Inside The Python Virtual Machine

Obi Ike-Nwosu

Inside The Python Virtual Machine

Obi Ike-Nwosu

This book is for sale at http://leanpub.com/insidethepythonvirtualmachine

This version was published on 2020-08-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Obi Ike-Nwosu

Intermediate Python

Contents

1. Introduction

The Python Programming language has been around for a long time. Guido van Rossum started development work on the first version in 1989, and it has since grown to become one of the more popular languages used in a wide range of applications from graphical interfaces to finance[1] and data analysis[2].

This write-up looks at the nuts and bolts of the Python interpreter. It targets `CPython`, the most popular, and reference implementation of Python at the point of this write-up.

Python and CPython are used interchangeably in this text, but any mention of Python refers to CPython, the version of Python implemented in `C`. Other implementations include `PyPy` - Python implemented in a restricted subset of Python, `Jython` - Python implemented on the Java Virtual Machine, etc.

I regard the execution of a Python program as split into two or three main phases, as listed below. The relevant stages depend on how the interpreter is invoked, and this write-up covers them in different measures:

1. Initialization: This step covers the set up of the various data structures needed by the Python process and is only relevant when a program is executed non-interactively through the command prompt.
2. Compiling: This involves activities such as building syntax trees from source code, creating the abstract syntax tree, building the symbol tables, generating code objects etc.
3. Interpreting: This involves the execution of the generated code object's bytecode within some context.

The methods used in generating parse trees and syntax trees from source code are language-agnostic, so we do not spend much time on these. On the other hand, building symbol tables and code objects from the *Abstract Syntax tree* is the more exciting part of the compilation phase. This step is more Python-centric, and we pay particular attention to it. Topics we will cover include generating symbol tables, Python objects, frame objects, code objects, function objects etc. We will also look at how code objects are interpreted and the data structures that support this process.

This material is for anyone interested in gaining insight into how the CPython interpreter functions. The assumption is that the reader is already familiar with Python and understands the fundamentals of the language. As part of this exposition, we go through a considerable amount of `C` code, so a

[1] http://tpq.io/
[2] http://pandas.pydata.org/

reader with a rudimentary understanding of `C` will find it easier to follow. All that is needed to get through this material is a healthy desire to learn about the CPython virtual machine.

This work is an expanded version of personal notes taken while investigating the inner working of the Python interpreter. There is a substantial amount of wisdom in videos available in Pycon videos[3], school lectures[4] and blog write-ups[5]. This work will be incomplete without acknowledging these fantastic sources.

At the end of this write-up, a reader should understand the processes and data structures that are crucial to the execution of a Python program. We start next with an overview of the execution of a script passed as a command-line argument to the interpreter. Readers can install the `CPython` executable from the source by following the instructions at the Python Developer's Guide[6].

This material makes use of Python 3.6.

[3] https://www.youtube.com/watch?v=XGF3Qu4dUqk
[4] http://pgbovine.net/cpython-internals.htm/
[5] https://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/
[6] https://docs.python.org/devguide/index.html#

2. The View From 30,000ft

This chapter is a high-level overview of the processes involved in executing a Python program. Regardless of the complexity of a Python program, the techniques described here are the same. In subsequent chapters, we zoom in to give details on the various pieces of the puzzle. The excellent explanation of this process provided by Yaniv Aknin in his Python Internal series[1] provides some of the basis and motivation for this discussion.

A method of executing a Python script is to pass it as an argument to the Python interpreter as such `$python test.py`. There are other ways of interacting with the interpreter - we could start the interactive interpreter, execute a string as code, etc. However, these methods are not of interest to us. Figure 2.1 is the flow of activities involved in executing a module passed to the interpreter at the command-line.

