how they affect the way the VM runs code.

I won't even try going into (1) since it's a broad topic completely out of the scope of this article. There are plenty of resources online – start with the official and continue Googling until you're fully enlightened. (3) is also out of scope as I'm currently focusing on the front-end of CPython. If you're interested, there's an excellent series of in-depth articles on Python focusing on the back-end, with a nice treatment of this very issue.

To answer (2) we need to understand how CPython uses symbol tables, which is what this

series of articles is about.

Where symbol tables fit in

A high-level view of the front-end of CPython is:

1. Parse source code into a parse tree
2. Transform parse tree into an Abstract Syntax Tree
3. Transform AST into a Control Flow Graph
4. Emit bytecode based on the Control Flow Graph

Symbol tables are created in step 3. The compiler builds a symbol table from the AST representing the Python source code. This symbol table, in conjunction with the AST is then used to generate the control flow graph (CFG) and ultimately the bytecode.

Exploring the symbol table

CPython does a great job exposing some of its internals via standard-library modules [3]. Symbol tables is yet another internal data structure that can be explored from the outside in pure Python code, with the help of the `symtable` module. From its description:

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. symtable provides an interface to examine these tables.

The `symtable` module provides a lot of information on the various identifiers encountered in Python code. Apart from telling their scope, it allows us to find out which variables are referenced in their scope, assigned in their scope, define new namespaces (like functions) and so on. To help with exploring the symbol table I've written the following function that simplifies working with the module:

```
def describe_symbol(sym):
    assert type(sym) == symtable.Symbol
    print("Symbol:", sym.get_name())
    for prop in [
            'referenced', 'imported', 'parameter',
            'global', 'declared_global', 'local',
            'free', 'assigned', 'namespace']:
        if getattr(sym, 'is_' + prop)():
            print('    is', prop)
```

Let's see what it has to say about the `inner` function from the example above:

```
Symbol: aa

    is referenced

    is free

Symbol: cc

    is referenced

    is global

Symbol: bb

    is referenced

    is local

    is assigned
```

Indeed, we see that the symbol table marks `aa` as lexically bound, or "free" (more on this in the next section), `bb` as local and `cc` as global. It also tells us that all these variables are referenced in the scope of `inner` and that `bb` is assigned in that scope.

The symbol table contains other useful information as well. For example, it can help distinguish between explicitly declared globals and implicit globals:

```
def outer():

    global gg

    return ff + gg
```

In this code both `ff` and `gg` are global to `outer`, but only `gg` was explicitly declared `global`. The symbol table knows this – the output of `describe_symbol` for this function is:

```
Symbol: gg

    is referenced

    is global

    is declared_global

Symbol: ff

    is referenced

    is global
```

Free variables

Unfortunately, there's a shorthand in the core of Python that may initially confuse readers as to exactly what constitutes a "free" variable. Fortunately, it's a very slight confusion that's easy to put in order. The execution model reference says:

If a variable is used in a code block but not defined there, it is a free variable.

This is consistent with the formal definition. In the source, however, "free" is actually used as a shorthand for "lexically bound free variable" (i.e. variables for which a binding has been found in an enclosing scope), with "global" being used to refer to all remaining free variables. So when reading the CPython source code it is important to remember that the full set of free variables includes both the variables tagged specifically as "free", as well as those tagged as "global".

Thus, to avoid a confusion I say "lexically bound" when I want to refer to the variables actually treated in CPython as free.

Catching errors

Although Python is duck-typed, some things can still be enforced at compile-time. The symbol table is a powerful tool allowing the compiler to catch some errors [4].

For example, it's not allowed to declare function parameters as global:

```
def outer(aa):

    global aa
```

When compiling this function, the error is caught while constructing the symbol table:

```
Traceback (most recent call last):

  File "symtab_1.py", line 33, in <module>

    table = symtable.symtable(code, '<string>', 'exec')

  File "symtable.py", line 13, in symtable

    raw = _symtable.symtable(code, filename, compile_type)

  File "<string>", line 2

SyntaxError: name 'aa' is parameter and global
```

The symbol table is useful here since it knows that `aa` is a parameter in the scope of `outer` and when `global aa` is encountered it's a sure sign of an error.

Other errors are handled by the symbol table: duplicate argument names in functions, using `import *` inside functions, returning values inside generators, and a few more.

Conclusion

This article serves mainly as an introduction for the next one, where I plan to explore the actual implementation of symbol tables in the core of CPython. A symbol table is a tool designed to solve some problems for the compiler, and I hope this article did a fair job describing a few of these problems and related terminology.

Special thanks to Nick Coghlan for reviewing this article.

[1] You'll note that in this article I'm using the terms Python and CPython interchangeably. They're not the same – by Python I mean the language (version 3.x) and by CPython I mean the official C implementation of the compiler + VM. There are several implementations of the Python language in existence, and while they all implement the same specification they may do it differently.

[2] It's even more complex than that, but we're not here to talk about C compilers. In the next article I will explain exactly the structure of symbol tables used by CPython.

[3] I've previously discussed how to use the `ast` module to tap into the compilation process of CPython.

[4] Actually, if you `grep` the CPython source, you'll find out that a good proportion of `SyntaxError` exceptions thrown by the compiler are from `Python/symtable.c`.

Python internals: Symbol tables, part 2

September 20th, 2010 at 7:59 am

This is the second part of the article. Make sure you read the first part before this one.

In this article I will explain how symbol tables are implemented in the CPython core [1]. The implementation itself is contained mainly in two files, the header `Include/symtable.h` and the C source file `Python/symtable.c`.

My strategy for understanding the implementation will follow Fred Brooks' advice from his book The Mythical Man-Month:

Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.

A more modern translation would be: "The key to understanding a program is to understand its data structures. With that in hand, the algorithms usually become obvious."

This is especially true when the data structures of some module closely model the problem this module intends to solve, and the algorithms' job is to correctly create and use these data structures. Fortunately, this is exactly the case in CPython's implementation of symbol tables. Without further ado, let's delve in.

Symbol table entries

The key data structure to study is the symbol table entry, named `PySTEntryObject` [2]:

```
typedef struct _symtable_entry {

    PyObject_HEAD

    PyObject *ste_id;

    PyObject *ste_symbols;
```

```
    PyObject *ste_name;
    PyObject *ste_varnames;
    PyObject *ste_children;
    _Py_block_ty ste_type;
    int ste_unoptimized;
    int ste_nested;
    unsigned ste_free : 1;
    unsigned ste_child_free : 1;
    unsigned ste_generator : 1;
    unsigned ste_varargs : 1;
    unsigned ste_varkeywords : 1;
    unsigned ste_returns_value : 1;
    int ste_lineno;
    int ste_opt_lineno;
    int ste_tmpname;
    struct symtable *ste_table;
} PySTEntryObject;
```

Before I explain what each field in the structure means, some background is in order. An entry object is created for each block in the Python source code. A block is defined as:

[...] A piece of Python program text that is executed as a unit. The following are blocks: a module, a function body and a class definition. [...]

Therefore, if we have the following definition in our Python source:

```
def outer(aa):
    def inner():
        bb = 1
        return aa + bb + cc
    dd = aa + inner()
    return dd
```

The definition of `outer` creates a block with its body. So does the definition of `inner`. In addition, the top-level module in which `outer` is defined is also a block. All these blocks are represented by symbol table entries.

In essence, each entry is a symbol table on its own, containing information on the symbols in the block it represents. These entries are linked together into hierarchies, to represent nested blocks.

Once again, the `symtable` module can be used to explore these entries. In the first part of the article I used it to show what CPython knows about the symbols, but here I want to show how entries work. Here's a function that uses `symtable` to show how entries nest:

```
def describe_symtable(st, recursive=True, indent=0):

    def print_d(s, *args):

        prefix = ' ' * indent

        print(prefix + s, *args)

    assert isinstance(st, symtable.SymbolTable)

    print_d('Symtable: type=%s, id=%s, name=%s' % (

                st.get_type(), st.get_id(), st.get_name()))

    print_d('  nested:', st.is_nested())

    print_d('  has children:', st.has_children())

    print_d('  identifiers:', list(st.get_identifiers()))

    if recursive:

        for child_st in st.get_children():

            describe_symtable(child_st, recursive, indent + 5)
```

When executed on the symbol table created from the Python code we saw earlier, it prints out:

```
Symtable: type=module, id=164192096, name=top

  nested: False

  has children: True

  identifiers: ['outer']

     Symtable: type=function, id=164192056, name=outer

       nested: False

       has children: True

       identifiers: ['aa', 'dd', 'inner']

          Symtable: type=function, id=164191736, name=inner

            nested: True

            has children: False
```

```
        identifiers: ['aa', 'cc', 'bb']
```

Note how entries are nested. The top-level entry representing the module has the entry for `outer` as its child, which in turn has the entry for `inner` as its child.

With this understood, we can go over the fields of the `PySTEntryObject` struct and explain what each one means [3]. Note that I use the terms "block" and "entry" interchangeably.

