CPython$_{internals}$

Oct 21, 2019

Contents:

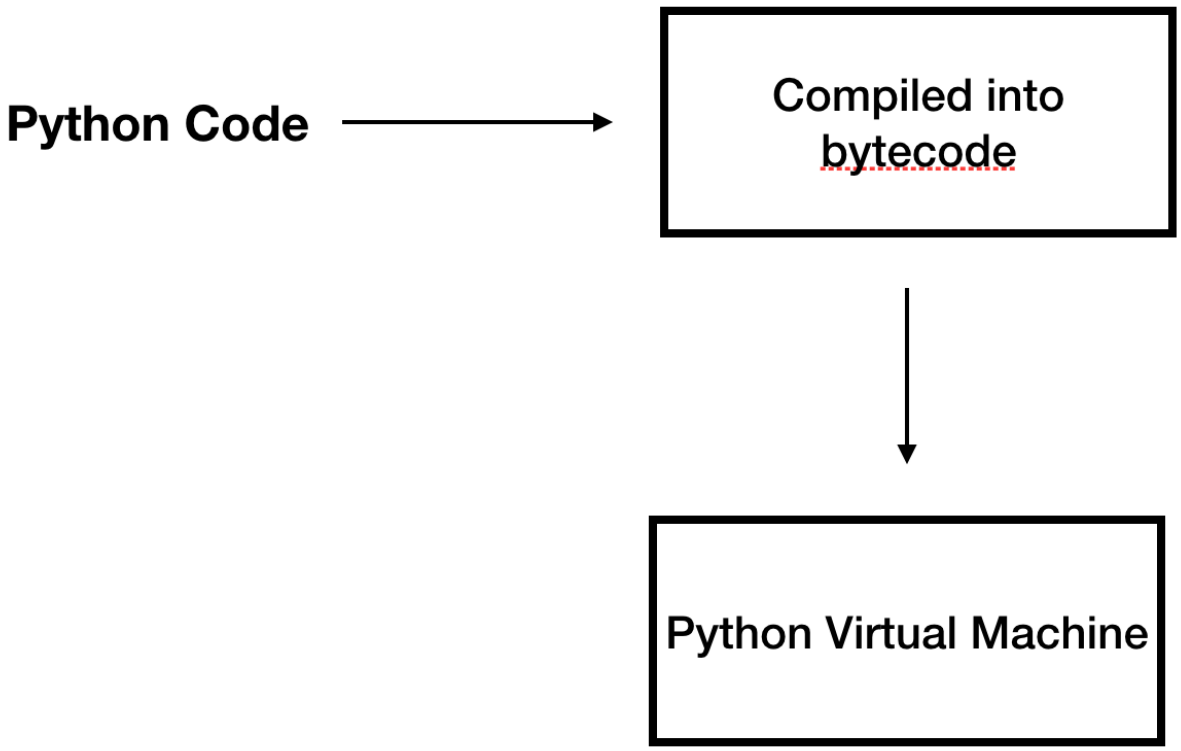
CHAPTER 1

Brief_introduction

Three steps to start CPython:

1. Initialization: data structure, memory.
2. Compiling: parse tree, ast, symbol tables, code object.
3. Interpreting: execution of the generated code objects

1.1 Project directory overview



- Doc: The manual
- Grammar: Where Grammer is defined
- Include: The C headers
- Lib: Python portion of the python library
- Modules: C portion of the python library
- Objects: The builtin object(string, list, bool, tuple)
- Parser: Grammer, lexer, parser, compiler
- Programs: The executable python program
- Python: The virtual machine

Python includes a compiler, interpreter. However, the compilation doesn't do much work. After compilation, the source code would turn into the bytecode, which is the code object.

Code object Compiled by CPython compiler can not be directly executed by the computer, it needs to be executed by the virtual machine, which is, interpreter.

Also, the virtual machine is running on a stack, and each frame has its own data stack. If not, we don't have the generator feature.