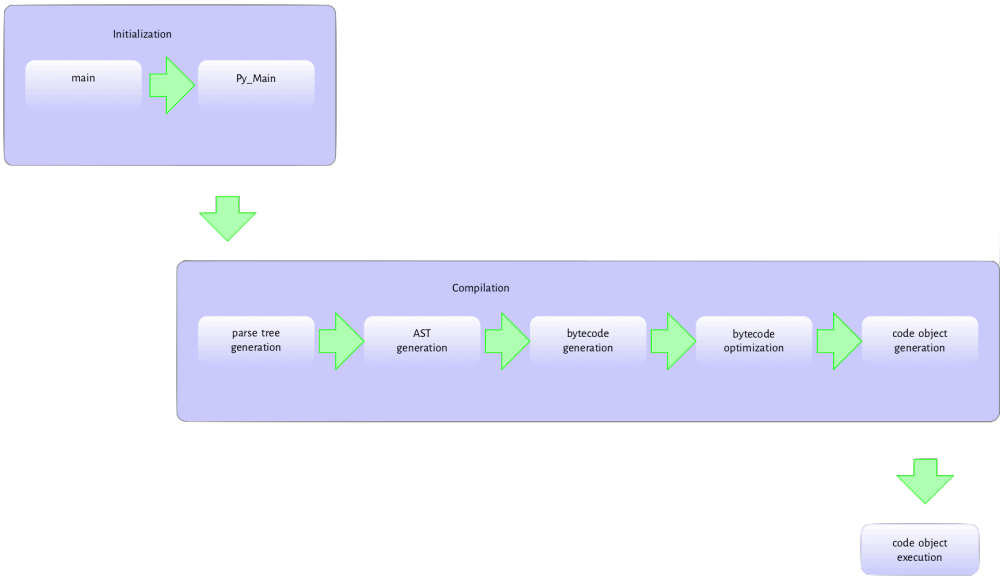


Figure 2.1: Flow during the execution of source code

The Python executable is a `C` program like any other `C` program such as the Linux kernel or a simple `hello world` program in `C` so pretty much the same process happens when we run the Python interpreter executable. The executable's entry point is the `main` method in the `Programs/python.c`. This `main` method handles basic initialization, such as memory allocation, locale setting, etc. Then, it invokes the `Py_Main` function in `Modules/main.c` responsible for the python specific initializations. These include parsing command-line arguments and setting program flags[2], reading environment variables, running hooks, carrying out hash randomization, etc. After, `Py_Main` calls the `Py_Initialize` function in `Programs/pylifecycle.c`; `Py_Initialize` is responsible for initializing

[1] https://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/
[2] https://docs.python.org/3.6/using/cmdline.html

the interpreter and all associated objects and data structures required by the Python runtime. After `Py_Initialize` completes successfully, we now have access to all Python objects.

This writeup assumes a Unix based operating system, so some specifics may differ when using a Windows operating system.

The interpreter state and interpreter state data structures are two examples of data structures that are initialized by the `Py_Initialize` call. A look at the data structure definitions for these provides some context into their functions. The interpreter and thread states are `C` structures with pointers to fields that hold information needed for executing a program. Listing 2.1 is the interpreter state `typedef` (just assume that `typedef` is `C` jargon for a type definition though this is not entirely true).

Listing 2.1: The interpreter state data structure

```
1    typedef struct _is {
2
3        struct _is *next;
4        struct _ts *tstate_head;
5
6        PyObject *modules;
7        PyObject *modules_by_index;
8        PyObject *sysdict;
9        PyObject *builtins;
10       PyObject *importlib;
11
12       PyObject *codec_search_path;
13       PyObject *codec_search_cache;
14       PyObject *codec_error_registry;
15       int codecs_initialized;
16       int fscodec_initialized;
17
18       PyObject *builtins_copy;
19   } PyInterpreterState;
```

Anyone who has used the Python programming language long enough may recognize a few of the fields mentioned in this structure *(sysdict, builtins, codec)*.

1. The `*next` field is a reference to another interpreter instance as multiple python interpreters can exist within the same process.
2. The `*tstate_head` field points to the main thread of execution - if the Python program is multithreaded, then the interpreter is shared by all threads created by the program - we discuss the structure of a thread state shortly.

3. The `modules`, `modules_by_index`, `sysdict`, `builtins`, and `importlib` are self-explanatory - they are all defined as instances of `PyObject` which is the root type of all Python objects in the virtual machine world. We provide more details about Python objects in the chapters that will follow.
4. The `codec*` related fields hold information that helps with the location and loading of encodings. These are very important for decoding bytes.

A Python program must execute in a thread. The thread state structure contains all the information needed by a thread to run some code. Listing 2.2 is a fragment of the thread data structure.

Listing 2.2: A cross-section of the thread state data structure

```
    typedef struct _ts {
        struct _ts *prev;
        struct _ts *next;
        PyInterpreterState *interp;

        struct _frame *frame;
        int recursion_depth;
        char overflowed;

        char recursion_critical;
        int tracing;
        int use_tracing;

        Py_tracefunc c_profilefunc;
        Py_tracefunc c_tracefunc;
        PyObject *c_profileobj;
        PyObject *c_traceobj;

        PyObject *curexc_type;
        PyObject *curexc_value;
        PyObject *curexc_traceback;

        PyObject *exc_type;
        PyObject *exc_value;
        PyObject *exc_traceback;

        PyObject *dict;  /* Stores per-thread state */
        int gilstate_counter;

        ...
    } PyThreadState;
```

More details on the interpreter and the thread state data structures will follow in subsequent chapters. The initialization process also sets up the import mechanisms as well as rudimentary stdio.