- `ste_id`: a unique integer ID for the entry taken as the Python object ID of the AST node it was created from.
- `ste_symbols`: the actual symbol table of this entry, a Python `dict` object mapping symbol names to flags that describe them. See the `describe_symbol` function in part 1 of the article to understand what information is stored in the flags for each symbol. All the symbols that are used in the block (whether defined or only referenced) are mapped here.
- `ste_name`: block name (if applicable). For example, for the function `outer`, the name is `outer`. Used primarily for debugging and error reporting.
- `ste_varnames`: the name of the field (as well as the comment that follows it) is somewhat misleading [4]. It's actually a list of the parameters of the block. For example, for the `outer` function in the example it's a list with the single name `aa`.
- `ste_children`: list of child blocks (also `PySTEntryObject` objects). As we saw earlier, blocks are nested, modeling the nesting of scopes in the Python source.
- `ste_type`: a value of the enumeration type `_Py_block_ty` which has three possible values: `FunctionBlock`, `ClassBlock`, `ModuleBlock` for the three kinds of blocks supported by the symbol table.
- `ste_unoptimized`: this flag helps deal with some special blocks (top-level and those containing `import *`). It's safe to ignore it for our purposes.
- `ste_nested`: An integer flag: 1 if it's a function block nested in some other function block (like our `inner` function), 0 otherwise.
- Next come some other flags with information about the block: `ste_free`, `ste_child_free`, `ste_generator`, `ste_varargs`, `ste_varkeywords` and `ste_returns_value`. The comments after these flags describe them quite well.
- `ste_lineno`: number of the first line of the block in the Python source – taken directly from the AST.
- `ste_opt_lineno`: related to the `ste_unoptimized` flag. Again, we'll ignore it for now.
- `ste_tmpname`: used to generate temporary names of variables in comprehensions
- `ste_table`: link to the `symtable` object this entry is part of.

As I mentioned above, the entry is the key data structure in CPython's symbol table code.

When the symbol table creation algorithm finishes running, we get a set of inter-linked (for nesting) entries which contain information about all the symbols defined and used in our code. These entries are used in later stages of the compiler to generate bytecode from the AST.

symtable

`symtable` is less important for the usage of symbol tables by the compiler, but it's essential for the initial construction of a symbol table from the AST.

The symbol table construction algorithm uses an instance of the `symtable` structure to keep its state as it walks the AST recursively and builds entries for the blocks it finds.

Here are the fields of `symtable` annotated:

- `st_filename`: name of the file being compiled – used for generating meaningful warnings and errors.
- `st_cur`: the current entry (`PySTEntryObject`) the construction algorithm is in. Think of it (together with `st_stack`) as the current state of the algorithm as it walks the AST nodes recursively to create new entries.
- `st_top`: top-level entry for the module. Serves as the entry-point for the second pass of the symbol table construction algorithm.
- `st_blocks`: a dict mapping entry IDs (`ste_id`) to entry objects. Can be used to find the entry for some AST node.
- `st_stack`: a list representing a stack of entries. Used when working on nested blocks.
- `st_global`: direct link to the `ste_symbols` dict of the top-level module entry (`st_top`). Useful for accessing global (module-level) definitions from anywhere.
- `st_nblocks` / `st_future`: these fields aren't being used anywhere in the source so we'll ignore them.
- `st_private`: the name of the current class. This field is used for "mangling" private variables in classes [5].

Midway recap: what we have so far

Now that we have the data structures covered, it should be much simpler to understand the code implementing symbol table construction. There's no need to memorize the meanings of all fields – they can serve as a reference when reading the rest of the article and/or the source code. But it's definitely recommended to go over them at least once to have a general sense of what each data structure contains.

Constructing the symbol table: algorithm outline

Once we have a clear notion in our head of what information the symbol table eventually

contains, the algorithm for constructing it is quite obvious. In the following diagram I've sketched an outline:

The algorithm is divided into two passes [6]. The first pass creates the basic structure of entries modeling the blocks in the source code and marks some of the simple information easily available directly in the AST – for example, which symbols are defined and referenced in each block.

The second pass walks the symbol table and infers the less obvious information about symbols: which symbols are free, which are implicitly global, etc.

Constructing the symbol table: first pass

The implementation of the first pass of the algorithm consumes a good chunk of the source-code of `Python/symtable.c`, but it's actually quite simple to understand, because the bulk of it exists to deal with the large variety of AST nodes Python generates.

First, let's take a look at how new blocks get created. When a new block-defining-statement (such as the top-level module or a function/class definition) is encountered, `symtable_enter_block` is called. It creates a new block and handles nesting by using `st->st_stack` [7], making sure `st->st_cur` always points to the currently processed block. The complementary function `symtable_exit_block` is called when a block has been processed. It uses `st->st_stack` to roll `st->st_cur` back to the enclosing block.

The next function that's important to understand is `symtable_add_def`. Its name could be clearer, though, since it's being called not only for symbol definitions but also for symbol uses (recall that the symbol table keeps track of which symbols are being used in which blocks). What it does is basically add a flag to the symbols dict (`ste_symbols`) of an entry that specifies the symbol's definition or use.

The rest of the code of the first pass is just a bunch of AST visiting functions that are implemented in a manner similar to other AST walkers in the CPython code base. There's a `symtable_visit_<type>` function for every major `<type>` of AST node the symbol table is interested in, along with a family of `VISIT_*` macros that help keep the code shorter.

I will pick a couple of examples to demonstrate the stuff explained earlier.

The most interesting would be handling function definitions in `symtable_visit_stmt`, under `case FunctionDef_kind`:

- First the function's name is added as a definition to the current block
- Next, default values, annotations and decorators are recursively visited.
- Since the function definition creates a new block, `symtable_enter_block` is called, only after which the actual function arguments and body get visited. Then, `symtable_exit_block` is called to get back to the parent block.

Another interesting example is the `case Yield_kind` code of `symtable_visit_expr` that handles `yield` statements. After visiting the yielded value (if any), the current block is marked as a

generator. Since returning values from generators isn't allowed, the algorithm raises a syntax error if the block is also marked as returning a value [8].

The output of the first pass is a structurally complete symbol table, consisting of nested entries that model the blocks in the source code. At this stage the symbol table contains only partial information about symbols, however. Although it already maps all symbols defined and used in the code, and even flags special cases such as global symbols and generators, it still lacks some information, such as the distinction between free symbols that are defined in enclosing scopes and implicitly global symbols. This is the job of the second pass [9].

Constructing the symbol table: second pass

The second pass is actually documented not badly in the comments inside `Python/symtable.c`. However, these comments are scattered all over the place, so I'll try to provide a quick summary that should serve as a good starting point for reading the commented source. I will use a concrete example along with my explanation to aid understanding.

Here's our sample Python code once again:

```
def outer(aa):

    def inner():

        bb = 1

        return aa + bb + cc

    dd = aa + inner()

    return dd
```

Let's focus on the symbols used in `inner`. After the first pass, the algorithm knows the following:

8. `bb` is bound in `inner`
9. `aa` is bound (as a parameter) in `outer`
10. `aa`, `bb`, `cc` are used in `inner`

With this information, the algorithm should infer that `aa` is free and `cc` is global. For this, it runs another analysis, this time on the entries, pushing information from parent blocks into child blocks and back up again.

For example, when it analyzes `inner`, the second pass takes along a set of all variables bound in `inner`'s parent (enclosing) scopes - `aa` and `dd`. In the analysis of `inner`, seeing that `aa` is used but not defined in `inner` but is defined in an enclosing scope, it can go on and mark `aa` as a free variable in the `ste_symbols` dict for `inner`'s entry. Similarly, seeing that `cc` is not defined in any enclosing scope, it marks it global.

The information has to travel back up as well. An implementation detail of CPython which

we won't get into in this article is "cells" which are used to implement free variables. For the purposes of the symbol table, all we need to know is that the symbol table is required to mark which variables serve as cells in an enclosing scopes for free variables in child scopes. So the algorithm should mark `aa` as a cell variable in `outer`. For this, it should first analyze `inner` and find out it's free there.

The most important function of the second pass is `analyze_block` - it is being called over and over again for each block in the symbol table. The arguments of `analyze_block` are:

- `ste`: The symbol table entry for the block to analyze
- `bound`: a set of all variables bound in enclosing scopes
- `global`: a set of all variables declared global in enclosing scopes
- `free`: output from the function - the set of all free variables in this entry and its enclosed scopes

Using a couple of auxiliary functions, `analyze_block` calls itself recursively on the child blocks of the given block, passing around and gathering the required information. Apart from creating the `free` set for the enclosing scope, `analyze_block` modifies `ste` as it finds new information about symbols in it.

Special thanks to Nick Coghlan for reviewing this article.

[1] The specific version I'm describing is a relatively up-to-date snapshot of the `py3k` branch, in other words the "cutting edge" of CPython.