The reason that Python is called a dynamic language is that many things are done when the program is running, such as type checking.

```
def add(a, b):
    return a+b
add(1,2)
add("hello","world")
```

```
>>> 3
>>> helloworld
```

CHAPTER 2

Interpreter_vs_Compiler

The main difference between the compiler and the interpreter

- The compiler translates the whole program, but the interpreter translates the program line by line.
- The compiler is relatively faster than the interpreter because the interpreter executes on a non-native virtual environment.
- To store object code, the compiler requires more memory than the interpreter.
- The compiler will display all errors at the same time, while the interpreter will display the error of each error line by line.

2.1 Compiler

In short, the compiler compiles the code into a machine code that can be executed directly by the computer. It is usually called executable file, and the process will also do the type check, so no source code is needed when executing. Therefore, the job of the compiler is to translate, translate into a language that the computer can understand

2.2 Interpreter

It reads the program first, translates it into the code corresponding to the virtual machine, known as bytecode, so we do not know what the type is, only when running the program to know, this also, lead to the so-called dynamic type checking, further called dynamic language.

2.3 Commonality

Both the compiler and the interpreter will convert the original file into tokens, which will generate AST, and will also generate the intermediate code, but the compiler produces the **machine code**, the interpreter produces the instruction set is to be executed for the virtual machine.

Interpreter	Compiler	
Line by line when executing	The whole file scanning	
Overall time is longer	It takes a lot of time to analyze the source code, but the overall execution time is relatively fast.	
No object code is generated, so it is memory efficient	Generate object code that needs further links, so more memory is needed	

2.4 Example

- Java is compiled into bytecode and then executed by the JVM.
- C language is compiled into object code, and then becomes the executable file after the linker
- Python is first converted to the bytecode and then executed via ceval.c. The interpreter directly executes the translated instruction set.

CHAPTER 3

Compiling

1. Python code -> Parse tree
2. Parse tree -> AST
3. Symbol table generated
4. Code object generated
5. Flow control graph generated
6. Code object optimization (Peephole optimization)
7. Bytecode generated

3.1 1. Lexer

Take the source code into each word.

- Parser/tokenizer.c -> PyTokenizer_FromString
- Parser/parsetok.c -> parsetok
- Lib/tokenize.py

3.1.1 Tokenizing

The token is the name of some sort of symbol

For example:

```
a = 4
if (a <= 3):
    print("hello")
```

so it would turn into a list like below:

- Name: a
- EQUAL: =
- NUMBER: 4
- IF: if
- LPAREN: (
- etc
- tokenize.py
- token.py

```
python3 -m tokenize test.py
```

```
0,0-0,0:            ENCODING       'utf-8'
1,0-1,1:            NAME           'x'
1,2-1,3:            OP             '='
1,4-1,5:            NUMBER         '1'
1,5-1,6:            OP             '+'
1,6-1,7:            NUMBER         '1'
1,7-1,8:            NEWLINE        '\n'
2,0-2,1:            NAME           'y'
2,2-2,3:            OP             '='
2,4-2,5:            NAME           'x'
2,5-2,6:            OP             '+'
2,6-2,7:            NUMBER         '2'
2,7-2,8:            NEWLINE        '\n'
3,0-3,5:            NAME           'print'
3,5-3,6:            OP             '('
3,6-3,7:            NAME           'y'
3,7-3,8:            OP             ')'
3,8-3,9:            NEWLINE        '\n'
4,0-4,0:            ENDMARKER      ''
```

3.2 2. Parsing

The parser does not know what the source file means for, instead, it just knows the token generated by the lexer, and the token object would use function `next()` to give a token to the parser one by one.