After the initialization, the `Py_Main` function invokes the `run_file` function also in the `main.c` module. The following series of function calls: `PyRun_AnyFileExFlags -> PyRun_SimpleFileExFlags->PyRun_FileExFlags->PyParser_ASTFromFileObject` are made to the `PyParser_ASTFromFileObject` function. The `PyRun_SimpleFileExFlags` function call creates the `__main__` namespace in which the file contents will be executed. It also checks for the presence of a `pyc` version of the module - the `pyc` file contains the compiled version of the executing module. If a `pyc` version exists, it will attempt to read and execute it. Otherwise, the interpreter invokes the`PyRun_FileExFlags` function followed by a call to the `PyParser_ASTFromFileObject` function and then the `PyParser_ParseFileObject` function. The `PyParser_ParseFileObject` function reads the module content and builds a parse tree from it. The `PyParser_ASTFromNodeObject` function is then called with the parse tree as an argument and creates an abstract syntax tree (AST) from the parse tree.

If you have been following the actual `C` source code, you must have run into the `Py_INCREF` and `Py_DECREF` by now. These are memory management functions that will be discussed later on in more detail. CPython manages the object life cycle using reference counting; whenever we create a new reference to an object, a call to the `Py_INCREF` function is used to increase with the reference count of that object by 1. Whenever a reference goes out of scope, a call to the `Py_DECREF` functions reduces the reference count by 1.

The *AST* generated is then passed to the `run_mod` function. This function invokes the `PyAST_CompileObject` function that creates code objects from the *AST*. Do note that the bytecode generated during the call to `PyAST_CompileObject` is passed through a simple `peephole` optimizer that carries out low hanging optimization of the generated bytecode before creating the code object. With the code objects created, it is time to execute the instructions encapsulated by the code objects. The `run_mod` function invokes `PyEval_EvalCode` from the `ceval.c file` with the code object as an argument. This results in another series of function calls: `PyEval_EvalCode->PyEval_EvalCode->_PyEval_EvalCodeWithName->_PyEval_EvalFrameEx`. The code object is an argument to most of these functions. The `_PyEval_EvalFrameEx` is the actual execution loop that handles executing the code objects. This function gets called with a *frame object* as an argument. This *frame object* provides the context for executing the code object. The execution loop reads and executes instructions from an array of instructions, adding or removing objects from the value stack in the process (*where is this value stack?*), till there are no more instructions to execute or something exceptional that breaks this loop occurs.

Python provides a set of functions that one can use to explore actual code objects. For example, a a simple program can be compiled into a code object and disassembled to get the opcodes that are executed by the Python virtual machine, as shown in listing 2.3.

Listing 2.3: Disassembling a python function

```
>>> from dis import dis
>>> def square(x):
...     return x*x
...

>>> dis(square)
2           0 LOAD_FAST                0 (x)
            2 LOAD_FAST                0 (x)
            4 BINARY_MULTIPLY
            6 RETURN_VALUE
```

The `./Include/opcodes.h` file contains a listing of the Python Virtual Machine's bytecode instructions. The opcodes are pretty straight forward conceptually. Take our example from listing 2.3 with four instructions - the LOAD_FAST opcode loads the value of its argument (`x` in this case) onto an evaluation (value) stack. The Python virtual machine is a stack-based virtual machine, so values for operations and results from operations live on a stack. The `BINARY_MULTIPLY` opcode then pops two items from the value stack, performs binary multiplication on both values, and places the result back on the value stack. The `RETURN VALUE` opcode pops a value from the stack, sets the return value object to this value, and breaks out of the interpreter loop. From the disassembly in listing 2.3, it is pretty clear that this rather simplistic explanation of the operation of the interpreter loop leaves out a lot of details. A few of these outstanding questions may include.

1. Where are the values such as that loaded by the LOAD_FAST instruction gotten from?
2. Where do arguments used as part of the instructions come from?
3. How are nested function and method calls managed? 4 How does the interpreter loop handle exceptions?

After the module's execution, the `Py_Main` function continues with the clean-up process. Just as `Py_Initialize` performs initialization during the interpreter startup, `Py_FinalizeEx` is invoked to do some clean-up work; this clean-up process involves waiting for threads to exit, calling any exit hooks, freeing up any memory allocated by the interpreter that is still in use, and so on, paving the way for the interpreter to exit.

The above is a high-level overview of the processes involved in executing a Python module. A lot of details are left out at this stage, but all will be revealed in subsequent chapters. We continue in the next chapter with a description of the compilation process.

3. Compiling Python Source Code

Although most people may not regard Python as a compiled language, it is one. During compilation, the interpreter generates executable bytecode from Python source code. However, Python's compilation process is a relatively simple one. It involves the following steps in order.

1. Parsing the source code into a parse tree.
2. Transforming the parse tree into an abstract syntax tree (AST).
3. Generating the symbol table.
4. Generating the code object from the AST. This step involves:
    1. Transforming the AST into a flow control graph, and
    2. Emitting a code object from the control flow graph.

