Senthil Kumaran, Python Design

This is page describing various aspects of CPython Design. It is derived from Documents written by various authors who have researched on this topic, CPython Code and the PEPs. If you find anything lacking, please feel free to suggest the changes or corrections.

Python Internals

This is a single page, python design notes from Yaniv Aknin's Python-Innards Docs and other resources. I collected and wrote them so that I can understand and grasp the design principles. The material is possibly augumented with other resources.

Overview

Explains the Python's byte-code evaluation. For e.g, what happen's when you do

python -c "print('hello,world')"

Python's binary is executed, the standard C library initialization happens and then the main function starts executing from `./Modules/python.c: main`, which soon calls `./Modules/main.c: Py_Main` and after some initialization stuff like parse arguments, see if environment variables should affect behaviour, assess the situation of the standard streams and act accordingly, etc, `./Python/pythonrun.c: Py_Initialize` is called.

In many ways, `Py_Initialize` function is what 'builds' and assembles together the pieces needed to run the CPython machine and makes 'a process' into 'a process with a Python interpreter in it'. Among other things, it creates two very important Python data-structures: the interpreter state and thread state. It also creates the built-in module sys and the module which hosts all builtins.

It will execute a single string, since we invoked it with -c. To execute this single string, `./Python/pythonrun.c: PyRun_SimpleStringFlags` is called. This function creates the `__main__` namespace 'where' our string will be executed. After the namespace is created, the string is executed in it. To do that, you must first transform the string into something that machine can work on.

The parser/compiler stage of `PyRun_SimpleStringFlags` goes largely like this: tokenize and create a Concrete Syntax Tree (CST) from the code, transform the CST into an Abstract Syntax Tree (AST) and finally compile the AST into a code object using `./Python/ast.c: PyAST_FromNode`. The code object as a binary string of machine code that Python VM's 'machinary' can operate on - so now we're ready to do interpretation (again, evaluation in Python's parlance).

We have an empty `__main__`, we have a code object, we want to evaluate it. Now what? Now this line: `Python/pythonrun.c: run_mod, v = PyEval_EvalCode(co, globals, locals);` does the trick. It receives a code object and a namespace for globals and for locals (in this case, both of them will be the newly created `__main__` namespace), creates a frame object from these and executes it.

We know that `Py_Initialize` creates a thread state. Get back to that, each Python thread is represented by its own thread state, which among other things points to the stack of currently executing frames. After the frame object is created and placed at the top of the thread state stack, the byte code pointed by it is evaluated, opcode by opcode, by means of the `./Python/ceval.c: PyEval_EvalFrameEx`. `PyEval_EvalFrameEx` function takes the frame, extracts opcode after opcode, and corresponding operands, if any and executes a short piece of C code matching the opcode.

Opcode looks like this.:

```
>>> from dis import dis
>>> co = compile("spam = eggs - 1", "<string>", "exec")
>>> dis(co)
  1           0 LOAD_NAME                0 (eggs)
              3 LOAD_CONST               0 (1)
              6 BINARY_SUBTRACT
              7 STORE_NAME               1 (spam)
             10 LOAD_CONST               1 (None)
             13 RETURN_VALUE
>>>
```

You “load” the name eggs (where do you load it from? where do you load it to?), and also load a constant value (1), then you do a `“binary subtract”` (what do you mean ‘binary’ in this context? between which operands?), and so on and so forth. The names are “loaded” from the globals and locals namespaces we’ve seen earlier, and they’re loaded onto an operand stack (not to be confused with the stack of running frames), which is exactly where the binary subtract will pop them from, subtract one from the other, and put the result back on that stack.

Look at `PyEval_EvalFrameEx at ./Python/ceval.c`. The following piece of code is run when BINARY_SUBTRACT opcode is found.:

```
TARGET(BINARY_SUBTRACT)
    w = POP();
    v = TOP();
    x = PyNumber_Subtract(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
```

After the frame is executed and `PyRun_SimpleStringFlags` returns, the main function does some cleanup (notably, `Py_Finalize`), the standard C library deinitialization stuff is done (`atexit`), and the process exits.

Python Objects

Objects are fundamental to the innards of python and Objects are not very tightly coupled with anything else in Python. Look at the implementation of objects as if they’re unrelated to the ‘rest’, as if they’re a general purpose C API for creating an object subsystem. Objects are just a bunch of structures and some functions to manipulate them.

Mostly everything in Python is an object, from integer to dictionaries, from user defined classes to built-in ones, from stack frames to code objects.

Given a pointer to a piece of memory, the very least you must expect of it to treat it as an object are just a couple of fields defined in a C structure called `./Objects/object.h: PyObject.`:

```
typedef struct _object {
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

Many objects extend this structure to accommodate other variables required to represent the object’s value, but these two fields must always exist: a reference count and type (in special debug builds, a couple other esoteric fields are added to track references).

The reference count is an integer which counts how many times the object is referenced. `>>> a = b = c = object()` instantiates an empty object and binds it to three different names: a, b and c. Each of these names creates another reference to it even though the object is allocated only once. Binding the object to yet another name or adding the object to a list will create another reference – but will not create another object!

There is much more to say about reference counting, but that’s less central to the overall object system and more related to Garbage Collection.

We can now better understand the `./Objects/object.h: Py_DECREF` macro we’ve seen used in the introduction and didn’t know how to explain: It simply decrements `ob_refcnt` (and initiates deallocation, if `ob_refcnt` hit zero). That’s all we’ll say about reference counting for now.

`ob_type`, a pointer to an object’s type, a central piece of Python’s object model. Every object has exactly one type, which never changes during the lifetime of the object. Most importantly, the type of an object (and only the type of an object) determines what can be done with an object.

When the interpreter evaluates the subtraction opcode, a single C function `(PyNumber_Subtract)` will be called regardless of whether its operands are an integer and an integer, an integer and a float or even something nonsensical (subtract an exception from a dictionary).:

```
# n2w: the type, not the instance, determines what can be done with an instance
>>> class Foo(object):
...     "I don't have __call__, so I can't be called"
...
>>> class Bar(object):
...     __call__ = lambda *a, **kw: 42
...
>>> foo = Foo()
>>> bar = Bar()
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Foo' object is not callable
>>> bar()
42
# will adding __call__ to foo help?
>>> foo.__call__ = lambda *a, **kw: 42
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Foo' object is not callable
# how about adding it to Foo?
>>> Foo.__call__ = lambda *a, **kw: 42
>>> foo()
42
>>>
```

How can a single C function be used to handle any kind of object that is thrown at it? It can receive a `void * pointer` (actually it receives a `PyObject *` pointer, which is also opaque insofar as the object's data is concerned), but how will it know how to manipulate the object it is given? In the object's type lies the answer. A type is in itself a Python object (it also has a reference count and a type of its own, the type of almost all types is type), but in addition to the refcount and the type of the type, there are many more fields in the C structure describing type objects.

`./Include/object.h: PyTypeObject` has the information about types as well as type's structure's definition. Many of the fields a type object has are called slots and they point to functions (or to structures that point to a bunch of related functions). These functions are what will actually be called when Python C-API functions are invoked to operate on an object instantiated from that type. So while you think you're calling `PyNumber_Subtract` on both a, say, `int and a float`, in reality what happens is that the types of it operands are `dereferenced` and the type-specific subtraction function in the 'subtraction' slot is used. So we see that the C-API functions aren't generic, but rather rely on types to abstract the details away and appear as if they can work on anything (valid work is also just to raise a TypeError).

`PyNumber_Subtract` calls a generic two-argument function called `./Object/abstract.c: binary_op`, and tells it to operate on the number-like `slot nb_subtract` (similar slots exists for other functionality, like, say, the number-like slot `nb_negative` or the sequence-like slot `sq_length`). `binary_op` is an error-checking wrapper around `binary_op1`, the real 'do work' function. `./Objects/abstract.c: binary_op1` receives `BINARY_SUBTRACT's` operands as v and w, and then tries to dereference `v->ob_type->tp_as_number`, a structure pointing to many numeric slots which represents how v can be used as a number. `binary_op1` will expect to find at `tp_as_number->nb_subtract` a C function that will either do the subtraction or return the special value `Py_NotImplemented`, to signal that these operands are 'insubtracticable' in relation to one another (this will cause a TypeError exception to be raised).

If you want to change how objects behave, you can write an extension in C which will statically define its own `PyObjectType` structure in code and fill the slots away as you see fit. But when we create our own types in Python (class and type are the same thing), we don't manually allocate a C structure and we don't fill up its slots.

How come these types behave just like built-in types? The answer is inheritance, where typing plays a significant role. See, Python arrives with some built-in types, like `list or dict`. As we said, these types have a certain set of functions populating their slots and thus objects instantiated from them behave in a certain way, like a mutable sequence of values or like a mapping of keys to values. When you define a new type in Python, a new C structure for that type is dynamically allocated on the `heap` (like any other object) and its slots are filled from whichever type it is inheriting, which is also called its base

Since the slots are copied over, the newly created sub-type has mostly identical functionality to its base. Python also arrives with a featureless base object type called object (`PyBaseObject_Type` in C), which has mostly null slots and which you can extend without inheriting any particular functionality. You never really 'create' a type in pure Python, you always inherit one (if you define a class without inheriting anything explicitly, you will implicitly inherit object; in Python 2.x, not inheriting anything explicitly leads to the creation of a so called 'classic class', which is out of our scope).

Of course, you don't have to inherit everything. You can, obviously, mutate the behaviour of a type created in pure Python, as I've demonstrated in the code snippet earlier in this post. By setting the special method __`call`__ on our class Bar, I made instances of that class callable. Someone, sometime during the creation of our class, noticed this __`call`__ method exists and wired it into our newly created type's `tp_call` slot. `./Objects/typeobject.c: type_new`, an elaborate and central function, is that function.

Let's look at a small line right at the end after the new type has been fully created and just before returning `fixup_slot_dispatchers(type);`. This function iterates over the correctly named methods defined for the newly created type and wires them to the correct slots in the type's structure, based on their particular name.

Another thing remains unanswered in the sea of small details: we've demonstrated already that setting the method __`call`__ on a type after it's created will also make objects instantiated from that type callable (even objects already instantiated from that type). Recall that a type is an object, and that the type of a type is type (if your head is spinning, try: >>> `class Foo(list): pass ; type(Foo)`).

So when we do stuff to a class, like calling a class, or subtracting a class, or, indeed, setting an attribute on a class, what happens is that the `class' object's ob_type` member is dereferenced, finding that the class' type is type. Then the `type->tp_setattro` slot is used to do the actual attribute setting. So a class, like an integer or a list can have its own attribute-setting function. And the type-specific attribute-setting function (`./Objects/typeobject.c:`

`type_setattro`) calls the very same function that `fixup_slot_dispatchers` uses to actually do the fixup work (update_one_slot) after it has set a new attribute on a class.

What is happening here?:

```
>>> a = object()
>>> class C(object): pass
...
>>> b = C()
>>> a.foo = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'foo'
>>> b.foo = 5
>>>
```

How I can set an arbitrary attribute to b, which is an instance of C, which is a class inheriting object and not changing anything, and yet I can't do the same with a, an instance of that very same object? Some wise crackers can say: b has a `__dict__` and a doesn't, and that's true, but how did this new (and totally non-trivial!) functionality come from if I didn't inherit it?!

Attributes of an object

An object's attributes are other objects related to it and accessible by invoking the . (dot) operator, like so: `>>> my_object.attribute_name`. A type can define one (or more) specially named methods that will customize attribute access to its instances and they will be wired into the type's slots using `fixup_slot_dispatchers` when the type is created.

These methods simply store the attribute as a key/value pair (attribute name/attribute value) in some object-specific dictionary when an attribute is set and retrieve the attribute from that dictionary when an attribute is get (or raise an AttributeError if the dictionary doesn't have a key matching the requested attribute's name).

Here is an example snippet which presents a particularly surprising behavior of attribute access.:

```
>>> print(object.__dict__)
{'__ne__': <slot wrapper '__ne__' of 'object' objects>, ... ,
'__ge__': <slot wrapper '__ge__' of 'object' objects>}
>>> object.__ne__ is object.__dict__['__ne__']
True
>>> o = object()
>>> o.__class__
<class 'object'>
>>> o.a = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'a'
>>> o.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute '__dict__'
>>> class C:
...     A = 1
...
>>> C.__dict__['A']
1
>>> C.A
1
>>> o2 = C()
>>> o2.a = 1
>>> o2.__dict__
{'a': 1}
>>> o2.__dict__['a2'] = 2
>>> o2.a2
2
>>> C.__dict__['A2'] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_proxy' object does not support item assignment
>>> C.A2 = 2
>>> C.__dict__['A2'] is C.A2
True
>>> type(C.__dict__) is type(o2.__dict__)
False
>>> type(C.__dict__)
<class 'dict_proxy'>
>>> type(o2.__dict__)
<class 'dict'>
>>>
```

We can see that object (as in, the most basic built-in type which we've discussed before) has a private dictionary, and we see that stuff we access on object as an attribute is identical to what we find in `object.__dict__`. Instances of object (o, in the example) don't support arbitrary attribute assignment and don't have a __dict__ at all, though they do support some attribute access (try `o.__class__`, `o.__hash__`, etc; these do return things).

After that we created our own class, C, derived from object and adding an attribute A, and saw that A was accessible via `C.A` and `C.__dict__['A']` just the same, as expected.

We then instantiated o2 from C, and demonstrated that as expected, attribute assignment on it indeed mutates its __dict__ and vice versa (i.e., mutations to its __dict__ are exposed as attributes). We were then probably more surprised to learn that even though attribute assignment on the class (C.A2) worked fine, our class' __dict__ is actually read-only. Finally, we saw that our class __dict__ is not of the same type as our object's __dict__, but rather an unfamiliar beast called dict_proxy. And if all that wasn't enough, recall the mystery from the end of Objects 101: if plain object instances like o have no __dict__, and C extends object without adding anything significant, why do instances of C like o2 suddenly do have a __dict__?

First, we shall look at the implementation of a type's __dict__. Looking at the definition of PyObjectType (a zesty and highly recommended exercise), we see a slot called tp_dict, ready to accept a pointer to a dictionary. All types must have this slot, and all types have a dictionary placed there when ./Objects/typeobject.c: PyType_Ready is called on them, either when the interpreter is first initialized (remember Py_Initialize? It invokes _Py_ReadyTypes which calls PyType_Ready on all known types) or when the type is created dynamically by the user (type_new calls PyType_Ready on the newborn type before returning).

In fact, every name you bind within a class statement will turn up in the newly created type's __dict__ (see ./Objects/typeobject.c: type_new: type->tp_dict = dict = PyDict_Copy(dict);). These functions use the dictionary each type has and pointed to by tp_dict to store/retrieve the attributes, that is, getting attributes on a type is directly wired to dictionary assignment for the type instance's private dictionary pointed to by the type's structure. So far I hope it's been rather simple, and explains types' attribute retrieval.

Descriptors

Descriptors play a special role in instances' attribute access. An object is said to be a descriptor if it's type has one or two slots (tp_descr_get and/or tp_descr_set) filled with non-NULL value. These slots are wired to the special method names __get__, __set__ and __delete__, when the type is defined in pure Python (i.e., if you create a class which has a __get__ method it will be wired to its tp_descr_get slot, and if you instantiate an object from that class, the object is a descriptor).

An object is said to be a data descriptor if its type has a non-NULL tp_descr_set slot (there's no particularly special term for a non-data descriptor). We've defined descriptors, and we know how types' dictionaries and attribute access work. Most objects aren't types, that is to say, their type isn't type, it's something more mundane like int or dict or a user defined class. All these rely on generic attribute access functions, which are either set on the type explicitly or inherited from the type's base when the type is created.

The generic attribute-getting function (PyObject_GenericGetAttr) and its algorithm is like so:

(a) Search the accessed instance's type's dictionary, and then all the type's bases' dictionaries. If a data descriptor was found, invoke it's tp_desr_get function and return the results. If something else is found, set it aside (we'll call it X).

(b) Now search the object's dictionary, and if something is found, return it.

(c) If nothing was found in the object's dictionary, inspect X, if one was set aside at all; if X is a non-data descriptor, invoke it's tp_descr_get function and return the result, and if it's a plain object it returns it.

(d) Finally, if nothing was found, it raise an `AttributeError` exception.

So we learn that descriptors can execute code when they're accessed as an attribute (so when you do `foo = o.a or o.a = foo`, a runs code). A powerful notion, that, and it's used in several cases to implement some of Python's more 'magical' features.

Data-descriptors are even more powerful, as they take precedence over instance attributes (if you have an `object o of class C`, class C has a foo data-descriptor and o has a foo instance attribute, when you do o.foo the descriptor will take precedence).

While descriptors are really important and you're advised to take the time to understand them, for brevity and due to the well written resources I've just mentioned I will explain them no further, other than show you how they behave in the interpreter (super simple example!):

```
>>> class ShoutingInteger(int):
...     # __get__ implements the tp_descr_get slot
...     def __get__(self, instance, owner):
...             print('I was gotten from %s (instance of %s)'
...                   % (instance, owner))
...             return self
...
>>> class Foo:
...     Shouting42 = ShoutingInteger(42)
...
>>> foo = Foo()
>>> 100 - foo.Shouting42
I was gotten from <__main__.Foo object at 0xb7583c8c> (instance of <class
__main__.'foo'>)
58
# Remember: descriptors are only searched on types!
>>> foo.Silent666 = ShoutingInteger(666)
>>> 100 - foo.Silent666
-566
>>>
```

We now understand that accessing attribute A on object O instantiated from class C1 which inherits C2 which inherits C3 can return A either from O, C1, C2 or C3, depending on something called the `method resolution order`. This way of resolving attributes, when coupled with slot inheritance, is enough to explain most of Python's inheritance functionality.

We've seen the definition of `PyObject`, and it most definitely didn't have a pointer to a dictionary, so where is the reference the object's dictionary stored? If you look closely at the definition of `PyTypeObject`, you will see a field called `tp_dictoffset`. This field provides a byte offset into the C-structure allocated for objects instantiated from this type; at this offset, a pointer to a regular Python dictionary should be found.

Under normal circumstances, when creating a new type, the size of the memory region necessary to allocate objects of that type will be calculated, and that size will be larger than the size of vanilla `PyObject`. The extra room will typically be used (among other things) to store the pointer to the dictionary (all this happens in `./Objects/typeobject.c` : `type_new, see may_add_dict = base->tp_dictoffset == 0`; onwards).:

```
>>> class C: pass
...
>>> o = C()
>>> o.foo = 'bar'
>>> o
<__main__.C object at 0x846b06c>
>>>
# break into GDB, see 'metablogging'->'tools' above
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0012d422 in __kernel_vsyscall ()
(gdb) p ((PyObject *)(0x846b06c))->ob_type->tp_dictoffset
$1 = 16
(gdb) p *((PyObject **)(((char *)0x846b06c)+16))
$3 = {u'foo': u'bar'}
(gdb)
```

We have created a new class, instantiated an object from it and set some attribute on the object (o.foo = ‘bar’), broke into gdb, dereferenced the object’s type (C) and checked its `tp_dictoffset` (it was 16), and then checked what’s to be found at the address pointed to by the pointer located at 16 bytes’ offset from the object’s C-structure, and indeed we found there a dictionary object with the key foo pointing to the value bar.

Of course, if you check `tp_dictoffset` on a type which doesn’t have a __dict__, like object, you will find that it is zero. I define a class C inheriting object and doing nothing much else in Python, and then I instantiate o from that class, causing the extra memory for the dictionary pointer to be allocated at `tp_dictoffset`.

I then type in my interpreter `o.__dict__`, which byte-compiles to the `LOAD_ATTR` opcode, which causes the `PyObject_GetAttr` function to be called, which dereferences the type of o and finds the `slot tp_getattro`, which causes the default attribute searching mechanism described earlier in this post and implemented in `PyObject_GenericGetAttr`.

So when all that happens, what returns my object’s dictionary? I know where the dictionary is stored, but I can see that __dict__ isn’t recursively inside itself, so there’s a chicken and egg problem here; who gives me my dictionary when I access __dict__ if it is not in my dictionary?

Someone who has precedence over the object’s dictionary - a descriptor. Check this out:

```
>>> class C: pass
...
>>> o = C()
>>> o.__dict__
{}
>>> C.__dict__['__dict__']
<attribute '__dict__' of 'C' objects>
>>> type(C.__dict__['__dict__'])
<class 'getset_descriptor'>
>>> C.__dict__['__dict__'].__get__(o, C)
{}
>>> C.__dict__['__dict__'].__get__(o, C) is o.__dict__
True
>>>
```

Seems like there's something called `getset_descriptor` (it's in `./Objects/typeobject.c`), which are groups of functions implementing the descriptor protocol and meant to be attached to an object placed in type's __dict__.

This descriptor will intercept all attribute access to `o.__dict__` on instances of this type, and will return whatever it wants, in our case, a reference to the dictionary found at the `tp_dictoffset` of o.

This is also the explanation of the dict_proxy business we've seen earlier. If in `tp_dict` there's a pointer to a plain dictionary, what causes it to be returned wrapped in this read only proxy, and why? The __dict__ descriptor of the type's type type does it.:

```
>>> type(C)
<class 'type'>
>>> type(C).__dict__['__dict__']
<attribute '__dict__' of 'type' objects>
>>> type(C).__dict__['__dict__'].__get__(C, type)
<dict_proxy object at 0xb767e494>
```

This descriptor is a function that wraps the dictionary in a simple object that mimics regular dictionaries' behaviour but only allows read only access to the dictionary it wraps. And why is it so important to prevent people from messing with a `type's __dict__`? Because a type's namespace might hold them specially named methods, like `__sub__`.

When you create a type with these specially named methods or when you set them on the type as an attribute, the function `update_one_slot` will patch these methods into one of the type's slots, as we've seen in 101 for the subtraction operation. If you were to add these methods straight into the type's __dict__, they won't be wired to any slot, and you'll have a type that looks like it has a certain behaviour (say, has __sub__ in its dictionary), but doesn't behave that way. `__slots__` are important construct when dealing with attributes access.

descriptors are objects whose type has their tp_descr_get and/or tp_descr_set slots set to non-NULL. However, I also wrote, incorrectly, that descriptors take precedence over regular instance attributes (i.e., attributes in the object's __dict__). This is partly correct but misleading, as it doesn't distinguish non-data descriptors from data-descriptors. An object is said to be a data descriptor if its type has its tp_descr_set slot implemented (there's no particularly special term for a non-data descriptor). Only data descriptors override regular object attributes, non-data descriptors do not.

Interpreter Threads

Look into the Interpreter State and the Thread State structures both implemented in ./Python/pystate.c In many operating systems user-space code is executed by an abstraction called threads that run inside another abstraction called processes. The kernel is in charge of setting up and tearing down these processes and execution threads, as well as deciding which thread will run on which logical CPU at any given time.

When a process invokes Py_Initialize another abstraction comes into play, and that is the interpreter. Any Python code that runs in a process is tied to an interpreter, you can think of the interpreter as the root of all other concepts we'll discuss. Python's code base supports initializing two (or more) completely separate interpreters that share little state with one another. This is rather rarely done (never in the vanilla executable), because too much subtly shared state of the interpreter core and of C extensions exists between these 'insulated' interpreters.

Anyhow, we said all execution of code occurs in a thread (or threads), and Python's Virtual Machine is no exception. However, Python's Virtual Machine itself is something which supports the notion of threading, so Python has its own abstraction to represent Python threads. This abstraction's implementation is fully reliant on the kernel's threading mechanisms, so both the kernel and Python are aware of each Python thread and Python threads execute as separate kernel-managed threads, running in parallel with all other threads in the system. Uhm, almost.

Many aspects of Python's CPython implementation are not thread safe. This is has some benefits, like simplifying the implementation of easy-to-screw-up pieces of code and guaranteed atomicity of many Python operations, but it also means that a mechanism must be put in place to prevent two (or more) Pythonic The GIL is a process-wide lock which must be held by a thread if it wants to do anything Pythonic - effectively limiting all such work to a single thread running on a single logical CPU at a time. Threads in Python multitask cooperatively by relinquishing the GIL voluntarily so other threads can do Pythonic work; this cooperation is built-in to the evaluation loop, so ordinarily authors of Python code and some extensions don't need to do something special to make cooperation work (from their point of view, they are preempted).

Do note that while a thread doesn't use any of Python's APIs it can (and many threads do) run in parallel to another Pythonic thread. With the concepts of a process (OS abstraction), interpreter(s) (Python abstraction) and threads (an OS abstraction and a Python abstraction) in mind, let's go inside-out by zooming out from a single opcode outwards to the whole process.

Let's look again at the disassembly of the bytecode generated for the simple statement `spam = eggs - 1`:

```
# uses 'diss'? tool.
>>> diss("spam = eggs - 1")
  1           0 LOAD_NAME                0 (eggs)
              3 LOAD_CONST               0 (1)
              6 BINARY_SUBTRACT
              7 STORE_NAME               1 (spam)
             10 LOAD_CONST               1 (None)
             13 RETURN_VALUE
>>>
```

In addition to the actual ‘do work’ opcode BINARY_SUBTRACT, we see opcodes like LOAD_NAME (eggs) and STORE_NAME (spam). It seems obvious that evaluating such opcodes requires some storage room: eggs has to be loaded from somewhere, spam has to be stored somewhere.

The inner-most data structures in which evaluation occurs are the frame object and the code object, and they point to this storage room. When you’re “running” Python code, you’re actually evaluating frames (recall `ceval.c: PyEval_EvalFrameEx`).

In this code-structure-oriented post, the main thing we care about is the `f_back` field of the frame object (though many others exist). In `frame n` this field points to frame n-1, i.e., the frame that called us (the first frame that was called in any particular thread, the top frame, points to NULL). This stack of frames is unique to every thread and is anchored to the thread-specific structure `./Include.h/pystate.h: PyThreadState`, which includes a pointer to the currently executing frame in that thread (the most recently called frame, the bottom of the stack).

PyThreadState is allocated and initialized for every Python thread in a process by `_PyThreadState_Prealloc` just before new thread creation is actually requested from the underlying OS (see `./Modules/_threadmodule.c: thread_PyThread_start_new_thread` and `>>> from _thread import start_new_thread`). Threads can be created which will not be under the interpreter’s control; these threads won’t have a `PyThreadState` structure and must never call a Python API. This isn’t so common in a Python application but is more common when Python is embedded into another application. It is possible to ‘Pythonize’ such foreign threads that weren’t originally created by Python code in order to allow them to run Python code (PyThreadState will have to be allocated for them). Finally, a bit like all frames are tied together in a backward-going stack of previous-frame pointers, so are all thread states tied together in a linked list of `PyThreadState *next` pointers.

The list of thread states is anchored to the interpreter state structure which owns these threads. The interpreter state structure is defined at `./Include.h/pystate.h: PyInterpreterState`, and it is created when you call `Py_Initialize` to initialize the Python VM in a process or `Py_NewInterpreter` to create a new interpreter state for multi-interpreter processes. Note carefully that `Py_NewInterpreter` does not return an interpreter state – it returns a (newly created) `PyThreadState` for the single automatically created thread of the newly created interpreter.

There’s no sense in creating a new interpreter state without at least one thread in it, much like there’s no sense in creating a new process with no threads in it.

Similarly to the list of threads anchored to its interpreter, so does the interpreter structure have a next field which forms a list by linking the interpreters to one another.This pretty much sums up our zooming out from the resolution of a single opcode to the whole process: opcodes belong to currently evaluating code objects (currently evaluating is specified as opposed to code objects which are just lying around as data, waiting for the opportunity to be called), which belong to currently evaluating frames, which belong to Pythonic threads, which belong to interpreters. The anchor which holds the root of this structure is the static variable `./Python/pystate.c: interp_head`, which points to the first interpreter state (through that all interpreters are reachable, through each of them all thread states are reachable, and so fourth).

The mutex `head_mutex` protects `interp_head` and the lists it points to so they won't be corrupt by concurrent modifications from multiple threads (I want it to be clear that this lock is not the GIL, it's just the mutex for interpreter and thread states). The macros `HEAD_LOCK` and `HEAD_UNLOCK` control this lock. `interp_head` is typically used when one wishes to add/remove interpreters or threads and for special purposes. That's because accessing an interpreter or a thread through the head variable would get you an interpreter state rather than the interpreter state owning the currently running thread (just in case there's more than one interpreter state).

A more useful variable similar to interp_head is `./Python/pystate.c: _PyThreadState_Current` which points to the currently running thread state This is how code typically accesses the correct interpreter state for itself: first find its your own thread's thread state, then dereference its interp field to get to your interpreter.

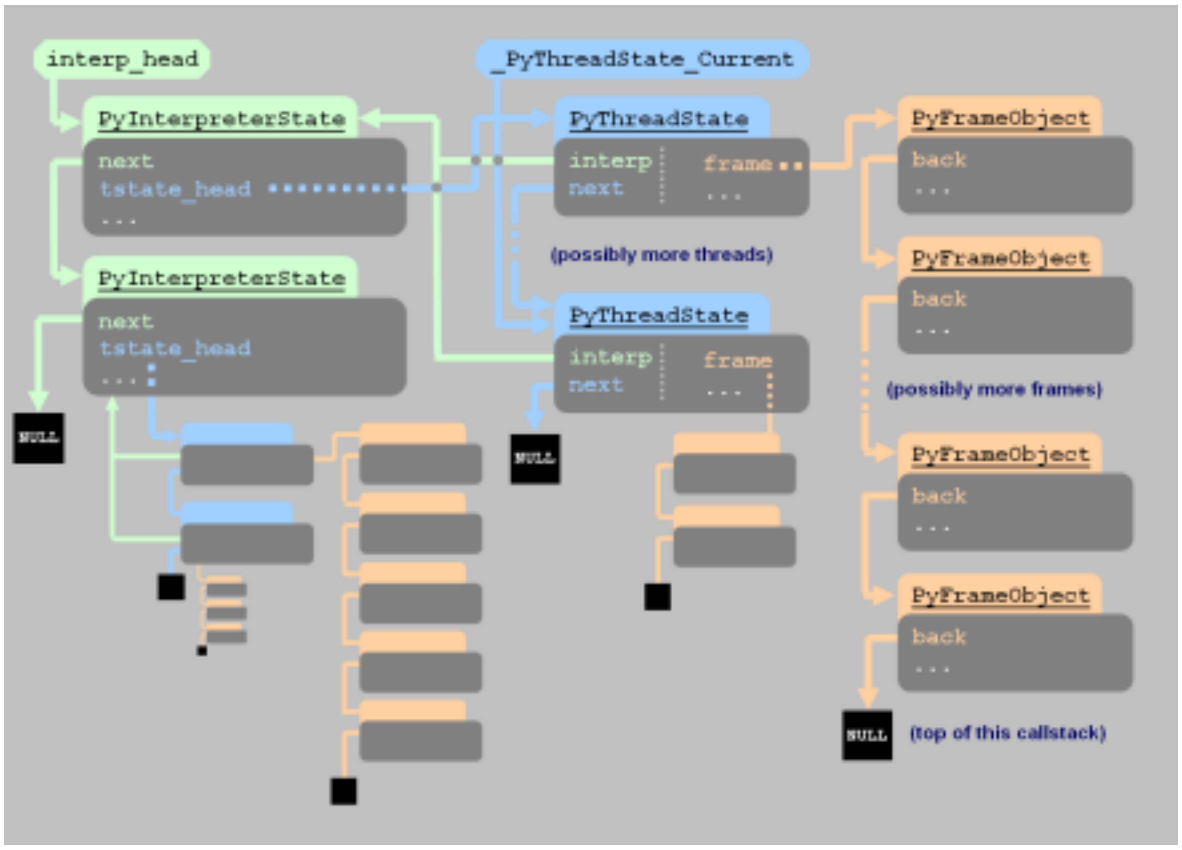
There are a couple of functions that let you access this variable (get its current value or swap it with a new one while retaining the old one) and they require that you hold the GIL to be used. This is important, and serves as an example of CPython's lack of thread safety (a rather simple one, others are hairier). If two threads are running and there was no GIL, to which thread would this variable point? "The thread that holds the GIL" is an easy answer, and indeed, the one that's used. `_PyThreadState_Current` is set during Python's initialization or during a new thread's creation to the thread state structure that was just created. When a Pythonic thread is bootstrapped and starts running for the very first time it can assume two things:

- It holds the GIL and

- It will find a correct value in _PyThreadState_Current.

As of that moment the Pythonic thread should not relinquish the GIL and let other threads run without first storing `_PyThreadState_Current` somewhere, and should immediately re-acquire the GIL and restore `_PyThreadState_Current` to its old value when it wants to resume running Pythonic code. This behaviour is what keeps `_PyThreadState_Current` correct for GIL-holding threads and is so common that macros exist to do the save-release/acquire-restore idioms (`Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`). There's much more to say about the GIL and additional APIs to handle it and it's probably also interesting to contrast it with other Python implementation (Jython and IronPython are thread safe and do run Pythonic threads concurrently).

Diagram shows the relation between the state structures within a single process hosting Python as described so far. We have in this example two interpreters with two threads each, you can see each of these threads points to its own call stack of frames.



Interpreter states contain several fields dealing with imported modules of that particular interpreter, so we can talk about that when we talk about importing.

In addition to managing imports they hold bunch of pointers related to handling Unicode codecs, a field to do with dynamic linking flags and a field to do with TSC usage for profiling. Thread states have more fields but to me they were more easily understood. Not too surprisingly, they have fields that deal with things that relate to the execution flow of a particular thread and are of too broad a scope to fit particular frame.

Take for example the fields recursion_depth, overflow and recursion_critical, which are meant to trap and raise a RuntimeError during overly deep recursions before the stack of the underlying platform is exhausted and the whole process crashes. In addition to these fields, this structure accommodates fields related to profiling and tracing, exception handling (exceptions can be thrown across frames), a general purpose per-thread dictionary for extensions to store arbitrary stuff in and counters to do with deciding when a thread ran too much and should voluntarily relinquish the GIL to let other threads run.

Naming

Discuss naming, which is the ability to bind names to an object, like we can see in the statement `a = 1` (in other words, this article is roughly about what many languages call variables). Naturally, naming is central to Python's behaviour and understanding both its semantics and mechanics are important precursors to our quickly approaching discussions of code evaluation, code objects and stack frames.

That said, it is also a delicate subject because anyone with some programming experience knows something about it, at least instinctively (you've done something like a = 1 before, now haven't you?).When we evaluate a = b = c = [], we create one list and give it three different names. In formal terms, we'd say that the newly instantiated list object is now bound to three identifiers that refer to it. This distinction between names and the objects bound to them is important. If we evaluate a.append(1), we will see that b and c are also affected; we didn't mutate a, we mutated its referent, so the mutation is uniformly visible via any name the object was referred to.

On the other hand, if we will now do a `b = []`, a and c will not change, since we didn't actually change the object which b referred to but rather did a re-binding of the name b to a (newly created and empty) list object. Also recall that binding is one of the ways to increase the referent's reference count, this is worthy of noting even though reference counting isn't our subject at the moment.

A name binding is commonly created by use of the assignment statement, which is a statement that has an 'equals' symbol (=) in the middle, "stuff to assign to" or targets on the left, and "stuff to be assigned" (an expression) on the right. A target can be a name (more formally called an identifier) or a more complex construct, like a sequence of names, an attribute reference (primary_name.attribute) or a subscript (primary_name[subscript])

Name binding is undone with the deletion statement del, which is roughly "del followed by comma-separated targets to unbind"

Finally, note that name binding can be done without an assignment as bindings are also created by `def, class, import (and others)`, this is also of less importance to us now.

Scope is a term relating to the visibility of an identifier throughout a block, or a piece of Python code executed as a unit: a module, a function body and a class definition are blocks (control-blocks like those of if and while are not code blocks in Python). A namespace is an abstract environment where the mapping between names and the objects they refer to is made (incidentally, in current CPython, this is indeed implemented with the dict mapping type). The rules of scoping determine in which namespace will a name be sought after when it is used, or rather resolved.

You probably know instinctively that a name bound in function foo isn't visible in an unrelated function bar, this is because by default names created in a function will be stored in a namespace that will not be looked at when name resolution happens in another, unrelated function.

Scope determines not just when a name will be visible as it is resolved or 'read' (i.e., if you do spam = eggs, where will eggs come from) but also as it is bound or 'written' (i.e., in the same example, where will spam go to). When a namespace will no longer be used (for example, the private namespace of a function which returns) all the names in it are unbound (this triggers reference count decrease and possibly deallocation, but this doesn't concern us now).

Scoping rules change based on the lexical context in which code is compiled. For example, in simpler terms, code compiled as a plain function's body will resolve names slightly differently when evaluated when compared with code compiled as part of a module's initialization code (the module top-level code). Special statements like global and nonlocal exist and can be applied to names thus that resolution rules for these names will change in the current code block, we'll look into that later.

When Python code is evaluated, it is evaluated within three namespaces: locals, globals and builtins. When we resolve a name, it will be sought after in the local scope, then the global scope, then the builtin scope (then a NameError will be raised). When we bind a name with a name binding statement (i.e., an assignment, an import, a def, etc) the name will be bound in the local scope, and hide any existing names in the global or builtin scope.

This hiding does not mean the hidden name was changed (formally: the hidden name was not re-bound), it just means it is no longer visible in the current block's scope because the newly created binding in the local namespace overshadows it.

We said scoping changes according to context, and one such case is when functions are lexically nested within one another (that is, a function defined inside the body of another function): resolution of a name from within a nested function will first search in that function's scope, then in the local scopes of its outer function(s) and only then proceed normally (in the globals and builtins) scope.

Lexical scoping is an interesting behaviour, let's look at it closely:

```
$ cat scoping.py ; python3.1
def outer():
    a = 1
    # creating a lexically nested function bar
    def inner():
        # a is visible from outer's locals
        return a
    b = 2 # b is here for an example later on
    return inner

# inner_nonlexical will be called from within
#  outer_nonlexical but it is not lexically nested
def inner_nonlexical():
    return a # a is not visible
def outer_nonlexical():
    a = 1
    inner = inner_nonlexical
    b = 2 # b is here for an example later on
    return inner_nonlexical
>>> from scoping import *
>>> outer()()
1
>>> outer_nonlexical()()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "scoping.py", line 13, in inner_nonlexical
    return a # a is not visible
>>>
```

As the example demonstrates, a is visible in the lexically nested inner but not in the call-stack nested but not lexically nested inner_nonlexical. I mean, Python is dynamic, everything is runtime, how does inner_nonlexical fail if it has the same Python code and is called in a similar fashion from within a similar environment as the original inner was called?

Further more, we can see that `inner` is actually called after `outer` has terminated: how can it use a value from a namespace that was already destroyed?

Once again, let's look at the bytecode emitted for the simple statement `spam = eggs - 1`:

```
>>> diss("spam = eggs - 1")
  1           0 LOAD_NAME                0 (eggs)
              3 LOAD_CONST               0 (1)
              6 BINARY_SUBTRACT
              7 STORE_NAME               1 (spam)
             10 LOAD_CONST               1 (None)
             13 RETURN_VALUE
>>>
```

Recall that BINARY_SUBTRACT will pop two arguments from the value-stack and feed them to `PyNumber_Subtract`, which is a C function that accepts two `PyObject * pointers` and certainly doesn't know anything about scoping.

What gets the arguments onto the stack are the `LOAD_NAME` and `LOAD_CONST` opcodes, and what will take the result out of the stack and into wherever it is heading is the `STORE_NAME` opcode. It is opcodes like this that implement the rules of naming and scoping, since the C code implementing them is what will actually look into the dictionaries representing the relevant namespaces trying to resolve the name and bring the resulting object unto the stack, or store whatever object is to be stored into the relevant namespace.

For example, take `LOAD_CONST`; this opcode loads a constant value unto the value stack, but it isn't about scoping (constants don't have a scope, by definition they aren't variables and they're never 'hidden').

Fortunately for you, I've already grepped the sources for 'suspect' opcodes ($ egrep -o '(LOAD|STORE)(_[A-Z]+)+' Include/opcode.h | sort) and believe I've mapped out the opcodes that actually implement scoping, so we can concentrate on the ones that really implement scoping. Note that among the list of opcodes I chose not to address are the ones that handles attribute reference and subscripting; I chose so since these opcodes rely on a different opcode to get the primary reference (the name before the dot or the square brackets) on the value stack and thus aren't really about scoping.

- •We should discuss four pairs of opcode:
    `LOAD_NAME and STORE_NAME`
- `LOAD_FAST and STORE_FAST`
- `LOAD_GLOBAL and STORE_GLOBAL`
- `LOAD_DEREF and STORE_DEREF`
-

I suggest we discuss each pair along with the situations in which the compiler chooses to emit an opcode of that pair in order to satisfy the semantics of scoping.

This is not necessarily an exhaustive listing of these opcodes' uses (it might be, I'm not checking if it is or isn't), but it should develop an understanding of these opcodes' behaviour and allow us to figure out other cases where the compiler chooses the emit them on our own; so if you ever see any of these in a disassembly, you'll be covered.

I'd like to begin with the obvious pair, `*_NAME`; it is simple to understand (and I suspect it was the first to be implemented). Explaining the `*_NAME` pair of opcodes is easiest by writing rough versions of them in Python-like psuedocode (you can and should read the actual implementation in `./Python/ceval.c: PyEval_EvalFrameEx`):

```
def LOAD_NAME(name):
    try:
        return current_stack_frame.locals[name]
    except KeyError:
        try:
            return current_stack_frame.globals[name]
        except KeyError:
            try:
                return current_stack_frame.builtins[name]
            except KeyError:
                raise NameError('name %r is not defined'
                                 % name)

def STORE_NAME(name, value):
    current_stack_frame.locals[name] = value
```

While they are the 'vanilla' case, `*_NAME`, in some cases they are not emitted at all as more specialized opcodes can achieve the same functionality in a faster manner. As we explore the other scoping-related opcodes, we will see why. A commonly used pair of scoping related opcodes is the `*_FAST` pair, which were originally implemented a long time ago as a speed enhancement over the `*_NAME` pair.

These opcodes are used in cases where compile time analysis can infer that a variable is used strictly in the local namespace. This is possible when compiling code which is a part of a function, rather than, say, at the module level (some subtleties apply about the meaning of 'function' in this context, a class' body may also use these opcodes under some circumstances, but this is of no interest to us at the moment; also see the comments below).

If we can decide at compile time which names are used in precisely one namespace, and that namespace is private to one code block, it may be easy to implement a namespace with cheaper machinery than dictionaries. Indeed, these opcodes rely on a local namespace implemented with a statically sized array, which is far faster than a dictionary lookup as in the global namespace and other namespaces.

In Python 2.x it was possible to confuse the compiler thus that it will not be able to use these opcodes in a particular function and have to revert to `*_NAME`, this is no longer possible in Python 3.x (also see the comments).

Let's look at the two `*_GLOBAL` opcodes. LOAD_GLOBAL (but not STORE_GLOBAL) is also generated when the compiler can infer that a name is resolved in a function's body but was never bound inside that body.

This behaviour is conceptually similar to the ability to decide when a name is both bound and resolved in a function's body, causing the generation of the `*_FAST` opcodes as we've seen above:

```
>>> def func():
...     a = 1
...     a = b
...     return a
...
>>> diss(func)
  2           0 LOAD_CONST               1 (1)
              3 STORE_FAST               0 (a)
  3           6 LOAD_GLOBAL              0 (b)
              9 STORE_FAST               0 (a)
  4          12 LOAD_FAST                0 (a)
             15 RETURN_VALUE
>>>
```

As described for `*_FAST`, we can see that a was bound within the function, which places it in the local scope private to this function, which means the `*_FAST` opcodes can and are used for a. On the other hand, we can see (and the compiler could also see…) that b was resolved before it was ever bound in the function.

The compiler figured it must either exist elsewhere or not exist at all, which is exactly what `LOAD_GLOBAL` does: it bypasses the local namespace and searches only the `global` and `builtin` namespaces (and then raises a `NameError`).

This explanation leaves us with missing functionality: what if you'd like to re-bind a variable in the global scope? Recall that binding a new name normally binds it locally, so if you have a module defining foo = 1, a function setting foo = 2 locally "hides" the global foo.

But what if you want to re-bind the global foo? Note this is not to mutate object referred to by foo but rather to bind the name foo in the global scope to a different referent; if you're not clear on the distinction between the two, skim back in this post until we're on the same page.

To do so, we can use the global statement which we mentioned in passing before; this statement lets you tell the compiler to treat a name always as a global both for resolving and for binding within a particular code block, generating only `*_GLOBAL` opcodes for manipulation of that name.

When binding is required, `STORE_GLOBAL` performs the new binding (or a re-binding) in the global namespace, thus allowing Python code to explicitly state which variables should be stored and manipulated in the global scope. What happens if you use a variable locally, and then use the global statement to make it global? Let's look (slightly edited):

```
>>> def func():
...     a = 1
...     global a
...
<stdin>:3: SyntaxWarning: name 'a' is assigned to before global declaration
>>> diss(func)
  2           0 LOAD_CONST               1 (1)
              3 STORE_GLOBAL             0 (a)
  3           6 LOAD_CONST               0 (None)
              9 RETURN_VALUE
>>>
```

The compiler still treats the name as a global all through the code block, but warns you not to shoot yourself (and other maintainers of the code) in the foot. Sensible.

We are left only with `LOAD_DEREF` and `STORE_DEREF`. To explain these, we have to revisit the notion of lexical scoping, which is what started our inspection of the implementation. Recall that we said that nested functions' resolution of names tries the namespaces' of all lexically enclosing functions (in order, innermost outwards) before it hits the global namespace, we also saw an example of that in code.

So how did inner return a value resolved from this no-longer-existing namespace of outer? When resolution of names is attempted in the global namespace (or in builtins), the name may or may not be there, but for sure we know that the scope is still there! How do we resolve a name in a scope which doesn't exist?

The answer is quite nifty, and becomes apparent with a disassembly (slightly edited) of both functions:

```
# see the example above for the contents of scoping.py
>>> from scoping import *
# recursion added to 'diss'; you can see metablogging->tools above
>>> diss(outer, recurse=True)
  2           0 LOAD_CONST               1 (1)
              3 STORE_DEREF              0 (a)
  3           6 LOAD_CLOSURE             0 (a)
              9 BUILD_TUPLE              1
             12 LOAD_CONST               2 (<code object inner ...)
             15 MAKE_CLOSURE             0
             18 STORE_FAST               0 (inner)
  5          21 LOAD_CONST               3 (2)
             24 STORE_FAST               1 (b)
  6          27 LOAD_FAST                0 (inner)
             30 RETURN_VALUE

recursing into <code object inner ...>:
  4           0 LOAD_DEREF               0 (a)
              3 RETURN_VALUE
>>>
```

We can see that outer (the outer function!) already treats a, the variable which will be used outside of its scope, differently than it treats b, a 'simple' variable in its local scope.

`a` is loaded and stored using the `*_DEREF` variants of the loading and storing opcodes, in both the outer and inner functions. The secret sauce here is that at compilation time, if a variable is seen to be resolved from a lexically nested function, it will not be stored and will not be accessed using the regular naming opcodes. Instead, a special object called a cell is created to store the value of the object. When various code objects (the outer function, the inner function, etc) will access this variable, the use of the `*_DEREF` opcodes will cause the cell to be accessed rather than the namespace of the accessing code object. Since the cell is actually accessed only after outer has finished executing, you could even define inner before a was defined, and it would still work just the same (!).

This is automagical for name resolution, but for outer scope rebinding the nonlocal statement exists. nonlocal was decreed by PEP 3014 and it is somewhat similar to the global statement

`nonlocal` explicitly declares a variable to be used from an outer scope rather than locally, both for resolution and re-binding. It is illegal to use nonlocal outside of a lexically nested function, and it must be nested inside a function that defines the identifiers listed by nonlocal.

There are several small gotchas about lexical scoping, but overall things behave as you would probably expect (for example, you can't cause a name to be used locally and as a lexically nested name in the same code block, as the collapsed snippet below demonstrates):

```
>>> def outer():
...     a = 1
...     def inner():
...             b = a
...             a = 1
...             return a,b
...     return inner
...
>>> outer()()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in inner
UnboundLocalError: local variable 'a' referenced before assignment
>>>
```

This sums up the mechanics of naming and scoping.

Byte Code

The compilation of Python source code emits Python bytecode, which is evaluated at runtime to produce whatever behaviour the programmer implemented. I guess you can think of bytecode as 'machine code for the Python virtual machine', and indeed if you look at some binary x86 machine code (like this one: 0x55 0x89 0xe5 0xb8 0x2a 0x0 0x0 0x0 0x5d) and some Python bytecode (like that one: 0x64 0x1 0x0 0x53) they look more or less like the same sort of gibberish.

The bytecode and these fields are lumped together in an object called a code object, our subject for this article.

You might initially confuse function objects with code objects, but shouldn't. Functions are higher level creatures that execute code by relying on a lower level primitive, the code object, but adding more functionality on top of that (in other words, every function has precisely one code object directly associated with it, this is the function's `__code__` attribute, or `f_code` in Python 2.x).

For example, among other things, a function keeps a reference to the global namespace (remember that?) in which it was originally defined, and knows the default values of arguments it receives. You can sometimes execute a code objects without a function (see eval and exec), but then you will have to provide it with a namespace or two to work in.

Finally, just for accuracy's sake, please note that `tp_call` of a function object isn't exactly like `exec` or `eval`; the latter don't pass in arguments or provide free argument binding (more below on these).

If this doesn't sit well with you yet, don't panic, it just means functions' code objects won't necessarily be executable using eval or exec. I hope we have that settled.

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`.

As usual, I don't want to dig too deeply into compilation, but basically when a code block is encountered, it has to be successfully transformed into an AST (which requires mostly that its syntax will be correct), which is then passed to `./Python/compile.c: PyAST_Compile`, the entry point into Python's compilation machinary.

You absolutely can't run this code meaningfully without its constants, and indeed 42 is referred to by one of the extra fields of the code object. We will best see the interaction between the actual bytecode and the accompanying fields as we do a manual disassembly:

```
# the opcode module has a mapping of opcode
#  byte values to their symbolic names
>>> import opcode
>>> def return42(): return 42
...
# this is the function's code object
>>> return42.__code__
<code object return42 ... >
# this is the actual bytecode
>>> return42.__code__.co_code
b'd\x01\x00S'
# this is the field holding constants
>>> return42.__code__.co_consts
(None, 42)
# the first opcode is LOAD_CONST
>>> opcode.opname[return42.__code__.co_code[0]]
'LOAD_CONST'
# LOAD_CONST has one word as an operand
#  let's get its value
>>> return42.__code__.co_code[1] + \
... 256 * return42.__code__.co_code[2]
1
# and which constant can we find in offset 1?
>>> return42.__code__.co_consts[1]
42
# finally, the next opcode
>>> opcode.opname[return42.__code__.co_code[3]]
'RETURN_VALUE'
>>>
```

In addition to dis, the function show_code from the same module is useful to look at code objects:

```
>>> diss(return42)
  1           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
>>> ssc(return42)
Name:              return42
Filename:          <stdin>
Argument count:    0
Kw-only arguments: 0
Number of locals:  0
Stack size:        1
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: None
   1: 42
>>>
```

We see diss and ssc generally agree with our disassembly, though ssc further parsed all sorts of other fields of the code object which we didn’t handle so far (you can run dir on a code object to see them yourself). Code objects are immutable and their fields don’t hold any references (directly or indirectly) to mutable objects. This immutability is useful in simplifying many things, one of which is the handling of nested code blocks.

An example of a nested code block is a class with two methods: the class is built using a code block, and this code block nests two inner code blocks, one for each method.

This situation is recursively handled by creating the innermost code objects first and treating them as constants for the enclosing code object (much like an integer or a string literal would be treated). Now that we have seen the relation between the bytecode and a code object field (co_consts), let's take a look at the myriad of other fields in a code object. Many of these fields are just integer counters or tuples of strings representing how many or which variables of various sorts are used in a code object. But looking to the horizon where ceval.c and frame object evaluation is waiting for us, I can tell you that we need an immediate and crisp understanding of all these fields and their exact meaning, subtleties included.

- •Identity or origin (strings)

co_name

A name (a string) for this code object; for a function this would be the function's name, for a class this would be the class' name, etc. The compile builtin doesn't let you specify this, so all code objects generated with it carry the name <module>.

co_filename

The filename from which the code was compiled. Will be <stdin> for code entered in the interactive interpreter or whatever name is given as the second argument to compile for code objects created with compile.

- •Different types of names (string tuples)

co_varnames

A tuple containing the names of the local variables (including arguments). To parse this tuple properly you need to look at co_flags and the counter fields listed below, so you'll know which item in the tuple is what kind of variable. In the 'richest' case, co_varnames contains (in order): positional argument names (including optional ones), keyword only argument names (again, both required and optional), varargs argument name (i.e., `*args`), kwds argument name (i.e., `**kwargs`), and then any other local variable names. So you need to look at co_argcount, co_kwonlyargcount and co_flags to fully interpret this tuple.

co_cellvars

A tuple containing the names of local variables that are stored in cells (discussed in the previous article) because they are referenced by lexically nested functions.

co_freevars

A tuple containing the names of free variables. Generally, a free variable means a variable which is referenced by an expression but isn't defined in it. In our case, it means a variable that is referenced in this code object but was defined and will be dereferenced to a cell in another code object (also see co_cellvars above and, again, the previous article).

co_names

A tuple containing the names which aren’t covered by any of the other fields (they are not local variables, they are not free variables, etc) used by the bytecode. This includes names deemed to be in the global or builtin namespace as well as attributes (i.e., if you do foo.bar in a function, bar will be listed in its code object’s names).

- •Counters and indexes (integers)

co_argcount

The number of positional arguments the code object expects to receive, including those with default values. For example, def foo(a, b, c=3): pass would have a code object with this value set to three. The code object of classes accept one argument which we will explore when we discuss class creation.

co_kwonlyargcount

The number of keyword arguments the code object can receive.

co_nlocals

The number of local variables used in the code object (including arguments).

co_firstlineno

The line offset where the code object’s source code began, relative to the module it was defined in, starting from one. In this (and some but not all other regards), each input line typed in the interactive interpreter is a module of its own.

co_stacksize

The maximum size required of the value stack when running this object. This size is statically computed by the compiler (./Python/compile.c: stackdepth when the code object is created, by looking at all possible flow paths searching for the one that requires the deepest value stack. To illustrate this, look at the diss and ssc outputs for a = 1 and a = [1,2,3]. The former has at most one value on the value stack at a time, the latter has three, because it needs to put all three integer literals on the stack before building the list.

- •Other stuff (various)

co_code

A string representing the sequence of bytecode instructions, contains a stream of opcodes and their operands (or rather, indexes which are used with other code object fields to represent their operands, as we saw above).

co_consts

A tuple containing the literals used by the bytecode. Remember everything in a code object must be immutable, running diss and ssc on the code snippets a=(1,2,3) versus [1,2,3] and yet again versus a=(1,2,3,[4,5,6]) recommended to dig this field.

co_lnotab

A string encoding the mapping from bytecode offsets to line numbers. If you happen to really care how this is encoded you can either look at ./Python/compile.c or ./Lib/dis.py: findlinestarts.

co_flags

An integer encoding a number of flags regarding the way this code object was created (which says something about how it should be evaluated). The list of possible flags is listed in ./Include/code.h, as a small example I can give CO_NESTED, which marks a code object which was compiled from a lexically nested function. Flags also have an important role in the implementation of the __future__ mechanism, which is still unused in Python 3.1 at the time of this writing, as no "future syntax" exists in Python 3.1. However, even when thinking in Python 3.x terms co_flags is still important as it facilitates the migration from the 2.x branch. In 2.x, __future__ is used when enabling Python 3.x like behaviour (i.e., from __future__ import print_function in Python 2.7 will disable the print statement and add a print function to the builtins module, just like in Python 3.x). If we come across flags from now on (in future posts), I'll try to mention their relevance in the particular scenario.

co_zombieframe

This field of the PyCodeObject struct is not exposed in the Python object; it (optionally) points to a stack frame object. This can aid performance by maintaining an association between a code object and a stack frame object, so as to avoid reallocation of frames by recycling the frame object used for a code object. There's a detailed comment in ./Objects/frameobject.c explaining zombie frames and their reanimation, we may mention this issue again when we discuss stack frames.

The above codeobjects list is not exhaustive. More can be added based on need and usage. This completes the codeobjects explaination, next will be frameobjects.

Core of Python's Virtual Machine, the "actually do work function" `./Python/ceval.c: PyEval_EvalFrameEx`

Last hurdle on our way there is to understand the three significant stack data structures used for CPython's code evaluation: the call stack, the value stack and the block stack. All three stacks are tightly coupled with the frame object, which will also be discussed today.

In computer science, a call stack is a stack data structure that stores information about the active subroutines of a computer program… A call stack is composed of stack frames (…). These are machine dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return.

Since CPython implements a virtual machine, its call stack and stack frames are dependant on this virtual machine, not on the physical machine it's running on.

Python tends to do, this internal implementation detail is exposed to Python code, either via the C-API or pure Python, as frame objects (`./Include/frameobject.h: PyFrameObject`).

We know that code execution in CPython is really the evaluation (interpretation) of a code object, so every frame represents a currently-being-evaluated code object. We'll see (and already saw before) that frame objects are linked to one another, thus forming a call stack of frames. Finally, inside each frame object in the call stack there's a reference to two frame-specific stacks (not directly related to the call stack), they are the value stack and the block stack.

The value stack (you may know this term as an 'evaluation stack') is where manipulation of objects happens when object-manipulating opcodes are evaluated

We have seen the value stack before on various occasions, like in the introduction and during our discussion of namespaces.

Recalling an example we used before, `BINARY_SUBTRACT` is an opcode that effectively pops the two top objects in the value stack, performs `PyNumber_Subtract` on them and sets the new top of the value stack to the result.

Namespace related opcodes, like `LOAD_FAST` or `STORE_GLOBAL`, load values from a namespace to the stack or store values from the stack to a namespace. Each frame has a value stack of its own (this makes sense in several ways, possibly the most prominent is simplicity of implementation), we'll see later where in the frame object the value stack is stored.

Python has a notion called a code block, which we have discussed in the article about code objects and which is also explained here. Completely unrelatedly, Python also has a notion of compound statements, which are statements that contain other statements (the language reference defines compound statements here). Compound statements consist of one or more clauses, each made of a header and a suite. Even if the terminology wasn't known to you until now, I expect this is all instinctively clear to you if you have almost any Python experience: for, try and while are a few compound statements.

In various places throughout the code, a block (sometimes "frame block", sometimes "basic block") is used as a loose synonym for a clause or a suite, making it easier to confuse suites and clauses with what's actually a code block or vice versa.

Both the compilation code (./Python/compile.c) and the evaluation code (./Python/ceval.c) are aware of various suites and have (ill-named) data structures to deal with them; but since we're more interested in evaluation in this series, we won't discuss the compilation-related details much (or at all).

Whenever I'll think wording might get confusing, I'll mention the formal terms of clause or suite alongside whatever code term we're discussing. With all this terminology in mind we can look at what's contained in a frame object.

Looking at the declaration of `./Include/frameobject.h: PyFrameObject`, we find (comments were trimmed and edited for your viewing pleasure):