[2] The struct declaration in `Include/symtable.h` has short comments after each field. I've removed those intentionally.

[3] It helps to have the relevant code open in a separate window while reading this article, in particular this section. Also keep in mind that many of the fields are implemented as actual Python objects (created by the CPython C API), so when I say a list or a dict it's an actual Python list or a dict, the interaction with which is via its C API.

[4] The reason for the confusing name may be this list's later role in the compiler's flow in the creation of a code object. The `co_varnames` field of a code object contains the names of all local variables in a block starting with the parameter names (actually taken by the compiler from the `ste_varnames` field of the symbol table).

[5] Look up "python private name mangling" on Google if you're not familiar with the mangling of private (marked by starting with two or more underscores) identifiers in Python classes.

[6] You may be wondering why two passes are required and a single one isn't enough. It's a good question that ponders some philosophical and stylistic issues behind the practice of software development. My guess is that while the task could've been accomplished in a single pass, the multi-pass approach allows the code to be simplified at the cost of a very modest (if any) hit in performance. Sometimes splitting algorithms into multiple steps makes them much easier to grasp - readability and maintainability are important traits of well-written code.

Consider, for example, this code:

```
def outer():

    def inner():

        bb = 1

        return aa + bb + cc

    aa = 2

    return inner
```

Here, `aa` is free in `inner`, lexically bound to the `aa` defined in `outer`. But a one-pass algorithm would see `inner` before it ever saw the definition of `aa`, so to implement this correctly it would be forced to use some complex data structure to remember the variable uses it saw. With a two-pass algorithm, handling this case is much simpler.

[7] Throughout the code `st` refers to the `symtable` object that's being passed into functions, and `ste` to an entry object.

[8] As an exercise, think whether this ensures that all generators returning values are caught as errors. Hint #1: what happens if the `yield` is found before the `return`? Hint #2: Locate the code for the handling of `return` statements by the first pass.

[9] Here and on I'm presenting a somewhat simplified view of the second pass. There are further complications like variables declared global in enclosing scopes and referenced in enclosed scopes, that affect the results. I'm ignoring this on purpose to try and focus on the main aim of the second pass. Any source code has important special cases and trying to accurately summarize 1000 lines of code in a few paragraphs of text is an endeavor destined to fail.

Understanding UnboundLocalError in Python

May 15th, 2011 at 5:43 am

If you're closely following the Python tag on StackOverflow, you'll notice that the same question comes up at least once a week. The question goes on like this:

```
x = 10

def foo():

    x += 1
```

```
    print x

foo()
```

Why, when run, this results in the following error:

```
Traceback (most recent call last):

  File "unboundlocalerror.py", line 8, in <module>

    foo()

  File "unboundlocalerror.py", line 4, in foo

    x += 1

UnboundLocalError: local variable 'x' referenced before assignment
```

There are a few variations on this question, with the same core hiding underneath. Here's one:

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

def foo():

    lst.append(5)   # OK

    #lst += [5]     # ERROR here

foo()

print lst
```

Running the `lst.append(5)` statement successfully appends 5 to the list. However, substitute it for `lst += [5]`, and it raises `UnboundLocalError`, although at first sight it should accomplish the same.

Although this exact question is answered in Python's official FAQ (right here), I decided to write this article with the intent of giving a deeper explanation. It will start with a basic FAQ-level answer, which should satisfy one only wanting to know how to "solve the damn problem and move on". Then, I will dive deeper, looking at the formal definition of Python to understand what's going on. Finally, I'll take a look what happens behind the scenes in the implementation of CPython to cause this behavior.

The simple answer

As mentioned above, this problem is covered in the Python FAQ. For completeness, I want to explain it here as well, quoting the FAQ when necessary.

Let's take the first code snippet again:

```
x = 10

def foo():
```

```
    x += 1

    print x

foo()
```

So where does the exception come from? Quoting the FAQ:

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope.

But `x += 1` is similar to `x = x + 1`, so it should first read `x`, perform the addition and then assign back to `x`. As mentioned in the quote above, Python considers `x` a variable local to `foo`, so we have a problem – a variable is read (referenced) before it's been assigned. Python raises the `UnboundLocalError` exception in this case [1].

So what do we do about this? The solution is very simple – Python has the global statement just for this purpose:

```
x = 10

def foo():

    global x

    x += 1

    print x

foo()
```

This prints `11`, without any errors. The `global` statement tells Python that inside `foo`, `x` refers to the global variable `x`, even if it's assigned in `foo`.

Actually, there is another variation on the question, for which the answer is a bit different. Consider this code:

```
def external():

    x = 10

    def internal():

        x += 1

        print(x)

    internal()

external()
```

This kind of code may come up if you're into closures and other techniques that use Python's lexical scoping rules. The error this generates is the familiar `UnboundLocalError`. However, applying the "global fix":

```
def external():
```

```
    x = 10

    def internal():

        global x

        x += 1

        print(x)

    internal()

external()
```

Doesn't help – another error is generated: `NameError: global name 'x' is not defined`. Python is right here – after all, there's no global variable named `x`, there's only an `x` in `external`. It may be not local to `internal`, but it's not global. So what can you do in this situation? If you're using Python 3, you have the `nonlocal` keyword. Replacing `global` by `nonlocal` in the last snippet makes everything work as expected. `nonlocal` is a new statement in Python 3, and there is no equivalent in Python 2 [2].

The formal answer

Assignments in Python are used to bind names to values and to modify attributes or items of mutable objects. I could find two places in the Python (2.x) documentation where it's defined how an assignment to a local variable works.

One is section 6.2 "Assignment statements" in the Simple Statements chapter of the language reference:

Assignment of an object to a single target is recursively defined as follows. If the target is an identifier (name):

- If the name does not occur in a global statement in the current code block: the name is bound to the object in the current local namespace.
- Otherwise: the name is bound to the object in the current global namespace.

Another is section 4.1 "Naming and binding" of the Execution model chapter:

If a name is bound in a block, it is a local variable of that block.

[...]

When a name is used in a code block, it is resolved using the nearest enclosing scope. [...] If the name refers to a local variable that has not been bound, a UnboundLocalError exception is raised.

This is all clear, but still, another small doubt remains. All these rules apply to assignments of the form `var = value` which clearly bind `var` to `value`. But the code snippets we're having a problem with here have the `+=` assignment. Shouldn't that just modify the bound value, without re-binding it?

Well, no. `+=` and its cousins (`-=`, `*=`, etc.) are what Python calls "augmented assignment statements" [emphasis mine]:

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, **and assigns the result to the original target**. The target is only evaluated once.

An augmented assignment expression like `x += 1` **can be rewritten as** `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed in-place, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

With the exception of assigning to tuples and multiple targets in a single statement, **the assignment done by augmented assignment statements is handled the same way as normal assignments**. Similarly, with the exception of the possible in-place behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

So when earlier I said that `x += 1` is similar to `x = x + 1`, I wasn't telling all the truth, but it was accurate with respect to binding. Apart for possible optimization, `+=` counts exactly as `=` when binding is considered. If you think carefully about it, it's unavoidable, because some types Python works with are immutable. Consider strings, for example:

```
x = "abc"

x += "def"
```

The first line binds `x` to the value "abc". The second line doesn't modify the value "abc" to be "abcdef". Strings are immutable in Python. Rather, it creates the new value "abcdef" somewhere in memory, and re-binds `x` to it. This can be seen clearly when examining the object ID for `x` before and after the `+=`:

```
>>> x = "abc"

>>> id(x)

11173824

>>> x += "def"

>>> id(x)

32831648

>>> x

'abcdef'
```

Note that some types in Python are mutable. For example, lists can actually be modified in-place:

```
>>> y = [1, 2]
```

```
>>> id(y)
32413376
>>> y += [2, 3]
>>> id(y)
32413376
>>> y
[1, 2, 2, 3]
```

`id(y)` didn't change after `+=`, because the object `y` referenced was just modified. Still, Python re-bound `y` to the same object [3].

The "too much information" answer

This section is of interest only to those curious about the implementation internals of Python itself.

One of the stages in the compilation of Python into bytecode is building the symbol table [4]. An important goal of building the symbol table is for Python to be able to mark the scope of variables it encounters – which variables are local to functions, which are global, which are free (lexically bound) and so on.

When the symbol table code sees a variable is assigned in a function, it marks it as local. Note that it doesn't matter if the assignment was done before usage, after usage, or maybe not actually executed due to a condition in code like this:

```
x = 10
def foo():
    if something_false_at_runtime:
        x = 20
    print(x)
```

We can use the `symtable` module to examine the symbol table information gathered on some Python code during compilation:

```
import symtable
code = '''
x = 10
def foo():
    x += 1
    print(x)
```

```
'''

table = symtable.symtable(code, '<string>', 'exec')

foo_namespace = table.lookup('foo').get_namespace()

sym_x = foo_namespace.lookup('x')

print(sym_x.get_name())

print(sym_x.is_local())
```

This prints:

```
x

True
```

So we see that `x` was marked as local in `foo`. Marking variables as local turns out to be important for optimization in the bytecode, since the compiler can generate a special instruction for it that's very fast to execute. There's an excellent article here explaining this topic in depth; I'll just focus on the outcome.

The `compiler_nameop` function in `Python/compile.c` handles variable name references. To generate the correct opcode, it queries the symbol table function `PyST_GetScope`. For our `x`, this returns a bitfield with `LOCAL` in it. Having seen `LOCAL`, `compiler_nameop` generates a `LOAD_FAST`. We can see this in the disassembly of `foo`:

```
35           0 LOAD_FAST                0 (x)
             3 LOAD_CONST               1 (1)
             6 INPLACE_ADD
             7 STORE_FAST               0 (x)

36          10 LOAD_GLOBAL              0 (print)
            13 LOAD_FAST                0 (x)
            16 CALL_FUNCTION            1
            19 POP_TOP
            20 LOAD_CONST               0 (None)
            23 RETURN_VALUE
```

The first block of instructions shows what `x += 1` was compiled to. You will note that already here (before it's actually assigned), `LOAD_FAST` is used to retrieve the value of `x`.

This `LOAD_FAST` is the instruction that will cause the `UnboundLocalError` exception to be raised at runtime, because it is actually executed before any `STORE_FAST` is done for `x`. The gory details are in the bytecode interpreter code in `Python/ceval.c`:

```
TARGET(LOAD_FAST)
```

```
    x = GETLOCAL(oparg);
    if (x != NULL) {
        Py_INCREF(x);
        PUSH(x);
        FAST_DISPATCH();
    }
    format_exc_check_arg(PyExc_UnboundLocalError,
        UNBOUNDLOCAL_ERROR_MSG,
        PyTuple_GetItem(co->co_varnames, oparg));
    break;
```

Ignoring the macro-fu for the moment, what this basically says is that once `LOAD_FAST` is seen, the value of `x` is obtained from an indexed array of objects [5]. If no `STORE_FAST` was done before, this value is still `NULL`, the `if` branch is not taken [6] and the exception is raised.

You may wonder why Python waits until runtime to raise this exception, instead of detecting it in the compiler. The reason is this code:

```
x = 10
def foo():
    if something_true():
        x = 1
    x += 1
    print(x)
```

Suppose `something_true` is a function that returns `True`, possibly due to some user input. In this case, `x = 1` binds `x` locally, so the reference to it in `x += 1` is no longer unbound. This code will then run without exceptions. Of course if `something_true` actually turns out to return `False`, the exception will be raised. Python has no way to resolve this at compile time, so the error detection is postponed to runtime.

[1] This is quite useful, if you think about it. In C & C++ you can use the value of an un-initialized variable, which is almost always a bug. Some compilers (with some settings) warn you about this, but in Python it's just a plain error.

[2] If you're using Python 2 and still need such code to work, the common workaround is the following: if you have data in `external` which you want to modify in `internal`, store it inside a `dict` instead of a stand-alone variable.

[3] Could this be spared? Due to the dynamic nature of Python, that would be hard to do. At compilation time, when Python is compiled to bytecode, there's no way to know what the real type of the objects is. `y` in the example above could be some user-defined type with an overloaded `+=` operator which returns a new object, so Python compiler has to create generic code that re-binds the variable.

[4] I've written comprehensively on the internals of symbol table construction in Python's compiler (part 1 and part 2).

[5] `GETLOCAL(i)` is a macro for `(fastlocals[i])`.

[6] Had the `if` been entered, the exception raising code would not have been reached, since `FAST_DISPATCH` expands to a `goto` that takes control elsewhere.

Less copies in Python with the buffer protocol and memoryviews

November 28th, 2011 at 7:48 am

For one of the hobby projects I'm currently hacking on, I recently had to do a lot of binary data processing in memory. Large chunks of data are being read from a file, then examined and modified in memory and finally used to write some reports.

This made me think about the most efficient way to read data from a file into a modifiable memory chunk in Python. As we all know, the standard file `read` method, for a file opened in binary mode, returns a `bytes` object [1], which is immutable:

```
# Snippet #1
f = open(FILENAME, 'rb')
data = f.read()
# oops: TypeError: 'bytes' object does not support item assignment
data[0] = 97
```

This reads the whole contents of the file into `data` - a `bytes` object which is read only. But what if we now want to perform some modifications on the data? Then, we need to somehow get it into a writable object. The most straightforward writable data buffer in Python is a `bytearray`. So we can do this:

```
# Snippet #2
f = open(FILENAME, 'rb')
data = bytearray(f.read())
data[0] = 97 # OK!
```

Now, the `bytes` object returned by `f.read()` is passed into the `bytearray` constructor, which copies its contents into an internal buffer. Since `data` is a `bytearray`, we can manipulate it.

Although it appears that the goal has been achieved, I don't like this solution. The extra copy made by `bytearray` is bugging me. Why is this copy needed? `f.read()` just returns a throwaway buffer we don't need anyway - can't we just initialize the `bytearray` directly, without copying a temporary buffer?

This use case is one of the reasons the Python buffer protocol exists.

The buffer protocol - introduction

The buffer protocol is described in the Python documentation and in PEP 3118 [2]. Briefly, it provides a way for Python objects to expose their internal buffers to other objects. This is useful to avoid extra copies and for certain kinds of sharing. There are many examples of the buffer protocol in use. In the core language - in builtin types such as `bytes` and `bytearray`, in the standard library (for example `array.array` and `ctypes`) and 3rd party libraries (some important Python libraries such as `numpy` and `PIL` rely extensively on the buffer protocol for performance).

There are usually two or more parties involved in each protocol. In the case of the Python buffer protocol, the parties are a "producer" (or "provider") and a "consumer". The producer exposes its internals via the buffer protocol, and the consumer accesses those internals.

Here I want to focus specifically on one use of the buffer protocol that's relevant to this article. The producer is the built-in `bytearray` type, and the consumer is a method in the `file` object named `readinto`.

A more efficient way to read into a bytearray

Here's the way to do what Snippet #2 did, just without the extra copy:

```
# Snippet #3
f = open(FILENAME, 'rb')
data = bytearray(os.path.getsize(FILENAME))
f.readinto(data)
```

First, a `bytearray` is created and pre-allocated to the size of the data we're going to read into it. The pre-allocation is important - since `readinto` directly accesses the internal buffer of `bytearray`, it won't write more than has been allocated. Next, the `file.readinto` method is used to read the data directly into the bytearray's internal storage, without going via temporary buffers.

The result: this code runs ~30% faster than snippet #2 [3].

Variations on the theme

Other objects and modules could be used here. For example, the built-in `array.array` class also supports the buffer protocol, so it can also be written and read from a file directly and efficiently. The same goes for `numpy` arrays. On the consumer side, the `socket` module can also read directly into a buffer with the `read_into` method. I'm sure that it's easy to find many other sample uses of this protocol in Python itself and some 3rd partly libraries - if you find something interesting, please let me know.

The buffer protocol - implementation

Let's see how Snippet #3 works under the hood using the buffer protocol [4]. We'll start with the producer.

`bytearray` declares that it implements the buffer protocol by filling the `tp_as_buffer` slot of

its type object [5]. What's placed there is the address of a `PyBufferProcs` structure, which is a simple container for two function pointers:

```
typedef int (*getbufferproc)(PyObject *, Py_buffer *, int);
typedef void (*releasebufferproc)(PyObject *, Py_buffer *);
/* ... */
typedef struct {
     getbufferproc bf_getbuffer;
     releasebufferproc bf_releasebuffer;
} PyBufferProcs;
```

`bf_getbuffer` is the function used to obtain a buffer from the object providing it, and `bf_releasebuffer` is the function used to notify the object that the provided buffer is no longer needed.

The `bytearray` implementation in `Objects/bytearrayobject.c` initializes an instance of `PyBufferProces` thus:

```
static PyBufferProcs bytearray_as_buffer = {
    (getbufferproc)bytearray_getbuffer,
    (releasebufferproc)bytearray_releasebuffer,
};
```

The more interesting function here is `bytearray_getbuffer`:

```
static int
bytearray_getbuffer(PyByteArrayObject *obj, Py_buffer *view, int flags)
{
    int ret;
    void *ptr;
    if (view == NULL) {
        obj->ob_exports++;
        return 0;
    }
    ptr = (void *) PyByteArray_AS_STRING(obj);
    ret = PyBuffer_FillInfo(view, (PyObject*)obj, ptr, Py_SIZE(obj), 0, flags);
    if (ret >= 0) {
        obj->ob_exports++;
    }
    return ret;
}
```

It simply uses the `PyBuffer_FillInfo` API to fill the buffer structure passed to it. `PyBuffer_FillInfo` provides a simplified method of filling the buffer structure, which is suitable for unsophisticated objects like `bytearray` (if you want to see a more complex example that has to fill the buffer structure manually, take a look at the corresponding function of `array.array`).

On the consumer side, the code that interests us is the `buffered_readinto` function in `Modules`

`\_io\bufferedio.c` [6]. I won't show its full code here since it's quite complex, but with regards to the buffer protocol, the flow is simple:

11. Use the `PyArg_ParseTuple` function with the `w*` format specifier to parse its argument as a R/W buffer object, which itself calls `PyObject_GetBuffer` - a Python API that invokes the producer's "get buffer" function.
12. Read data from the file directly into this buffer.
13. Release the buffer using the `PyBuffer_Release` API [7], which eventually gets routed to the `bytearray_releasebuffer` function in our case.

To conclude, here's what the call sequence looks like when `f.readinto(data)` is executed in the Python code:

```
buffered_readinto
|
\--> PyArg_ParseTuple(..., "w*", ...)
|    |
|    \--> PyObject_GetBuffer(obj)
|         |
|         \--> obj->ob_type->tp_as_buffer->bf_getbuffer
|
|--> ... read the data
|
\--> PyBuffer_Release
     |
     \--> obj->ob_type->tp_as_buffer->bf_releasebuffer
```

Memory views

The buffer protocol is an internal implementation detail of Python, accessible only on the C-API level. And that's a good thing, since the buffer protocol requires certain low-level behavior such as properly releasing buffers. Memoryview objects were created to expose it to a user's Python code in a safe manner:

memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

The linked documentation page explains `memoryviews` quite well and should be immediately comprehensible if you've reached so far in this article. Therefore I'm not going to explain how a `memoryview` works, just show some examples of its use.

It is a known fact that in Python, slices on strings and bytes make copies. Sometimes when performance matters and the buffers are large, this is a big waste. Suppose you have a large buffer and you want to pass just half of it to some function (that will send it to a socket or do something else [8]). Here's what happens (annotated Python pseudo-code):

```
mybuf = ... # some large buffer of bytes
func(mybuf[:len(mybuf)//2])
```

```
  # passes the first half of mybuf into func
  # COPIES half of mybuf's data to a new buffer
```

The copy can be expensive if there's a lot of data involved. What's the alternative? Using a `memoryview`:

```
mybuf = ... # some large buffer of bytes
mv_mybuf = memoryview(mybuf) # a memoryview of mybuf
func(mv_mybuf[:len(mv_mybuf)//2])
  # passes the first half of mybuf into func as a "sub-view" created
  # by slicing a memoryview.
  # NO COPY is made here!
```

A `memoryview` behaves just like `bytes` in many useful contexts (for example, it supports the mapping protocol) so it provides an adequate replacement if used carefully. The great thing about it is that it uses the buffer protocol beneath the covers to avoid copies and just juggle pointers to data. The performance difference is dramatic - I timed a 300x speedup on slicing out a half of a 1MB `bytes` buffer when using a `memoryview` as demonstrated above. And this speedup will get larger with larger buffers, since it's O(1) vs. the O(n) of copying.

But there's more. On writable producers such as `bytearray`, a `memoryview` creates a writable view that can be modified:

```
>>> buf = bytearray(b'abcdefgh')
>>> mv = memoryview(buf)
>>> mv[4:6] = b'ZA'
>>> buf
bytearray(b'abcdZAgh')
```

This gives us a way to do something we couldn't achieve by any other means - read from a file (or receive from a socket) directly into the middle of some existing buffer [9]:

```
buf = bytearray(...) # pre-allocated to the needed size
mv = memoryview(buf)
numread = f.readinto(mv[some_offset:])
```

Conclusion

This article demonstrated the Python buffer protocol, showing both how it works and what it can be used for. The main use of the buffer protocol to the Python programmer is optimization of certain patterns of coding, by avoiding unnecessary data copies.

Any mention of optimization in Python code is sure to draw fire from people claiming that if I want to write fast code, I shouldn't use Python at all. But I disagree. Python these days is fast enough to be suitable for many tasks that were previously only in the domain of C/C++. I want to keep using it while I can, and only resort to low-level C/C++ when I must.

Employing the buffer protocol to have zero-copy buffer manipulations on the Python level is IMHO a huge boon that can stall (or even avoid) the transition of some performance-sensitive code from Python to C. That's because when dealing with data processing, we

often use a lot of C APIs anyway, the only Python overhead being the passing of data between these APIs. A speed boost in this code can make a huge difference and bring the Python code very close to the performance we could have with plain C.

The article also gave a glimpse into one aspect of the implementation of Python, hopefully showing that it's not difficult at all to dive right into the code and understand how Python does something it does.

[1] In this article I'm focusing on the latest Python 3.x, although most of it also applies to Python 2.7

[2] The buffer protocol existed in Python prior to 2.6, but was then greatly enhanced. The PEP also describes the change that was made to the buffer protocol with the move to Python 3.x (and later backported to the 2.x line starting with 2.6).

[3] This is on the latest build of Python 3.3. Roughly the same speedup can be seen on Python 2.7. With Python 3.2 there appears to be a speed regression that makes the two snippets perform similarly, but it has been fixed in 3.3

Another note on the benchmarking: it's recommended to use large files (say, ~100 MB and up) to get reliable measurements. For small files too many irrelevant factors come into play and offset the benchmarks. In addition, the code should be run in a loop to avoid differences due to warm/cold disk cache issues. I'm using the `timeit` module, which is perfect for this purpose.

[4] All the code displayed here is taken from the latest development snapshot of Python 3.3 (`default` branch).

[5] Type objects are a fascinating aspect of Python's implementation, and I hope to cover it in a separate article one day. Briefly, it allows Python objects to declare which services they provide (or, in other terms, which interfaces they implement). More information can be found in the documentation.

[6] Since the built-in `open` function, when asked to open a file in binary mode for reading, returns an `io.BufferedReader` object by default. This can be controlled with the buffering argument.

[7] Releasing the buffer structure is an important part of the buffer protocol. Each time it's requested for a buffer, `bytearray` increments a reference count, and decrements it when the buffer is released. While the refcount is positive (meaning that there are consumer objects directly relying on the internal buffer), `bytearray` won't agree to resize or do other operations that may invalidate the internal buffer. Otherwise, this would be an avenue for insidious memory bugs.

[8] Networking code is actually a common use case. When sending data over sockets, it's frequently sliced and diced to build frames. This can involve a lot of copying. Other data-munging applications such as encryption and compression are also culprits.

[9] This code snippet was borrowed from Antoine Pitrou's post on python-dev.

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:

14. The Python source code is compiled to bytecode.
15. 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.

Python objects, types, classes, and instances - a glossary

March 30th, 2012 at 7:35 am

While writing the article on the internals of Python callables, it occurred to me that some things in Python have more than one name. At the same time, some names are sometimes used to refer to more than one entity, and which one is implied has to be understood from context. Therefore, I think it's a good idea to collect this nomenclature in a single place for the sake of my future writings. This way I'll just be able to point here every time I discuss these topics, instead of explaining them over and over again.

Specifically, I want to define what I mean by types, objects, classes and instances. Note that this refers to Python 3.x, but is mostly applicable for 2.x as well [1].

Objects

It's easiest to start with objects. The Python data model reference has a pretty good definition:

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer," code is also represented by objects.)

Every object has an identity, a type and a value.

So, everything in Python is an object. Lists are objects. 42 is an object. Modules are objects. Functions are objects. Python bytecode is also kept in an object. All of these have types and unique IDs:

```
>>> def foo(): pass
...
>>> type(foo), id(foo)
(<class 'function'>, 38110760)
>>> type(foo.__code__), id(foo.__code__)
(<class 'code'>, 38111680)
```

This "everything is an object" model is backed by the CPython implementation. Indeed, if you look into the code of CPython, you'll notice that every entity mentioned above can be manipulated via a pointer to the `PyObject` base struct.

Types

The data model reference is useful here too:

[...] An object's type determines the operations that the object supports (e.g., "does it have a length?") and also defines the possible values for objects of that type.

So, every object in Python has a type. Its type can be discovered by calling the `type` builtin function [2]. The type is an object too, so it has a type of its own, which is called `type`. This last fact may not be very exciting or useful when you're just writing Python code, but it's hugely important if you want to understand the internals of CPython:

```
>>> type(42)
<class 'int'>
>>> type(type(42))
<class 'type'>
>>> type(type(type(42)))
<class 'type'>
```

Yep, it's turtles all the way down.

Classes

In the olden days, there was a difference between user-defined classes and built in types. But since 2.2, as long as you're using "new-style" classes (classes that inherit from `object` in 2.x, and are default in 3.x), there is no real difference. Essentially, a class is a mechanism Python gives us to create new user-defined types from Python code.