Parsing source code into a parse tree and creating an AST from such a parse is a standard process and Python does not introduce any complicated nuances, so the focus of this chapter is on the transformation of an AST into a control flow graph and the emission of code object from the control flow graph. For anyone interested in parse tree and AST generation, the dragon book[1] provides an in-depth *tour de force* of both topics.

3.1 From Source To Parse Tree

The Python parser is an LL(1)[2] parser based on the description of such parsers laid out in the Dragon book. The `Grammar/Grammar` module contains the Extended Backus-Naur Form *(EBNF)* grammar specification of the Python language. Listing 3.0 is a cross-section of this grammar.

Listing 3.0: A cross section of the Python BNF Grammar

```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
        import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (augassign (yield_expr|testlist) |
                ('=' (yield_expr|testlist_star_expr))*)
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^='
        | '<<=' | '>>=' | '**=' | '//=')

```

[1] https://www.amazon.co.uk/Compilers-Principles-Techniques-Alfred-Aho/dp/0201100886
[2] https://en.wikipedia.org/wiki/LL_parser

```
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt |
        yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
        'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

        ...
```

The `PyParser_ParseFileObject` function in `Parser/parsetok.c` is the entry point for parsing any module passed to the interpreter at the command-line. This function invokes the `PyTokenizer_FromFile` function that is responsible for generating tokens from the supplied modules.

3.2 Python tokens

Python source code consists of tokens. For example, `return` is a keyword token; `2` is a literal numeric token. Tokenization, the splitting of source code into constituent tokens, is the first task during parsing. The tokens from this step fall into the following categories.

1. identifiers: These are names defined by a programmer. They include function names, variable names, class names, etc. These must conform to the rules of identifiers specified in the Python documentation.
2. operators: These are special symbols such as `+`, `*` that operate on data values, and produce results.
3. delimiters: This group of symbols serve to group expressions, provide punctuations, and assignment. Examples in this category include `(`, `)`, `{`, `}`, `=`, `*=` etc.

4. literals: These are symbols that provide a constant value for some type. We have the string and byte literals such as `"Fred"`, `b"Fred"` and numeric literals which include integer literals such as `2`, floating-point literal such as `1e100` and imaginary literals such as `10j`.
5. comments: These are string literals that start with the hash symbol. Comment tokens always end at the end of the physical line.
6. NEWLINE: This is a unique token that denotes the end of a logical line.
7. INDENT and DEDENT: These token represent indentation levels that group compound statements.

A group of tokens delineated by the **NEWLINE** token makes up a logical line; hence we could say that a Python program consists of a sequence of *logical lines*. Each of these logical lines consists of several physical lines that are each terminated by an end-of-line sequence. Most times, logical lines map to physical lines, so we have a logical line delimited by end-of-line characters. These logical lines usually map to Python statements. Compound statements may span multiple physical lines; parenthesis, square brackets or curly braces around a statement implicitly joins the logical lines that make up such statement. The backslash character, on the other hand, is needed to join multiple logical lines explicitly.

Indentation also plays a central role in grouping Python statements. One of the lines in the Python grammar is thus `suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT` so a crucial task of the tokenizer generating `indent` and `dedent` tokens that go into the parse tree. The tokenizer uses an algorithm similar to that in Listing 3.1 to generate these `INDENT` and `DEDENT` tokens.

Listing 3.1: Python indentation algorithm for generting INDENT and DEDENT tokens

```
Init the indent stack with the value 0.
For each logical line taking into consideration line-joining:
    A.  If the current line's indentation is greater than the
    indentation at the top of the stack
        1.  Add the current line's indentation to the top of the stack.
        2.  Generate an INDENT token.
    B.  If the current line's indentation is less than the indentation
        at the top of the stack
        1. If there is no indentation level on the stack that matches the current li\
ne's indentation, report an error.
        2.  For each value at the top of the stack, that is unequal to the current l\
ine's indentation.
            a.  Remove the value from the top of the stack.
            b.  Generate a DEDENT token.
    C.  Tokenize the current line.
For every indentation on the stack except 0, produce a DEDENT token.
```

The `PyTokenizer_FromFile` function in the `Parser/tokenizer.c` scans the source file from left to right and top to bottom tokenizing the file's contents and then outputting a `tokenizer` structure.

Whitespaces characters other than terminators serve to delimit tokens but are not compulsory. In cases of ambiguity such as in `2+2`, a token comprises the longest possible string that forms a legal token reading from left to right; in this example, the tokens are the literal `2`, the operator `+` and the literal `2`.

The tokenizer structure generated by the `PyTokenizer_FromFile` function gets passed to the `parsetok` function that attempts to build a parse tree according to the Python grammar of which Listing 3.0 is a subset. When the parser encounters a token that violates the Python grammar, it raises a `SyntaxError` exception. The `parser`[3] module provides limited access to the parse tree of a block of Python code, and listing 3.2 is a basic demonstration.