- Python/pythonrun.c -> PyParser_ASTFromStringObject

```
import parser
code = "x = 2 + 2"
st = parser.suite(code)
```

```
>>> print(parser.st2list(st))
[257, [269, [270, [271, [272, [274, [305, [309, [310, [311, [312, [315, [316, [317,␣
↪[318, [319, [320, [321, [322, [323, [324, [1, 'x']]]]]]]]]]]]]]]]]], [22, '='], [274,
↪ [305, [309, [310, [311, [312, [315, [316, [317, [318, [319, [320, [321, [322, [323,
↪ [324, [2, '2']]]]]]], [14, '+'], [320, [321, [322, [323, [324, [2, '2
↪']]]]]]]]]]]]]]]]]], [4, '']]], [4, ''], [0, '']]
```

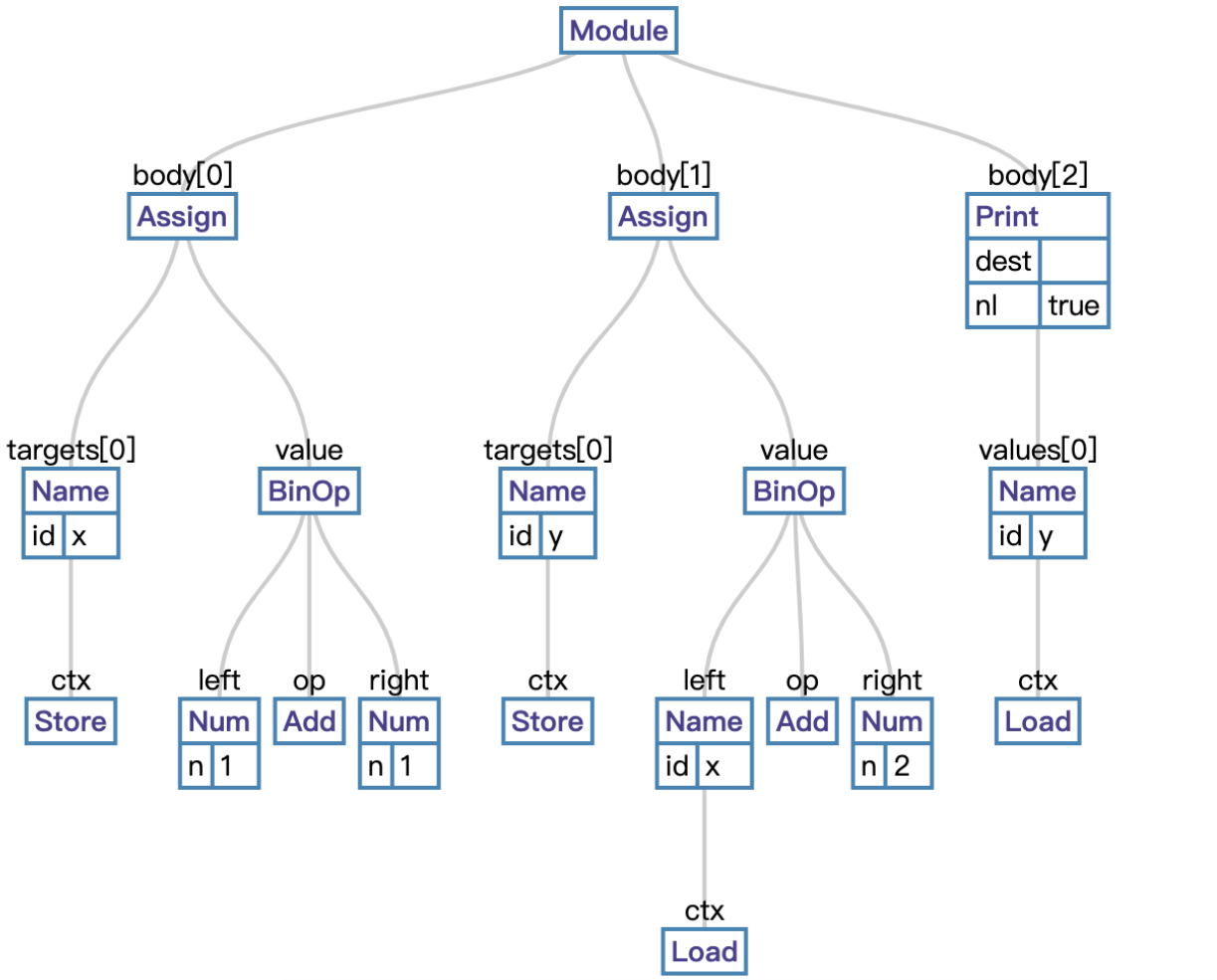
LL_parser Full Grammar specification: https://docs.python.org/3/reference/grammar.html

