Python VM Internals Tutorials Documentation

Release 1.0.0

Smruthi Manjunath

January 22, 2017

Contents

This is a tutorial giving an overview of Python VM including CPython, bytecodes and their meaning, .pyc files and differences between various versions of Python.

CHAPTER 1

CPython Overview

Cpython is a python bytecode compiler written in C. For every bytecode in python, there are corresponding C functions that implement their functionalities. The main directories of CPython are `Include/`, where the header files such as object.h, python.h, etc are present, the `Object/` directory that contains the implementations such as string, int, floats, etc., the `Python/` directory that contains the parser importer, `Lib/` that contains the library modules and the `Module/` that contains the C extension modules.

CHAPTER 2

Structure of Python Objects

In python, everything is considered as an object, from integers, to strings, to functions and classes. So, it is very important to understand their structure.

2.1 PyObject

The following is the structure of a python object defined in ./Include/object.h:PyObject

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

This is the basic structure and it contains 2 fields `ob_refcnt` and `ob_type` that are mandatory for all the objects. All objects have references and the number of references to an object are stored in `ob_refcnt`. If we execute this statement on the python interpreter `x=object()`. The address of the memory location allocated to the object on the heap is stored in `x`. The reference count of object is incremented by 1 and is stored in `ob_refcnt` variable of the object. This will be used for garbage collection. Object type is of type `_typeobject`, which is again a structure that gives more details of the object such as, various functions and protocols implemented and supported by the object.

2.2 PyTypeObject

The structure `_typeobject` is defined below:

```
typedef struct _typeobject {
        PyObject_VAR_HEAD
        char *tp_name;
        int tp_basicsize, tp_itemsize;
        destructor tp_dealloc;
        printfunc tp_print;
        getattrfunc tp_getattr;
        setattrfunc tp_setattr;
        cmpfunc tp_compare;
        PyNumberMethods *tp_as_number;
        PySequenceMethods *tp_as_sequence;
        PyMappingMethods *tp_as_mapping;
        .
        .
```

```
        .
}PyTypeObject;
```

Python_VAR_HEAD, corresponds to basic structure of the object and an additional field ob_size that is 0 for statically allocated type objects and has a value for dynamically allocated type objects. The next field is the tp_name, that gives the type represented by a String. There are pointers to destructors,print, getters and setters functions. there are also, other fields such as tp_basicsize and tp_itemsize, where tp_basicsize represents the size of fixed length instances and tp_itemsize represents the size variable length instances. The tp_as_number, tp_as_sequence, tp_as_mapping represents the protocols that are implemented by the object. This includes the functions object supports. Each of the protocols is a pointer to another structure that has the functions that are implemented by the object type. There are other fields that an object type can contain.

CHAPTER 3

Generated Python Bytecodes for a simple python code

Below is a simple python program and the bytecodes generated for the same:

```
#usr/bin python
# encoding: utf-8
import sys
def foo(a,b):
        return a-b
print(foo(17,25))
```

Bytecode obtained

```
3           0 LOAD_CONST               0 (-1)
            3 LOAD_CONST               1 (None)
            6 IMPORT_NAME              0 (sys)
            9 STORE_NAME               0 (sys)

4          12 LOAD_CONST               2 (<code object foo at 0x7fe405b04a30, file "bytecc.py", line
           15 MAKE_FUNCTION            0
           18 STORE_NAME               1 (foo)

6          21 LOAD_NAME                1 (foo)
           24 LOAD_CONST               3 (17)
           27 LOAD_CONST               4 (25)
           30 CALL_FUNCTION            2
           33 PRINT_ITEM
           34 PRINT_NEWLINE
           35 LOAD_CONST               1 (None)
           38 RETURN_VALUE
```

The first bytecode is `LOAD_CONST` that pushes a constant on top of stack. The constant it pushes is indexed from the co_consts array. The next line is the `IMPORT_NAME` that imports the module mentioned in the `co_const[]` array. Import calls `__import__()` that takes 5 parameters, namely name(sys), local, global, from list (list of modules imported from a class eg:from a import a,b,c) and level (absolute or relative). It pushes the module object on top of the stack, `STORE_NAME`, pops it off the top of the stack and associates the module object with name sys and stores it in the `co_names[]` array in index 0.

The next set of bytecodes are for the function definition. `LOAD_CONST` is used to push the co_names[2] on top of the stack. So, it looks up co_names[] and finds foo code object in that location. The `MAKE_FUNCTION` pops the top of the stack and converts the code into an actual callable and pushes it on top of the stack. `STORE_NAME` pops it from top of the stack into the names array corresponding to foo.