Listing 3.2: Using the parser module to obtain the parse tree of python code

```
>>>code_str = """def hello_world():
                    return 'hello world'
                """
>>> import parser
>>> from pprint import pprint
>>> st = parser.suite(code_str)
>>> pprint(parser.st2list(st))
[257,
[269,
[294,
[263,
    [1, 'def'],
    [1, 'hello_world'],
    [264, [7, '('], [8, ')']],
    [11, ':'],
    [303,
    [4, ''],
    [5, ''],
    [269,
    [270,
    [271,
        [277,
        [280,
        [1, 'return'],
        [330,
        [304,
            [308,
            [309,
            [310,
            [311,
```

[3] https://docs.python.org/3.6/library/parser.html#module-parse

```
                [314,
                [315,
                [316,
                [317,
                    [318,
                    [319,
                    [320,
                    [321,
                        [322, [323, [3, '"hello world"']]]]]]]]]]]]]]]]]]],
    [4, '']]],
    [6, '']]]]],
[4, ''],
[0, '']]
>>>
```

The `parser.suite(source)` call in listing 3.2 returns an intermediate representation of a parse tree (`ST`) object while the call to `parser.st2list` returns the parse tree represented by a Python list - each list represents a node of the parse tree. The first items in each list, the integer, identifies the production rule in the Python grammar responsible for that node.

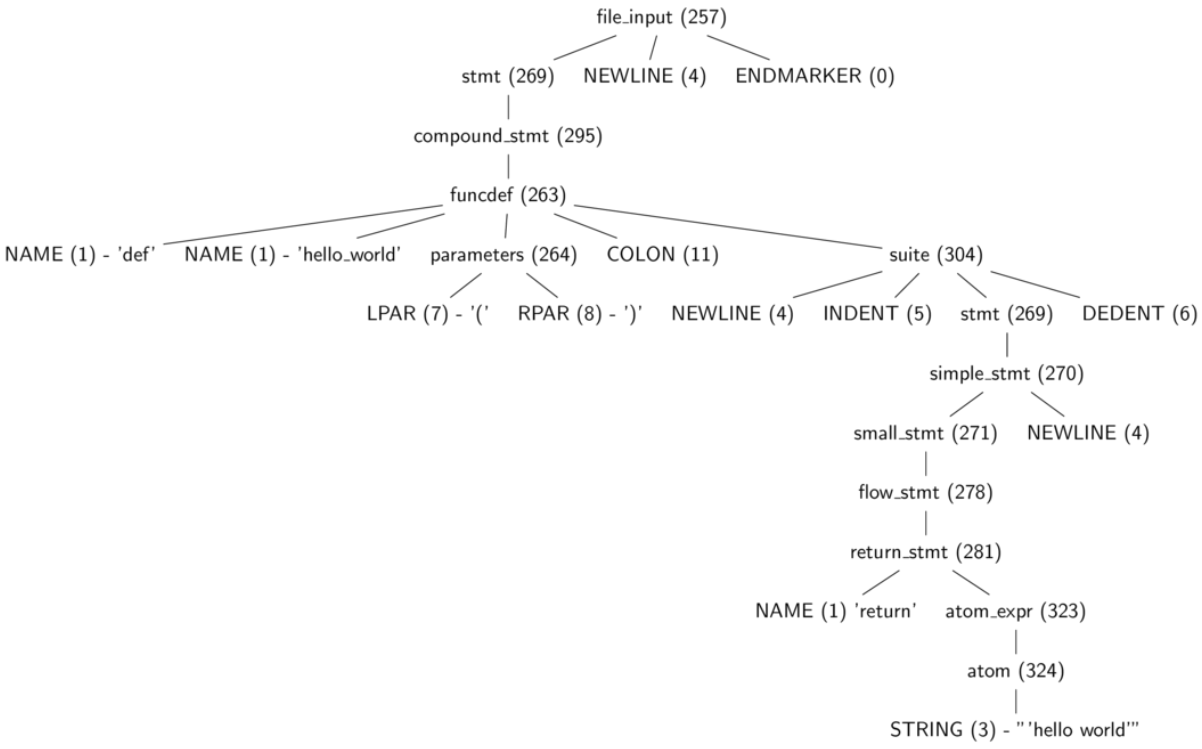


Figure 3.0: A parse tree for listing 3.2 (a function that returns the 'hello world' string)

Figure 3.0 is a tree diagram of the same parse tree from listing 3.2 with some tokens stripped away, and one can see more easily the part of the grammar each of the integer value represents. These production rules are all specified in the `Include/token.h` (terminals) and `Include/graminit.h` (terminals) header files.