3.3 3. AST

```
import dis
import ast
tree = ast.parse("x=2+2")
print(type(ast.dump(tree)))
```

AST example:

```
x= 1 + 1
y= x + 2
print(y)
```



Generated by Python AST Visualizer: https://vpyast.appspot.com/

3.4 4. Compiler

Python/compile.c

```
import dis
import ast
tree = ast.parse("x=2+2")
code_obejct = compile(tree,'test.py',mode='exec')
dis.dis(code_obejct)
```

```
c = compile(open('test.py').read(), 'test.py', 'exec')
```

CHAPTER 4

Bytecode

Bytecode definition: https://docs.python.org/3/library/dis.html

stack based

`dis` module: https://docs.python.org/3/library/dis.html

`dis` module could show us the disassembling bytecode from Python code, and the bytecode will map to the opcode.h

You can write some code like below:

```
x = 1+1
y = x+2
print(y)
```

save it to **test.py** and then back to the command line typing `python3 -m dis test.py`

```
1           0 LOAD_CONST               0 (2)
            2 STORE_NAME               0 (x)

2           4 LOAD_NAME                0 (x)
            6 LOAD_CONST               0 (2)
            8 BINARY_ADD
           10 STORE_NAME               1 (y)

3          12 LOAD_NAME                2 (print)
           14 LOAD_NAME                1 (y)
           16 CALL_FUNCTION            1
           18 POP_TOP
           20 LOAD_CONST               1 (None)
           22 RETURN_VALUE
```

4.1 pyc

pyc is sort of bytecode that putting on disk,and a sequence of instructions for machine to run.

When Python program runs, the bytecode is temporarily stored in the PyCodeObject in the memory. Once the Python program ends, the Python interpreter writes the PyCodeObject to the `.pyc` file.

We can also execute directly through pyc file: `python xxx.pyc`

When the Python program is executed for the second time, it will first look for the .pyc file in the current directory. If it is found, it will be loaded directly. If it is not found, repeat the above process.

If Python finds that Python's source code has been modified, it will check the timestamp. In short, it is to determine the update time of the two files before deciding whether to compile or load directly.

marshal.c : Write Python objects to files and read them back

Each data type is mapping to a function to write them into pyc file like `w_string`, `w_object`. When these python data types are written to the pyc file, all data structures will disappear, so you must rely on such as function `W_TYPE (TYPE_LIST, p);` to write the type into file to recover.

In any case, there are only two forms for making pyc file, one for numeric and one for string.

A python source file will eventually become a PyCodeObject, and the inside classes or functions will be compiled to a code block, which will also be compiled into a PyCodeObject and stored in the first PyCodeObject->co_const.

See also:

Scott Sanderson, Joe Jevnik - Playing with Python Bytecode - PyCon 2016

CHAPTER 5

Interpreter

CPython's Virtual Machine

Code evaluator : ceval.c

The so-called Interpreter is just a huge loop.

- opcode.h
- code.h

```
$ python test.py
```

Python needs to turn test.py into something that can be executed, so what is that? That is bytecode.

The execution of Python is that there are multiple cases in a loop, which corresponds to different instructions.

CHAPTER 6

Python_object

PyObject Definitions

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

Everything in Python is an object. Objects in C are represented by contiguous memories and are using a large number of macro definitions. Because there is no inheritance in C, so put the macro again in the place where you need to use inheritance.