```
>>> class Joe: pass
...
>>> j = Joe()
>>> type(j)
<class '__main__.Joe'>
```

Using the class mechanism, we've created `Joe` - a user-defined type. `j` is an instance of the

class `Joe`. In other words, it's an object and its type is `Joe`.

As any other type, `Joe` is an object itself, and it has a type too. This type is `type`:

```
>>> type(type(j))
<class 'type'>
```

The terms "class" and "type" are an example of two names referring to the same concept. To avoid this confusion, I will always try to say "type" when I mean a type, and "user-defined class" (or "user-defined type") when referring to a new type created using the `class` construct. Note that when we create new types using the C API of CPython, there's no "class" mentioned – we create a new "type", not a new "class".

Instances

Not unlike the ambiguity between "class" and "type", "instance" is synonymous to "object". Think of it this way: objects are instances of types. So, "42 is an instance of the type `int`" is equivalent to "42 is an `int` object". I usually use "instance" and "object" interchangeably. In some cases when I want to specifically refer to objects as artifacts of the CPython implementation, I will try to use "instance" to refer to actual instances of classes. Another place where the term "instance" is explicitly used by Python is in built-ins like `isinstance` and the special `__instancecheck__` attribute.

Conclusion

As we've seen, there are two pairs of roughly synonymous terms in Python nomenclature. Types and classes are interchangeable concepts. I prefer to say "type" wherever possible, leaving the term "class" for user-defined types created with the "class" construct. IMHO "type" is a better term, and Python wouldn't be worse if the "class" concept was wiped out completely.

Similarly, objects and instances are terms that mean the same thing, but perhaps from slightly different angles. Sometimes it's more convenient to use "instance" (i.e. when specifically talking about specific objects being instances of specific types – as in "`j` is an instance of `Joe`"), and sometimes it's better to use "object" (i.e. when discussing the guts of the CPython implementation).

I sincerely hope this post is more helpful than confusing! For me, it's an aid that serves as a simple glossary when my usage of these terms in some article may be unclear or ambiguous.

[1] As long as you forget about the existence of classic 2.x classes and take it as a fact that all user-defined classes inherit from `object`.

[2] An alternative is the `__class__` attribute.

The fundamental types of Python – a diagram

April 3rd, 2012 at 8:33 pm

The aim of this post is to present a succinct diagram that correlates some basic properties of all Python objects with the fundamental types `type` and `object`. This is not a tutorial – it's more of a reference snapshot that puts things in order. To properly understand why things are the way they are, check out the existing and future writings in the Python internals category of this blog, as well as other resources available online.

In Python, every object has a type. Types are also objects – rather special objects. A type object, like any other object, has a type of its own. It also has a sequence of "base types" – in other words, types from which it inherits. This is unlike non-type objects, which don't have base types.

Consider this exemplary piece of code (Python 3):

```
# Some types
class Base:
    pass
class Klass(Base):
    pass
class Meta(type):
    pass
class KlassWithMeta(metaclass=Meta):
    pass
# Non-types
kwm = KlassWithMeta()
mylist = []
```

The following diagram describes the types and bases of all the objects created in this code. Non-type objects only have types and no bases:

Some interesting things to note:

- The default type of all new types is `type`. This can be overridden by explicitly specifying the metaclass for a type.
- Built-in types like `list` and user-defined types like `Base` are equivalent as far as Python is concerned.
- The special type `type` is the default type of all objects – including itself. It is an object, and as such, inherits from `object`.
- The special type `object` is the pinnacle of every inheritance hierarchy – it's the ultimate base type of all Python types.
- `type` and `object` are the only types in Python that really stand out from other types (and hence they are colored differently). `type` is its own type. `object` has no base type.

Python object creation sequence

April 16th, 2012 at 7:03 am

[The Python version described in this article is 3.x]

This article aims to explore the process of creating new objects in Python. As I explained in a previous article, object creation is just a special case of calling a callable. Consider this Python code:

```
class Joe:
    pass
j = Joe()
```

What happens when `j = Joe()` is executed? Python sees it as a call to the callable `Joe`, and routes it to the internal function `PyObject_Call`, with `Joe` passed as the first argument. `PyObject_Call` looks at the type of its first argument to extract its `tp_call` attribute.

Now, what is the type of `Joe`? Whenever we define a new Python class, unless we explicitly specify a metaclass for it, its type is `type`. Therefore, when `PyObject_Call` attempts to look at the type of `Joe`, it finds `type` and picks its `tp_call` attribute. In other words, the function `type_call` in `Objects/typeobject.c` is invoked [1].

This is an interesting function, and it's short, so I'll paste it wholly here:

```
static PyObject *
type_call(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    PyObject *obj;
    if (type->tp_new == NULL) {
        PyErr_Format(PyExc_TypeError,
                     "cannot create '%.100s' instances",
                     type->tp_name);
        return NULL;
    }
    obj = type->tp_new(type, args, kwds);
    if (obj != NULL) {
        /* Ugly exception: when the call was type(something),
           don't call tp_init on the result. */
        if (type == &PyType_Type &&
            PyTuple_Check(args) && PyTuple_GET_SIZE(args) == 1 &&
            (kwds == NULL ||
             (PyDict_Check(kwds) && PyDict_Size(kwds) == 0)))
            return obj;
        /* If the returned object is not an instance of type,
           it won't be initialized. */
        if (!PyType_IsSubtype(Py_TYPE(obj), type))
            return obj;
        type = Py_TYPE(obj);
        if (type->tp_init != NULL &&
            type->tp_init(obj, args, kwds) < 0) {
```

```
            Py_DECREF(obj);
            obj = NULL;
        }
    }
    return obj;
}
```

So what arguments is `type_call` being passed in our case? The first one is `Joe` itself - but how is it represented? Well, `Joe` is a class, so it's a type (all classes are types in Python 3). Types are represented inside the CPython VM by `PyTypeObject` objects [2].

What `type_call` does is first call the `tp_new` attribute of the given type. Then, it checks for a special case we can ignore for simplicity, makes sure `tp_new` returned an object of the expected type, and then calls `tp_init`. If an object of a different type was returned, it is not being initialized.

Translated to Python, what happens is this: if your class defines the `__new__` special method, it gets called first when a new instance of the class is created. This method has to return some object. Usually, this will be of the required type, but this doesn't have to be the case. Objects of the required type get `__init__` invoked on them. Here's an example:

```
class Joe:
    def __new__(cls, *args, **kwargs):
        obj = super(Joe, cls).__new__(cls)
        print('__new__ called. got new obj id=0x%x' % id(obj))
        return obj
    def __init__(self, arg):
        print('__init__ called (self=0x%x) with arg=%s' % (id(self), arg))
        self.arg = arg
j = Joe(12)
print(type(j))
```

This prints:

```
__new__ called. got new obj id=0x7f88e7218290
__init__ called (self=0x7f88e7218290) with arg=12
<class '__main__.Joe'>
```

Customizing the sequence

As we saw above, since the type of `Joe` is `type`, the `type_call` function is invoked to define the creation sequence for `Joe` instances. This sequence can be changed by specifying a custom type for `Joe` - in other words, a metaclass. Let's modify the previous example to specify a custom metaclass for `Joe`:

```
class MetaJoe(type):
    def __call__(cls, *args, **kwargs):
        print('MetaJoe.__call__')
        return None
```

```
class Joe(metaclass=MetaJoe):
    def __new__(cls, *args, **kwargs):
        obj = super(Joe, cls).__new__(cls)
        print('__new__ called. got new obj id=0x%x' % id(obj))
        return obj
    def __init__(self, arg):
        print('__init__ called (self=0x%x) with arg=%s' % (id(self), arg))
        self.arg = arg
j = Joe(12)
print(type(j))
```

So now the type of `Joe` is not `type`, but `MetaJoe`. Consequently, when `PyObject_Call` picks the call function to execute for `j = Joe(12)`, it takes `MetaJoe.__call__`. The latter prints a notice about itself and returns `None`, so we don't expect the `__new__` and `__init__` methods of `Joe` to be called at all. Indeed, this is the outcome:

```
MetaJoe.__call__
<class 'NoneType'>
```

Digging deeper – tp_new

Alright, so now we have a better understanding of the object creation sequence. One crucial piece of the puzzle is still missing, though. While we almost always define `__init__` for our classes, defining `__new__` is rather rare [3]. Moreover, from a quick look at the code it's obvious that `__new__` is more fundamental in a way. This method is used to create a new object. It is called once and only once per instantiation. `__init__`, on the other hand, already gets a constructed object and may not be called at all; it can also be called multiple times.

Since the `type` parameter passed to `type_call` in our case is `Joe`, and `Joe` does not define a custom `__new__` method, then `type->tp_new` defers to the `tp_new` slot of the base type. The base type of `Joe` (and all other Python objects, except `object` itself) is `object`. The `object.tp_new` slot is implemented in CPython by the `object_new` function in `Objects/typeobject.c`.

`object_new` is actually very simple. It does some argument checking, verifies that the type we're trying to instantiate is not abstract, and then does this:

```
return type->tp_alloc(type, 0);
```

`tp_alloc` is a low-level slot of the type object in CPython. It's not directly accessible from Python code, but should be familiar to C extension developers. A custom type defined in a C extension may override this slot to supply a custom memory allocation scheme for instances of itself. Most C extension types will, however, defer this allocation to the function `PyType_GenericAlloc`.

This function is part of the public C API of CPython, and it also happens to be assigned to the `tp_alloc` slot of `object` (defined in `Objects/typeobject.c`). It figures out how much memory the new object needs [4], allocates a memory chunk from CPython's memory allocator and initializes it all to zeros. It then initializes the bare essential `PyObject` fields

(type and reference count), does some GC bookkeeping and returns. The result is a freshly allocated instance.

Conclusion

Lest we lose the forest for the trees, let's revisit the question this article began with. What happens when CPython executes `j = Joe()`?

- Since `Joe` has no explicit metaclass, `type` is its type. So the `tp_call` slot of `type`, which is `type_call`, is called.
- `type_call` starts by calling the `tp_new` slot of `Joe`:
    - Since `Joe` has no explicit base clase, its base is `object`. Therefore, `object_new` is called.
    - Since `Joe` is a Python-defined class, it has no custom `tp_alloc` slot. Therefore, `object_new` calls `PyType_GenericAlloc`.
    - `PyType_GenericAlloc` allocates and initializes a chunk of memory big enough to contain `Joe`.
- `type_call` then goes on and calls `Joe.__init__` on the newly created object.
    - Since `Joe` does not define `__init__`, its base's `__init__` is called, which is `object_init`.
    - `object_init` does nothing.
- The new object is returned from `type_call` and is bound to the name `j`.

This is the vanilla flow for an object of a class that doesn't have a custom metaclass, doesn't have an explicit base class, and doesn't define its own `__new__` and `__init__` methods. However, this article should have made it quite clear where these custom capabilities plug in to modify the object creation sequence. As you can see, Python is amazingly flexible. Practically every single step of the process described above can be customized, even for user-defined types implemented in Python. Types implemented in a C extension can customize even more, such as the exact memory allocation strategy used to create instances of the type.

[1] The `PyTypeObject` structure definition for `type` is `PyType_Type` in `Objects/typeobject.c`. You can see that `type_call` is being assigned to its `tp_call` slot.
[2] A future article will show how this comes to be when a new class is created.
[3] Even when we do explicitly override `__new__` in our classes, we almost certainly defer the actual object creation to the base's `__new__`.
[4] This information is available in the `PyObject` header of any type.

Under the hood of Python class definitions

June 15th, 2012 at 5:51 am

This is a fast-paced walk-through of the internals of defining new classes in Python. It shows what actually happens inside the Python interpreter when a new class definition is encountered and processed. Beware, this is advanced material. If the prospect of pondering the metaclass of the metaclass of your class makes you feel nauseated, you better stop now.

The focus is on the official (CPython) implementation of Python 3. For modern releases of Python 2 the concepts are similar, although there will be some slight differences in the details.

On the bytecode level

I'll start right with the bytecode, ignoring all the good work done by the Python compiler [1]. For simplicity, this function will be used to demonstrate the bytecode generated by a class definition, since it's easy to disassemble functions:

```
def myfunc():
    class Joe:
        attr = 100.02
        def foo(self):
            return 2
```

Disassembling `myfunc` will show us the steps needed to define a new class:

```
>>> dis.disassemble(myfunc.__code__)
 14           0 LOAD_BUILD_CLASS
              1 LOAD_CONST               1 (<code object Joe at 0x7fe226335b80, file
"disassemble.py", line 14>)
              4 LOAD_CONST               2 ('Joe')
              7 MAKE_FUNCTION            0
             10 LOAD_CONST               2 ('Joe')
             13 CALL_FUNCTION            2
             16 STORE_FAST               0 (Joe)
             19 LOAD_CONST               0 (None)
             22 RETURN_VALUE
```

The number immediately preceding the instruction name is its offset in the binary representation of the code object. All the instructions until and including the one at offset 16 are for defining the class. The last two instructions are for `myfunc` to return `None`.

Let's go through them, step by step. Documentation of the Python bytecode instructions is available in the dis module.

`LOAD_BUILD_CLASS` is a special instruction used for creating classes. It pushes the function `builtins.__build_class__` onto the stack. We'll examine this function in much detail later.

Next, a code object, followed by a name (`Joe`) are pushed onto the stack as well. The code object is interesting, let's peek inside:

```
>>> dis.disassemble(myfunc.__code__.co_consts[1])
 14           0 LOAD_FAST                0 (__locals__)
              3 STORE_LOCALS
              4 LOAD_NAME                0 (__name__)
              7 STORE_NAME               1 (__module__)
             10 LOAD_CONST               0 ('myfunc.<locals>.Joe')
             13 STORE_NAME               2 (__qualname__)
 15          16 LOAD_CONST               1 (100.02)
             19 STORE_NAME               3 (attr)
 16          22 LOAD_CONST               2 (<code object foo at 0x7fe226335c40, file
"disassemble.py", line 16>)
             25 LOAD_CONST               3 ('myfunc.<locals>.Joe.foo')
             28 MAKE_FUNCTION            0
             31 STORE_NAME               4 (foo)
             34 LOAD_CONST               4 (None)
             37 RETURN_VALUE
```

This code defines the innards of the class. Some generic bookkeeping, followed by definitions for the `attr` attribute and `foo` method.

Now let's get back to the first disassembly. The next instruction (at offset 7) is `MAKE_FUNCTION` [2]. This instruction pulls two things from the stack – a name and a code object. So in our case, it gets the name `Joe` and the code object we saw disassembled above. It creates a function with the given name and the code object as its code and pushes it back to the stack.

This is followed by once again pushing the name `Joe` onto the stack. Here's what the stack looks like now (TOS means "top of stack"):

```
TOS> name "Joe"
     function "Joe" with code for defining the class
     function builtins.__build_class__
     ----------------------------------------------
```

At this point (offset 13), `CALL_FUNCTION 2` is executed. The 2 simply means that the function was passed two positional arguments (and no keyword arguments). `CALL_FUNCTION` first takes the arguments from the stack (the rightmost on top), and then the function itself. So the call is equivalent to:

```
builtins.__build_class__(function defining "Joe", "Joe")
```

Build me a class, please

A quick peek into the `builtins` module in `Python/bltinmodule.c` reveals that `__build_class__` is implemented by the function `builtin___build_class__` (I'll call it BBC for simplicity) in the same file.

As any Python function, BBC accepts both positional and keyword arguments. The positional arguments are:

```
func, name, base1, base2, ... baseN
```

So we see only the function and name were passed for `Joe`, since it has no base classes. The only keyword argument BBC understands is `metaclass` [3], allowing the Python 3 way of defining metaclasses:

```
class SomeOtherJoe(metaclass=JoeMeta):
  [...]
```

So back to BBC, here's what it does [4]:

16. The first chunk of code deals with extracting the arguments and setting defaults.

17. Next, if no metaclass is supplied, BBC looks at the base classes and takes the metaclass of the first base class. If there are no base classes, the default metaclass `type` is used.

18. If the metaclass is really a class (note that in Python any callable can be given as a metaclass), look at the bases again to determine "the most derived" metaclass.

The last point deserves a bit of elaboration. If our class has bases, then some rules apply for the metaclasses that are allowed. The metaclasses of its bases must be either subclasses or superclasses of our class's metaclass. Any other arrangement will result in this `TypeError`:

```
metaclass conflict: the metaclass of a derived class must be a (non-strict)
                    subclass of the metaclasses of all its bases
```

Eventually, given that there are no conflicts, the most derived metaclass will be chosen. The most derived metaclass is the one which is a subtype of the explicitly specified metaclass and the metaclasses of all the base classes. In other words, if our class's metaclass is `Meta1`, only one of the bases has a metaclass and that's `Meta2`, and `Meta2` is a subclass of `Meta1`, it is `Meta2` that will be picked to serve as the eventual metaclass of our class.

5. At this point BBC has a metaclass [5], so it starts by calling its `__prepare__` method to create a namespace dictionary for the class. If there's no such method, an empty dict is used.

As documented in the data model reference:

If the metaclass has a __prepare__() attribute (usually implemented as a class or static method), it is called before the class body is evaluated with the name of the class and a tuple of its bases for arguments. It should return an object that supports the mapping interface that will be used to store the namespace of the class. The default is a plain dictionary. This could be used, for example, to keep track of the order that class attributes are declared in by returning an ordered dictionary.

5. The function argument is invoked, passing the namespace dict as the only argument. If we look back at the disassembly of this function (the second one), we see that the first argument is placed into the `f_locals` attribute of the frame (with the `STORE_LOCALS` instruction). In other words, this dictionary is then used to populate the class

attributes. The function itself returns `None` - its outcome is modifying the namespace dictionary.

6. Finally, the metaclass is called with the name, list of bases and namespace dictionary as arguments.

The last step defers to the metaclass to actually create a new class with the given definition. Recall that when some class `MyKlass` has a metaclass `MyMeta`, then the class definition of `MyKlass` is equivalent to [6]:

```
MyKlass = MyMeta(name, bases, namespace_dict)
```

The flow of BBC outlined above directly embodies this equivalence.

So what happens next? Well, the metaclass `MyMeta` is a class, right? And what happens when a class is "called"? It's instantiated. How is a class's instantiation done? By invoking its metaclass's `__call__`. So wait, this is the metaclass's metaclass we're talking about here, right? Yes! A metaclass is just a class, after all [7], and has a metaclass of its own - so Python has to keep the meta-flow going.

Realistically, what probably happens is this:

Most chances are that your class has no metaclass specified explicitly. Then, its default metaclass is `type`, so the call above is actually:

```
MyKlass = type(name, bases, namespace_dict)
```

The metaclass of `type` happens to be `type` itself, so here `type.__call__` is called.

In the more complex case that your class does have a metaclass, most chances are that the metaclass itself has no metaclass [8], so `type` is used for it. Therefore, the `MyMeta(...)` call is also served by `type.__call__`.

type_call

In `Objects/typeobject.c`, the `type.__call__` slot is getting mapped to the function `type_call`. I've already spent some time explaining how it works, so it's important to review that article at this point.

Things are a bit different here, however. The object creation sequence article explained how instances are created, so the `tp_new` slot called from `type_call` went to `object`. Here, since `type_call` will actually call `tp_new` on a metaclass, and the metaclass's base is `type` (see this diagram), we'll have to study how the `type_new` function (also from `Objects/typeobject.c`) works.

A brief recap

I feel that the flow here is relatively convoluted, so lest we lose focus, let's have a brief recap of how we got thus far. The following is a much simplified version of the flow described so far in this article:

1. When a new class `Joe` is defined...
2. The Python interpreter arranges the builtin function `builtin__build_class__` (BBC) to be called, giving it the class name and its innards compiled into a code object.
3. BBC finds the metaclass of `Joe` and calls it to create the new class.
4. When any class in Python is called, it means that its metaclass's `tp_call` slot is invoked. So to create `Joe`, this is the `tp_call` of its metaclass's metaclass. In most cases this is the `type_call` function (since the metaclass's metaclass is almost always `type`, or something that eventually delegates to it).
5. `type_call` creates a new instance of the type it's bound to by calling its `tp_new` slot.
6. In our case, that is served by the `type_new` function.

The next section picks up from step 6.

type_new

The `type_new` function is a complex beast - it's over 400 lines long. There's a good reason for this, however, since it plays a very fundamental role in the Python object system. It's literally responsible for creating all Python types. I'll go over its functionality in major blocks, pasting short snippets of code where relevant.

Let's start at the beginning. The signature of `type_new` is:

```
static PyObject *
type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
```

When called to create our class `Joe`, the arguments will be:

- `metatype` - the metaclass, so it's `type` itself.
- `args` - we saw in the description of BBC above that this is the class name, list of base classes and a namespace dict.
- `kwds` - since `Joe` has no metaclass, this will be empty.

At this point, it may be useful to recall that:

```
class Joe:
  ... contents
```

Is equivalent to:

```
Joe = type('joe', (), dict of contents)
```

`type_new` serves both approaches, of course.

It starts by handling the special 1-argument call of the `type` function, which returns the

type. Then, it tries to see if the requested type has a metaclass that's more suitable than the one passed in. This is necessary to handle a direct call to `type` as shown above - if one of the bases has a metaclass, that metaclass should be used for the creation [9].

Next, `type_new` handles some special class methods (for example `__slots__`).

Finally, the type object itself is allocated and initialized. Since the unification of types and classes in Python, user-defined classes are represented similarly to built-in types inside the CPython VM. However, there's still a difference. Unlike built-in types (and new types exported by C extension) which are statically allocated and are essentially "singletons", user-defined classes have to be implemented by dynamically allocated type objects on the heap [10]. For this purpose, `Include/object.h` defines an "extended type object", `PyHeapTypeObject`. This struct starts with a `PyTypeObject` member, so it can be passed around to Python C code expecting any normal type. The extra information it carries is used mainly for book-keeping in the type-handling code (`Objects/typeobject.c`). `PyHeapTypeObject` is an interesting type to discuss but would deserve an article of its own, so I'll stop right here.

Just as an example of one of the special cases handled by `type_new` for members of new classes, let's look at `__new__`. The data model reference says about it:

Called to create a new instance of class cls. __new__() is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument.

It's interesting to see how this statement is embodied in the code of `type_new`:

```
/* Special-case __new__: if it's a plain function,
   make it a static function */
tmp = _PyDict_GetItemId(dict, &PyId___new__);
if (tmp != NULL && PyFunction_Check(tmp)) {
    tmp = PyStaticMethod_New(tmp);
    if (tmp == NULL)
        goto error;
    if (_PyDict_SetItemId(dict, &PyId___new__, tmp) < 0)
        goto error;
    Py_DECREF(tmp);
}
```

So when the dict of the new class has a `__new__` method, it's automatically replaced with a corresponding static method.

After some more handling of special cases, `type_new` returns the object representing the newly created type.

Conclusion

This has been a relatively dense article. If you got lost, don't despair. The important part to remember is the flow described in "A brief recap" - the rest of the article just explains the items in that list in more detail.

The Python type system is very powerful, dynamic and flexible. Since this all has to be implemented in the low-level and type-rigid C, and at the same time be relatively efficient, the implementation is almost inevitably complex. If you're just writing Python code, you almost definitely don't have to be aware of all these details. However, if you're writing non-trivial C extensions, and/or hacking on CPython itself, understanding the contents of this article (at least on an approximate level) can be useful and educational.

Many thanks to Nick Coghlan for reviewing this article.

[1] If you're interested in the compilation part, this article provides a good overview.

[2] In the distant past, `MAKE_FUNCTION` was used both for creating functions and classes. However, when lexical scoping was added to Python, a new instruction for creating functions was added - `MAKE_CLOSURE`. So nowadays, as strange as it sounds, `MAKE_FUNCTION` is only used for creating classes, not functions.

[3] The other keyword arguments, if they exist, are passed to the metaclass when it's getting called.

[4] You may find it educational to open the file `Python/bltinmodule.c` from the Python source distribution and follow along.

[5] There always is some metaclass, because all classes eventually derive from `object` whose metaclass is `type`.

[6] With the caveat that BBC also calls `__prepare__`. For a more equivalent sequence, take a look at types.new_class.

[7] As I mentioned earlier, any callable can be specified as a metaclass. If the callable is a function and not a class, it's simply called as the last step of BBC - the rest of the discussion doesn't apply.

[8] I've never encountered real-world Python code where a metaclass has a metaclass of its own. If you have, please let me know - I'm genuinely curious about the use cases for such a construct.

[9] If you've noticed that this is a duplication of effort, you're right. BBC also computes the metaclass, but to handle the `type(...)` call, `type_new` has to do this again. I think that creating new classes is a rare enough occurrence that the extra work done here doesn't count for much.

[10] Since they have to be garbage collected and fully deleted when no longer needed.

Faster XML iteration with ElementTree

June 17th, 2012 at 5:28 am

As I've mentioned previously, starting with Python 3.3 the C accelerator of the `xml.etree.ElementTree` module is going to be imported by default. This should make quite a bit of code faster for those who were not aware of the existence of the accelerator, and reduce the amount of boilerplate importing for everyone.

As Python 3.3 is nearing its first beta, more work was done in the past few weeks; mostly fixing all kinds of problems that arose from the aforementioned transition. But in this post I want to focus on one feature that was added this weekend - much faster iteration over the

parsed XML tree.

`ElementTree` offers a few tools for iterating over the tree and for finding interesting elements in it, but the basis for them all is the `iter` method:

Creates a tree iterator with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If tag is not None or '*', only elements whose tag equals tag are returned from the iterator.

And until very recently, this `iter` was implemented in Python, even when the C accelerator was loaded. This was achieved by calling `PyRun_String` on a "bootstrap" string defining the method (as well as a bunch of other Python code), when the C extension module was being initialized. In the past few months I've been slowly and surely decimating this bootstrap code, trying to move as much functionality as possible into the C code and replacing stuff with actual C API calls. The last bastion was `iter` (and its cousin `itertext`) because its implementation in C is not trivial.

Well, that last bastion has now fallen and the C accelerator of `ElementTree` no longer has any Python bootstrap code - `iter` is actually implemented in C. And the great "side effect" of this is that the `iter` method (and all the other methods that rely on it, like `find`, `iterfind` and others) is now much faster. On a relatively large XML document I timed a **10x speed boost** for simple iteration looking for a specific tag. I hope that this will make a lot of XML processing code in Python much faster out-of-the-box.

This change is already in Python trunk and will be part of the 3.3 release. I must admit that I didn't spend much time optimizing the C code implementing `iter`, so there may still be an area for improvement. I have a hunch that it can be made a few 10s of percents faster with a bit of effort. If you're interested to help, drop me a line and I will be happy to discuss it.