Listing 3.3 from the `Include/node.h` shows the data structure that represents a parse tree in CPython. Each production rule is a *node* on the tree data structure. Each node has child nodes, and together the root node and its offspring nodes represent the parse tree.

Listing 3.3: The node data structure used in the CPython virtual machine

```
typedef struct _node {
    short           n_type;
    char            *n_str;
    int             n_lineno;
    int             n_col_offset;
    int             n_nchildren;
    struct _node    *n_child;
} node;
```

The macros for interacting with parse tree nodes are also in the `Include/node.h` file.

3.3 From Parse Tree To Abstract Syntax Tree

The parse tree is dense with information about Python's syntax, and all that information such as how lines are delimited is irrelevant for generating bytecode. This is where the abstract syntax tree (AST) comes in. The abstract syntax tree is a representation of the code that is independent of Python's syntax niceties. For example, a parse tree contains syntax constructs such as colon and NEWLINE nodes, as shown in figure 3.0, but the AST does not include such syntax construct as shown in listing 3.4. The transformation of the parse tree to the abstract syntax tree is the next step in the compilation pipeline.

Listing 3.4: Using the `ast` module to manipulate the AST of python source code

```
>>> import ast
>>> import pprint
>>> node = ast.parse(code_str, mode="exec")
>>> ast.dump(node)
("Module(body=[FunctionDef(name='hello_world', args=arguments(args=[], "
'vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None, defaults=[]), '
"body=[Return(value=Str(s='hello world'))], decorator_list=[], "
'returns=None)])')
```

Python makes use of the Zephyr Abstract Syntax Definition Language (ASDL), and the ASDL definitions of the various Python constructs are in the file `Parser/Python.asdl` file. Listing 3.5 is a fragment of the *ASDL* definition of a Python statement.

Listing 3.5: A fragment of ASDL definition of a Python statement

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? returns)

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list)
      | Return(expr? value)

      | Delete(expr* targets)
      | Assign(expr* targets, expr value)
      | AugAssign(expr target, operator op, expr value)
      -- 'simple' indicates that we annotate simple name without parens
      | AnnAssign(expr target, expr annotation, expr? value, int simple)

      -- use 'orelse' because else is a keyword in target languages
      | For(expr target, expr iter, stmt* body, stmt* orelse)
      | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
      | While(expr test, stmt* body, stmt* orelse)
      | If(expr test, stmt* body, stmt* orelse)
      | With(withitem* items, stmt* body)
      | AsyncWith(withitem* items, stmt* body)
```

In the CPython, `C` structures defined in `Include/Python-ast.h` represent the various `AST` nodes. The `Parser/asdl_c.py` module auto-generates generates this file from the `AST` *ASDL* definitions. Listing 3.6 is a fragment of the C code generated for a Python `statement` node.

Listing 3.6: A cross-section of an AST statement node data A> A> structure

```
      struct _stmt {
           enum _stmt_kind kind;
           union {
               struct {
                    identifier name;
                   arguments_ty args;
                    asdl_seq *body;
                    asdl_seq *decorator_list;
                   expr_ty returns;
               } FunctionDef;

               struct {
                   identifier name;
                   arguments_ty args;
                    asdl_seq *body;
                    asdl_seq *decorator_list;
                    expr_ty returns;
               } AsyncFunctionDef;

               struct {
                    identifier name;
                   asdl_seq *bases;
                    asdl_seq *keywords;
                    asdl_seq *body;
                    asdl_seq *decorator_list;
               } ClassDef;
               ...
           }v;
           int lineno;
           int col_offset
       }
```

The union type in listing 3.6 is a `C` type that can represent any of the types in the union.

The `PyAST_FromNode` function in the `Python/ast.c` calls `PyAST_FromNodeObject` also in `Python/ast.c` which walks the various parse tree nodes and generates AST nodes accordingly using functions defined in `Python/ast.c`. The heart of this function is a large switch statement that calls node specific functions on each node type. For example, the code responsible for generating the `AST` node for an

`if` expression is in listing 3.7.

Listing 3.7: Function for generating an AST node for an if expression

```
static expr_ty
ast_for_ifexpr(struct compiling *c, const node *n)
{
    /* test: or_test 'if' or_test 'else' test */
    expr_ty expression, body, orelse;

    assert(NCH(n) == 5);
    body = ast_for_expr(c, CHILD(n, 0));
    if (!body)
        return NULL;
    expression = ast_for_expr(c, CHILD(n, 2));
    if (!expression)
        return NULL;
    orelse = ast_for_expr(c, CHILD(n, 4));
    if (!orelse)
        return NULL;
    return IfExp(expression, body, orelse, LINENO(n), n->n_col_offset,
                 c->c_arena);
}
```

We will see that all other interactions with the AST make use of a similar pattern where node specific functions recursively walk the AST from left to right with the help of macros in `Include/asdl.h` and node-specific functions.

Listing 3.8: A simple python function

```
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

Take the code in Listing 3.8, for example, the transformation of its parse tree to an AST will result in an AST similar to figure 3.1.