```
typedef struct _frame {
   PyObject_VAR_HEAD
   struct _frame *f_back;   /* previous frame, or NULL */
   PyCodeObject *f_code;    /* code segment */
   PyObject *f_builtins;    /* builtin symbol table */
   PyObject *f_globals;     /* global symbol table */
   PyObject *f_locals;      /* local symbol table */
   PyObject **f_valuestack; /* points after the last local */
   PyObject **f_stacktop;   /* current top of valuestack */
   PyObject *f_trace;       /* trace function */

   /* used for swapping generator exceptions */
   PyObject *f_exc_type, *f_exc_value, *f_exc_traceback;

   PyThreadState *f_tstate; /* call stack's thread state */
   int f_lasti;             /* last instruction if called */
   int f_lineno;            /* current line # (if tracing) */
   int f_iblock;            /* index in f_blockstack */

   /* for try and loop blocks */
   PyTryBlock f_blockstack[CO_MAXBLOCKS];

   /* dynamically: locals, free vars, cells and valuestack */
   PyObject *f_localsplus[1]; /* dynamic portion */
} PyFrameObject;
```

We see various fields used to store the state of this invocation of the code object as well as maintain the call stack's structure. Both in the C-API and in Python these fields are all prefixed by `f_`, though not all the fields of the C structure PyFrameObject are exposed in the pythonic representation.

We already mentioned the relation between frame and code objects, so the f_code field of every frame points to precisely one code object.

Insofar as structure goes, frames point backwards thus that they create a stack (f_back) as well as point "root-wards" in the interpreter state/thread state/call stack structure by pointing to their thread state (f_tstate), as explained here. Finally, since you always execute Python code in the context of three namespaces (as discussed there), frames have the f_builtins, f_globals and f_locals fields to point to these namespaces.

Before we dig into the other fields of a frame object, please notice frames are a variable size Python object (they are a PyObject_VAR_HEAD).

The reason is that when a frame object is created it should be dynamically allocated to be large enough to contain references (pointers, really) to the locals, cells and free variables used by its code object, as well as the value stack needed by the code objects 'deepest' branch.

Indeed, the last field of the frame object, f_localsplus (locals plus cells plus free variables plus value stack…) is a dynamic array where all these references are stored. `PyFrame_New` will show you exactly how the size of this array is computed.

`co_nlocals`, `co_cellvars`, `co_freevars` and `co_stacksize` - during evaluation, all these 'dead' parts of the inert code object come to 'life' in space allocated at the end of the frame. As we'll probably see in the next article, when the frame is evaluated, these references at the end of the frame will be used to get (or set) "fast" local variables, free variables and cell variables, as well as to the variables on the value stack ("fast" locals was explained when we discussed namespaces).

Looking back at the commented declaration above and given what I said here, I believe you should now understand `f_valuestack`, `f_stacktop` and `f_localsplus`.

As you can maybe imagine, compound statements sometimes require state to be evaluated. If we're in a loop, we need to know where to go in case of a break or a continue. If we're raising an exception, we need to know where is the innermost enclosing handler (the suite of the closest except header, in more formal terms).

This state is stored in `f_blockstack`, a fixed size stack of `PyTryBlock structures` which keeps the current compound statement state for us (`PyTryBlock` is not just for try blocks; it has a `b_type` field to let it handle various types of compound statements' suites). `f_iblock` is an offset to the last allocated PyTryBlock in the stack. If we need to bail out of the current "block" (that is, the current clause), we can pop the block stack and find the new offset in the bytecode from which we should resume evaluation in the popped `PyTryBlock` (look at its b_handler and b_level fields).

A somewhat special case is a raised exception which exhausts the block stack without being caught, as you can imagine, in that case a handler will be sought in the block stack of the previous frames on the call stack.

All this should easily click into place now if you read three code snippets. First, look at this disassembly of a for statement (this would look strikingly similar for a try statement):

```
>>> def f():
...     for c in 'string':
...             my_global_list.append(c)
...
>>> diss(f)
 2          0 SETUP_LOOP              27 (to 30)
            3 LOAD_CONST               1 ('string')
            6 GET_ITER
      >>    7 FOR_ITER                19 (to 29)
           10 STORE_FAST               0 (c)

 3         13 LOAD_GLOBAL              0 (my_global_list)
           16 LOAD_ATTR                1 (append)
           19 LOAD_FAST                0 (c)
           22 CALL_FUNCTION            1
           25 POP_TOP
           26 JUMP_ABSOLUTE            7
      >>   29 POP_BLOCK
      >>   30 LOAD_CONST               0 (None)
           33 RETURN_VALUE
>>>
```

Look at how the opcodes `SETUP_LOOP` and `POP_BLOCK` are implemented in `./Python/ceval.c`.

Notice that `SETUP_LOOP` and `SETUP_EXCEPT` or `SETUP_FINALLY` are rather similar, they all push a block matching the relevant suite unto the block stack, and they all utilize the same `POP_BLOCK`:

```
TARGET_WITH_IMPL(SETUP_LOOP, _setup_finally)
TARGET_WITH_IMPL(SETUP_EXCEPT, _setup_finally)
TARGET(SETUP_FINALLY)
_setup_finally:
    PyFrame_BlockSetup(f, opcode, INSTR_OFFSET() + oparg,
               STACK_LEVEL());
    DISPATCH();

TARGET(POP_BLOCK)
    {
        PyTryBlock *b = PyFrame_BlockPop(f);
        UNWIND_BLOCK(b);
    }
    DISPATCH();
```

Finally, look at the actual implementation of `./Object/frameobject.c: PyFrame_BlockSetup` and `./Object/frameobject.c`:

```
PyFrame_BlockPop:

void
PyFrame_BlockSetup(PyFrameObject *f, int type, int handler, int level)
{
   PyTryBlock *b;
   if (f->f_iblock >= CO_MAXBLOCKS)
       Py_FatalError("XXX block stack overflow");
   b = &f->f_blockstack[f->f_iblock++];
   b->b_type = type;
   b->b_level = level;
   b->b_handler = handler;
}

PyTryBlock *
PyFrame_BlockPop(PyFrameObject *f)
{
   PyTryBlock *b;
   if (f->f_iblock <= 0)
       Py_FatalError("XXX block stack underflow");
   b = &f->f_blockstack[--f->f_iblock];
   return b;
}
```

If you keep the terminology straight, `f_blockstack` turns out to be rather simple. We're left with the rather esoteric fields, some simpler, some a bit more arcane. In the 'simpler' range we have f_lasti, an integer offset into the bytecode of the last instructions executed (initialized to -1, i.e., we didn't execute any instruction yet).

This index lets us iterate over the opcodes in the bytecode stream. Heading towards the 'more arcane' area we see f_trace and f_lineno. f_trace is a pointer to a tracing function (see sys.settrace; think implementation of a tracer or a debugger). `f_lineno` contains the line number of the line which caused the generation of the current opcode; it is valid only when tracing (otherwise use `PyCode_Addr2Line`).

Last but not least, we have three exception fields (f_exc_type, f_exc_value and f_exc_traceback), which are rather particular to generators so we'll discuss them when we discuss that beast (there's a longer comment about these fields in ./Include/frameobject.h if you're curious right now). On a parting note, we can mention when frames are created. This happens in ./Objects/frameobject.c: PyFrame_New, usually called from ./Python/ceval.c: PyEval_EvalCodeEx (and ./Python/ceval.c: fast_function, a specialized optimization of PyEval_EvalCodeEx).

Frame creation occurs whenever a code object should be evaluated, which is to say when a function is called, when a module is imported (the module's top-level code is executed), whenever a class is defined, for every discrete command entered in the interactive interpreter, when the builtins eval or exec are used and when the -c switch is used (I didn't absolutely verify this is a 100% exhaustive list, but it think it's rather complete).

Looking at the list in the previous paragraph, you probably realized frames are created very often, so two optimizations are implemented to make frame creation fast: first, code objects have a field (co_zombieframe) which allows them to remain associated with a ‘zombie’ (dead, unused) frame object even when they’re not evaluated. If a code object was already evaluated once, chances are it will have a zombie frame ready to be reanimated by PyFrame_New and returned instead of a newly allocated frame (trading some memory to reduce the number of allocations).

Second, allocated and entirely unused stack frames are kept in a special free-list (./Objects/frameobject.c: free_list), frames from this list will be used if possible, instead of actually allocating a brand new frame. This is all kindly commented in ./Objects/frameobject.c.

./Python/ceval.c: PyEval_EvalFrameEx is important function in the Python interpreter.

Well, as I said, this switch can be found in the rather lengthy file ceval.c, in the rather lengthy function PyEval_EvalFrameEx, which takes more than half the file’s lines (it’s roughly 2,250 lines, the file is about 4,400).

PyEval_EvalFrameEx implements CPython’s evaluation loop, which is to say that it’s a function that takes a frame object and iterates over each of the opcodes in its associated code object, evaluating (interpreting, executing) each opcode within the context of the given frame (this context is chiefly the associated namespaces and interpreter/thread states). There’s more to ceval.c than PyEval_EvalFrameEx, and we may discuss some of the other bits later in this post (or perhaps a follow-up post), but PyEval_EvalFrameEx is obviously the most important part of it.

Having described the evaluation loop in the previous paragraph, let’s see what it looks like in C (edited):

```
PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    /* variable declaration and initialization stuff */
    for (;;) {
        /* do periodic housekeeping once in a few opcodes */
        opcode = NEXTOP();
        if (HAS_ARG(opcode)) oparg = NEXTARG();
        switch (opcode) {
            case NOP:
                goto fast_next_opcode;
            /* lots of more complex opcode implementations */
            default:
                /* become rather unhappy */
        }
        /* handle exceptions or runtime errors, if any */
    }
    /* we are finished, pop the frame stack */
    tstate->frame = f->f_back;
    return retval;
}
```

As you can see, iteration over opcodes is infinite (forever: fetch next opcode, do stuff), breaking out of the loop must be done explicitly.

CPython (reasonably) assumes that evaluated bytecode is correct in the sense that it terminates itself by raising an exception, returning a value, etc. Indeed, if you were to synthesize a code object without a RETURN_VALUE at its end and execute it (exercise to reader: how?1), you're likely to execute rubbish, reach the default handler (raises a SystemError) or maybe even segfault the interpreter (I didn't check this thoroughly, but it looks plausible).

In order for you to be able to get a feel for what more serious opcode implementations look like, here's the (edited) implementation of three more opcodes, illustrating a few more principles:

```
case BINARY_SUBTRACT:
    w = *--stack_pointer; /* value stack POP */
    v = stack_pointer[-1];
    x = PyNumber_Subtract(v, w);
    stack_pointer[-1] = x; /* value stack SET_TOP */
    if (x != NULL) continue;
    break;
case LOAD_CONST:
    x = PyTuple_GetItem(f->f_code->co_consts, oparg);
    *stack_pointer++ = x; /* value stack PUSH */
    goto fast_next_opcode;
case SETUP_LOOP:
case SETUP_EXCEPT:
case SETUP_FINALLY:
    PyFrame_BlockSetup(f, opcode, INSTR_OFFSET() + oparg,
               STACK_LEVEL());
    continue;
```

We see several things. First, we see a typical value manipulation opcode, BINARY_SUBTRACT. This opcode (and many others) works with values on the value stack as well as with a few temporary variables, using CPython's C-API abstract object layer (in our case, a function from the number-like object abstraction) to replace the two top values on the value stack with the single value resulting from subtraction.

As you can see, a small set of temporary variables, such as v, w and x are used (and reused, and reused...) as the registers of the CPython VM.

The variable stack_pointer represents the current bottom of the stack (the next free pointer in the stack). This variable is initialized at the beginning of the function like so: stack_pointer = f->f_stacktop;

In essence, together with the room reserved in the frame object for that purpose, the value stack is this pointer. To make things simpler and more readable, the real (unedited by me) code of ceval.c defines several value stack manipulation/observation macros, like PUSH, TOP or EMPTY.

Next, we see a very simple opcode that loads values from somewhere into the valuestack. I chose to quote LOAD_CONST because it's very brief and simple, although it's not really a namespace related opcode.

"Real" namespace opcodes load values into the value stack from a namespace and store values from the value stack into a namespace; LOAD_CONST loads constants, but doesn't fetch them from a namespace and has no STORE_CONST counterpart (we explored all this at length in the article about namespaces).

The final opcode I chose to show is actually the single implementation of several different control-flow related opcodes (SETUP_LOOP, SETUP_EXCEPT and SETUP_FINALLY), which offload all details of their implementation to the block stack manipulation function PyFrame_BlockSetup; we discussed the block stack in our discussion of interpreter stacks.

Something we can observe looking at these implementations is that different opcodes exit the switch statement differently. Some simply break, and let the code after the switch resume.

Some use continue to start the for loop from the beginning. Some goto various labels in the function. Each exit has different semantic meaning.

If you break out of the switch (the 'normal' route), various checks will be made to see if some special behaviour should be performed – maybe a code block has ended, maybe an exception was raised, maybe we're ready to return a value. Continuing the loop or going to a label lets certain opcodes take various shortcuts; no use checking for an exception after a NOP or a LOAD_CONST, for instance.

If you look at the code itself, you will see that none of the case expressions for the big switch are really there. The code for the NOP opcode is actually (remember this series is about Python 3.x unless noted otherwise, so this snippet is from Python 3.1.2):

```
TARGET(NOP)
    FAST_DISPATCH();
```

TARGET? FAST_DISPATCH? What are these? Let me explain. Things may become clearer if we'd look for a moment at the implementation of the NOP opcode in ceval.c of Python 2.x. Over there the code for NOP looks more like the samples I've shown you so far, and it actually seems to me that the code of ceval.c gets simpler and simpler as we look backwards at older revisions of it.

The reason is that although I think PyEval_EvalFrameEx was originally written as a really exceptionally straightforward piece of code, over the years some necessary complexity crept into it as various optimizations and improvements were implemented (I'll collectively call them 'additions' from now on, for lack of a better term).

To further complicate matters, many of these additions are compiled conditionally with preprocessor directives, so several things are implemented in more than one way in the same source file. I can understand trading simplicity to optimize a tight loop which is used very often, and the evaluation loop is probably one of the more used loops in CPython (and probably as tight as its contributors could make it). So while this is all very warranted, it doesn't help the readability of the code. Anyway, I'd like to enumerate these additions here explicitly (some in more depth than others); this should aid future discussion of ceval.c, as well as prevent me from feeling like I'm hiding too many important things with my free spirited editing of quoted code.

Fortunately, most if not all these additions are very well commented -actually, some of the explanations below will be just summaries or even taken verbatim from these comments, as I believe that they're accurate (eek!). So, as you read `PyEval_EvalFrameEx` (and indeed ceval.c in general), you're likely to run into any of these

"Threaded Code" (Computed-GOTOs)

Let's start with the addition that gave us TARGET, FAST_DISPATCH and a few other macros. The evaluation loop uses a "switch" statement, which decent compilers optimize as a single indirect branch instruction with a lookup table of addresses. Alas, since we're switching over rapidly changing opcodes (it's uncommon to have the same opcode repeat), this would have an adverse effect on the success rate of CPU branch prediction.

Fortunately gcc supports the use of C-goto labels as values, which you can generally pass around and place in an array (restrictions apply!). Using an array of adresses in memory obtained from labels, as you can see in ./Python/opcode_targets.h, we create an explicit jump table and place an explicit indirect jump instruction at the end of each opcode. This improves the success rate of CPU prediction and can yield as much as 20% boost in performance.

Thus, for example, the NOP opcode is implemented in the code like so:

```
TARGET(NOP)
    FAST_DISPATCH();
```

In the simpler scenario, this would expand to a plain case statement and a goto, like so:

```
case NOP:
    goto fast_next_opcode;
```

But when threaded code is in use, that snippet would expand to (I highlighted the lines where we actually move on to the next opcode, using the dispatch table of label-values):

```
TARGET_NOP:
    opcode = NOP;
    if (HAS_ARG(NOP))
        oparg = NEXTARG();
case NOP:
    {
        if (!_Py_TracingPossible) {
            f->f_lasti = INSTR_OFFSET();
            goto *opcode_targets[*next_instr++];
        }
        goto fast_next_opcode;
    }
```

Same behaviour, somewhat more complicated implementation, up to 20% faster Python. Nifty.

Opcode Prediction

Some opcodes tend to come in pairs. For example, COMPARE_OP is often followed by JUMP_IF_FALSE or JUMP_IF_TRUE, themselves often followed by a POP_TOP.

What's more, there are situations where you can determine that a particular next-opcode can be run immediately after the execution of the current opcode, without going through the 'outer' (and expensive) parts of the evaluation loop.

`PREDICT` (and a few others) are a set of macros that explicitly peek at the next opcode and jump to it if possible, shortcutting most of the loop in this fashion (i.e., `if (*next_instr == op) goto PRED_##op`).

Note that there is no relation to real hardware here, these are simply hardcoded conditional jumps, not an exploitation of some mechanism in the underlying CPU (in particular, it has nothing to do with "Threaded Code" described above).

Low Level Tracing

An addition primarily geared towards those developing CPython (or suffering from a horrible, horrible bug), Low Level Tracing is controlled by the LLTRACE preprocessor name, which is enabled by default on debug builds of CPython (see –with-pydebug). As explained in ./Misc/SpecialBuilds.txt: when this feature is compiled-in, PyEval_EvalFrameEx checks the frame's global namespace for the variable __lltrace__.

If such a variable is found, mounds of information about what the interpreter is doing are sprayed to stdout, such as every opcode and opcode argument and values pushed onto and popped off the value stack. Not useful very often, but very useful when needed.

This is the what the low level trace output looks like (slightly edited):

```
>>> def f():
...     global a
...     return a - 5
...
>>> dis(f)
  3           0 LOAD_GLOBAL              0 (a)
              3 LOAD_CONST               1 (5)
              6 BINARY_SUBTRACT
              7 RETURN_VALUE
>>> exec(f.__code__, {'__lltrace__': 'foo', 'a': 10})
0: 116, 0
push 10
3: 100, 1
push 5
6: 24
pop 5
7: 83
pop 5
# trace of the end of exec() removed
>>>
```

As you can guess, you're seeing a real-time disassembly of what's going through the VM as well as stack operations. For example, the first line says: line 0, do opcode 116

(LOAD_GLOBAL) with the operand 0 (expands to the global variable a), and so on, and so forth. This is a bit like (well, little more than) adding a bunch of printf calls to the heart of VM.

Advanced Profiling

Under this heading I'd like to briefly discuss several profiling related additions. The first relies on the fact that some processors (notably Pentium descendants and at least some PowerPCs) have built-in wall time measurement capabilities which are cheap and precise (correct me if I'm wrong).

As an aid in the development of a high-performance CPython implementation, Python 2.4's ceval.c was instrumented with the ability to collect per-opcode profiling statistics using these counters.

This instrumentation is controlled by the somewhat misnamed -with-tsc configuration flag (TSC is an Intel Pentium specific name, and this feature is more general than that). Calling sys.settscdump(True) on an instrumented interpreter will cause the function ./Python/ceval.c: dump_tsc to print these statistics every time the evaluation loop loops.

The second advanced profiling feature is Dynamic Execution Profiling. This is only available if Python was built with the DYNAMIC_EXECUTION_PROFILE preprocessor name.

As ./Tools/scripts/analyze_dxp.py says, [this] will tell you which opcodes have been executed most frequently in the current process, and, if Python was also built with -DDXPAIRS, will tell you which instruction _pairs_ were executed most frequently, which may help in choosing new instructions.

One last thing to add here is that enabling Dynamic Execution Profiling implicitly disables the "Threaded Code" addition.

The third and last addition in this category is function call profiling, controlled by the preprocessor name CALL_PROFILE. Quoting ./Misc/SpecialBuilds.txt again: When this name is defined, the ceval mainloop and helper functions count the number of function calls made. It keeps detailed statistics about what kind of object was called and whether the call hit any of the special fast paths in the code.

Two preprocessor names, USE_STACKCHECK and CHECKEXC include extra assertions. Testing an interpreter with these enabled may catch a subtle bug or regression, but they are usually disabled as they're too expensive.

That's the end of how eval loop operates.