Using `dir()` function to inspect what inside is.

```
>>> dir(123)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
↪'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv_
↪_', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash_
↪_', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
↪ '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__
↪or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '_
↪_reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
↪'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__
↪rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__
↪subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
↪'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

```
>>> dir("Python")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__
↪', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__
↪gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
↪ '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
↪ '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__
↪subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
↪'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
↪'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
↪'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
↪'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
↪'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title
↪', 'translate', 'upper', 'zfill']
```

(continues on next page)

(continued from previous page)

Return the memory of an object

```
>>> id("Python")
4331023408
```

Each object has a reference count, if count reaches zero, it means no one is going to refer to it, then it will be garbage collected.

6.1 PyTypeObject

/Include/cpython/object.h

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;
```

(continues on next page)

(continued from previous page)

```
    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

#ifdef COUNT_ALLOCS
    /* these must be last and never explicitly initialized */
    Py_ssize_t tp_allocs;
    Py_ssize_t tp_frees;
    Py_ssize_t tp_maxalloc;
    struct _typeobject *tp_prev;
    struct _typeobject *tp_next;
#endif
} PyTypeObject;
```

CHAPTER 7

Code_object

CHAPTER 8

Frame_object

Each of function call is also called frame and each function has own frame and stack

_frame

PyFrameObject

the python interpreter maintain threads as PyThreadStates. Each PyThreadState keeps track of the top of its stack the stack is just a linked list of PyFrameObject PyFrameObject is the basic unit of bytecode evaluation Each frame uses its own stack to perform code execution

https://www.youtube.com/watch?v=smiL_aV1SOc https://www.youtube.com/watch?v=cSSpnq362Bk

CHAPTER 9

Python Execution Process

9.1 Install the necessary debug tool

```
$ sudo apt-get install gdb python3.5-dbg
$ gdb python3
```

1. Modules/Python.c (`break main`)
   - copy argv
   - sets locale
   - calls Py_Main()

9.1.1 Memory Management

- PyMem_ (Include/pymem.h, Objects/object.c)
- PyObject_memory

Py_Mem_Malloc() is based on malloc().

All of the python memory usages are based on `malloc`, `realloc`, `free`

2. Modules/main.c (`break Py_Main`)
   - _PyOS_Getopt
   - Py_GETENV
   - stdio
   - Py_Initialize: like bootstrapping, initialize all internal modules (Python/pythonrun.c)
   - run script / -m module / -c command
3. Objects/object.c (`break _PyObject_New`)

- Include/object.h
- Objects/object.c

Before next step, you need to run `clear _PyObject_New`

9.1.2 Protocols in C

- Include/abstract.c
- Objects/abstract.c
- Objects
- Buffer tp_as_buffer
- Number tp_as_number
- Mapping tp_as_mapping
- Sequence tp_as_sequence

9.1.3 Reference Counting

- Py_INCREF()
- Py_DECREF()

Does a null check:

- Py_XINCREF()
- Py_XDECREF():
- Py_CLEAR()

9.1.4 PyObject_memory

Include/objimpl.h Objects/obmalloc.c

PyObject_ small block allocator

struct arena_object -> Link List -> 256k address

4k block -> struct pool_header

- Double-free problem
- Memory leak

9.1.5 PyEval_EvalFrameEx

Main execution function. The big eval loop with a big switch.

- Python/ceval.c
- stack-based
- It has register
- It has opcode

Issue 4753 opcode optimization.

9.1.6 .pyc

- Python/marshal.c module: serialization the python code
- Include/marshal.h

The magic tag will change each time pyc file changes. Depends on time `st_mtime` of .pyc file.

CHAPTER 10

Disadvantage

- GIL
- Dynamic type

Python does not check the type at the compile time until it is executed. Therefore, an exception may be thrown when encountering wrong type operations. However, although Python uses a dynamic type mechanism, it is also a strong type. Python prohibits operations that are not explicitly defined, such as numbers and strings.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search