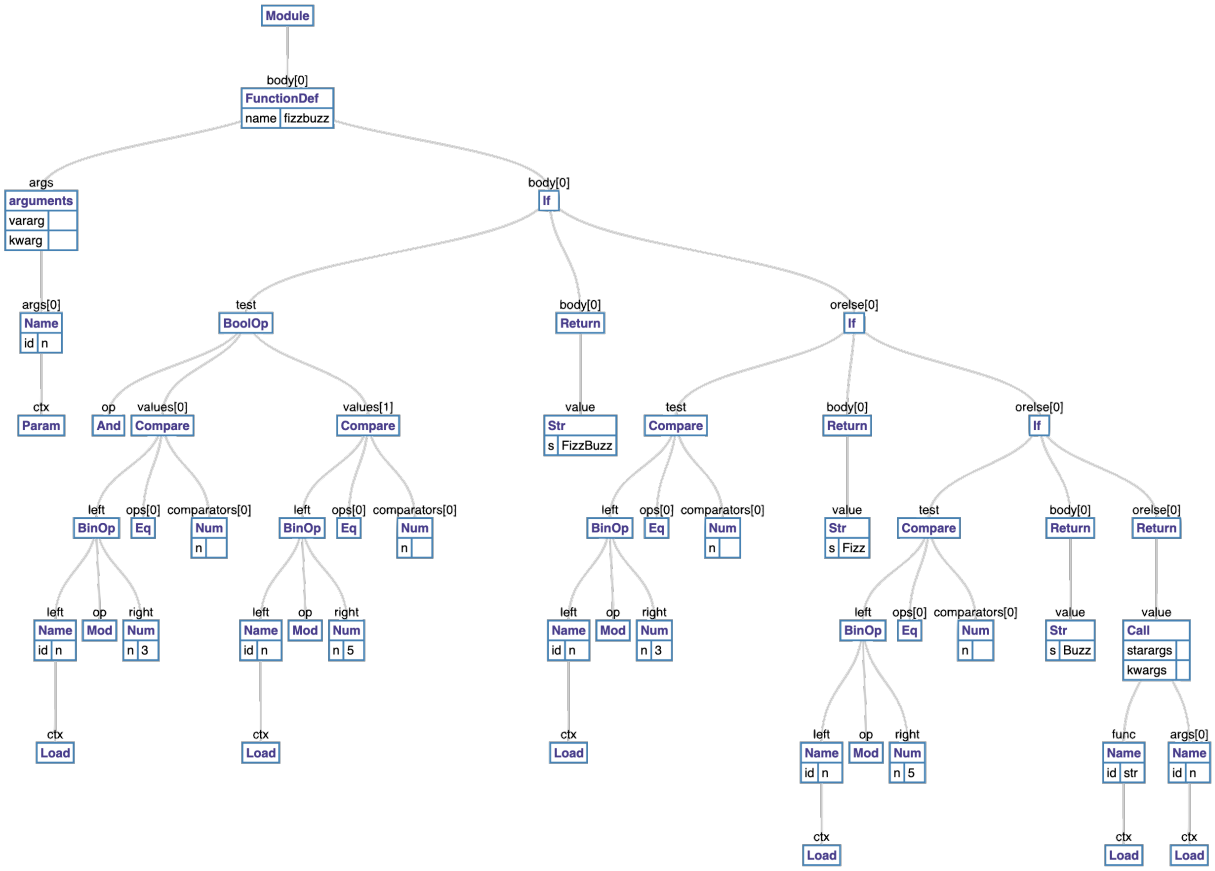


Figure 3.1: An AST for listing 3.2

The `ast` module bundled with the Python interpreter provides us with the ability to manipulate a Python AST. Tools such as codegen[4] can take an `AST` representation in Python and output the corresponding Python source code.

With the AST generated, the next step is creating the symbol table.

3.4 Building The Symbol Table

The symbol table, as the name suggests, is a collection of symbols and their use context within a code block. Building the symbol table involves analyzing and assigning scoping to the names in a code block.

Names and Binding

In Python, objects are referenced by *names*. `names` are analogous to *but not exactly* variables in `C++` and `Java`.

[4] https://pypi.python.org/pypi/codegen/1.0

```
>>> x = 5
```

In the above example, `x` is a name that references the object, `5`. The process of *assigning* a reference to `5` to `x` is called *binding*. A binding causes a name to be associated with an object in the innermost scope of the currently executing program. Bindings may occur during several instances such as during variable assignment or function or method call when the supplied parameter is bound to the argument. It is important to note that names are just symbols and they have no *type* associated with them; **names are only references to objects that have types**.