The last set of bytecodes that are generated represent the print statement and the nested function call. First the callable foo is pushed on top of the stack, then the positional arguments are pushed on top of the stack the `CALL_FUNCTION`,

pops top 2 elements from the stack and puts it into a tuple and then again pops the top of the stack which now contains the callable foo, the tuple is passed to the callable, it performs the function and pushes the result value on top of the stack. This is then popped by the print bytecode, that puts it on to the stdout. This is followed by a print newline, if the print statement is not terminated by a comma. The `RETURN_VALUE` bytecode pops the top of the stack and returns it to the caller. Lastly, the bytecode generated for the return(a-b) statement is

```
0 LOAD_FAST     0 (a)
3 LOAD_FAST     1 (b)
6 BINARY_SUBTRACT
7 RETURN_VALUE
```

The parameters a and b are stored in a fast array that is associated with a method where locals and arguments are stored. The `LOAD_FAST` pushes arguments a and b on top of the stack. `BINARY_SUBTRACT` pops the top two elements of the stack, performs the subtraction and pushes the result on top of the stack which is returned to the caller by `RETURN_VALUE` by popping the stack.

`BINARY_SUBTRACT` is implemented as follows:

```
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;
```

The values a and b that were previously pushed on top of the stack are now popped and put into registers. `u`, `v` and `x` are registers. `PyNumber_Subtract()` is called and the value returned by it is put in `x` . The references for object in `v` and `w` are decremented and the returned value is pushed on top of the stack. If `x` is null that is if there was an exception or any error then it breaks out else `dispatch()` is called to execute the next instruction.

The `PyNumber_Subtract()` is used to find the product of the two numbers. Since, the type of the object on which it is operating is unknown, it should be determined. This is achieved as follows, it calls another C function `binary_op1()`, which receives the *PyObject pointer, which it dereferences to find the type of the object (u->ob_type). Then it checks if tp_as_number protocol is implemented by it, (u->ob_type->tp_as_number) and then checks if the subtraction number is supported. If it is, it performs the subtraction and returns the value else it returns PyNot_Implemented.

CHAPTER 4

.pyc files

The .pyc files are compiled bytecodes, their structure is very simple, it has a 4 byte magic tag and then a 4 byte modification timestamp that helps in determining whether or not the compiled bytecodes can be used for interpreting, that is, if the modification time is the same as the last time the py file was compiled, it will directly interpret the bytecodes. The rest of the file is a serialized byte stream whose format is dictated by the magic tag.

4.1 Advantages of shipping pyc files over py files:

- They are compact so easier to transmit over the network.
- Load faster as only interpretation of the bytecodes is required, compilation into bytecodes can be skipped.
- Provide some security, as you are not sending the source code, but it is easy to decode and get the bytecodes and know what the program is doing.

4.2 Disadvanatge of pyc files

- Not compatible across python releases, as the magic tag changes, the pyc file format is different and hence not recognized by the interpreter.

CHAPTER 5

Differences between the various versions of python are:

There were a lot of changes that were made in each of the python releases, these are a few.

5.1 Differences between python 2.6 and 2.7:

- The garbage collector has been optimized for a common usage pattern where, the objects are allocated and not deallocated for a long time. The garbage collector has 3 generations, young, middle and old. The young generation is collected for every 700 allocations, the middle generation is collected for every 10 collections of the middle generation but the old generation was collected for every 10 collections of the middle generation previously. This has been modified to collecting the old generation not only for every 10 collections of the middle generation but also, when the middle generation reaches atleast 10% of the objects in the old generation. This does not happen that often as objects allocated are not deallocated that frequently. So, it is optimized for full collection.
- The garbage collector tries to avoid tracking simple containers such as integers and strings which can't be part of a cycle. In Python 2.7, this is now true for tuples and dicts containing atomic types (such as ints, strings, etc.). So, they will not be tracked by the garbage collector.
- Long integers are now stored internally either in base 2**15 or in base 2**30, the base being determined at build time. Previously, they were always stored in base 2**15.
- The division algorithm for long integers has been made faster by tightening the inner loop, doing shifts instead of multiplications, and fixing an unnecessary extra iteration and the string functions such as split(), replace() methods have been optimized.

5.2 Difference between python 2.5 and 2.6:

- Type objects now have a cache of methods that can reduce the work required to find the correct method implementation for a particular class; once cached, the interpreter doesn't need to traverse base classes to figure out the right method to call
- Function calls that use keyword arguments are significantly faster by doing a quick pointer comparison, usually saving the time of a full string comparisons. Since, parameter names are stored as interned strings.
- To reduce memory usage, the garbage collector will now clear internal free lists when garbage-collecting the highest generation of objects.

CHAPTER 6

References:

- Python Innards : http://www.cyberhades.com/2011/11/16/interesante-listado-de-enlaces-sobre-python/
- Stepping through CPython by Larry Hastings, PyCon 2012
- http://docs.python.org/2/c-api/typeobj.html
- http://docs.python.org/2/whatsnew/2.7.html
- http://docs.python.org/2/whatsnew/2.6.html

Contents:

CHAPTER 7

Indices and tables

- genindex
- modindex
- search