Code Blocks

Code blocks are central to Python program, and a understanding of them is crucial. A code block is a piece of program code that executes as a single unit in Python. Modules, functions, and classes are all examples of code blocks. Commands typed in interactively at the REPL, script commands run with the `-c` option are also code blocks. A code block has several namespaces associated with it. For example, a module code block has access to the `global` namespace while a function code block has access to the `local` as well as the `global` namespaces.

Namespaces

A * namespace* defines a context in which a given set of names is bound to objects. Namespaces are implemented using dictionary mappings. The *builtin* namespace is an example of a namespace that contains all the built-in functions and this can be accessed by entering `__builtins__.__dict__` at the terminal (the result is of a considerable amount). The interpreter has access to multiple namespaces including *the global namespace*, *the builtin namespace* and *the local namespace*. These namespaces are created at different times and have different lifetimes. For example, invoking a function will create a new *local* namespace and the interpreter drops that namespace when the function exits or returns. The *global* namespace is created at the start of the execution of a module and all names defined in this namespace are available module-wide while the *built-in* namespace comes into existence when the interpreter is invoked and contains all the builtin names. These three namespaces are the main namespaces available to the interpreter.

Scopes

A scope is an area of a program in which a set of name bindings (namespaces) is visible and directly accessible without using any dot notation. At runtime, the following scopes may be available.

1. Innermost scope with local names,
2. The scope of enclosing functions if any (this is applicable for nested functions),
3. The current module's `globals` scope, and
4. The scope containing the builtin namespace.

When a name used, the interpreter searches the scopes' namespaces in the order listed above; if the interpreter does not find that name, it raises an exception. Python supports *static scoping* also known as *lexical scoping*; this means that the visibility of a set of name bindings can be inferred by only inspecting the program text.

Note

Python has a quirky scoping rule that prohibits modifying a reference to an object in the *global* scope from a local scope; such an attempt will throw an `UnboundLocalError` exception. In order to modify an object from the global scope within a local scope, the `global` keyword is used with the object name before attempting any modification. The snippet in Listing 3.9 illustrates this.

Listing 3.9: Attempting to modify a global variable from a function

```
>>> a = 1
>>> def inc_a(): a += 2
...
>>> inc_a()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 1, in inc_a
UnboundLocalError: local variable 'a' referenced before assignment
```

The `global` statement is used to modify the object from the global scope, as shown in listing 3.10.

Listing 3.10: Using the global keyword to modify a global variable from a function

```
>>> a = 1
>>> def inc_a():
...     global a
...     a += 1
...
>>> inc_a()
>>> a
2
```

Python also has the `nonlocal` keyword used when there is a need to modify a variable bound in an outer non-global scope from an inner scope. This keyword is handy when working with nested functions (also referred to as closures). The snippet in listing 3.11 that defines a simple counter object illustrates the usage of the `nonlocal` keyword.

Listing 3.11: Creating blocks from an AST

```
>>> def make_counter():
...     count = 0
...     def counter():
...         nonlocal count #capture count binding from enclosing not global scope
...         count += 1
...         return count
...     return counter
...
>>> counter_1 = make_counter()
>>> counter_2 = make_counter()
>>> counter_1()
1
```

```
>>> counter_1()
2
>>> counter_2()
1
>>> counter_2()
2
```

Symbol table data structures

Two data structures that are central to generating a symbol table are:

1. The `symtable` data structure.
2. The `symtable_entry` data structure.

Listing 3.13 shows the `symtable` data structure. This data structure is a table composed of entries that hold information on A> the names used in a module's code blocks.

Listing 3.13: The symtable data structure

```
        struct symtable {
            PyObject *st_filename;          /* name of file being compiled */
            struct _symtable_entry *st_cur; /* current symbol table entry */
            struct _symtable_entry *st_top; /* symbol table entry for module */
            PyObject *st_blocks;            /* dict: map AST node addresses
                                            *       to symbol table entries */
            PyObject *st_stack;             /*list: stack of namespace info */
            PyObject *st_global;            /*borrowed ref to
                                            st_top->ste_symbols*/
            int st_nblocks;                 /* number of blocks used. kept for
                                            consistency with the corresponding
                                            compiler structure */
            PyObject *st_private;           /* name of current class or NULL */
            PyFutureFeatures *st_future;    /* module's future features that
                                            affect the symbol table */
            int recursion_depth;            /* current recursion depth */
            int recursion_limit;            /* recursion limit */
        };
```

A module may contain many code blocks - such as multiple function definitions - and the `st_blocks` field maps each of the A> code blocks to a symbol table entry. The `st_top` is a symbol table entry for the module (recall that a module is also a code block), so it contains the names defined in the modules' global namespace. The `st_cur` is the symbol table entry for the current code block. Figure 3.2 is a graphical illustration of this structure.

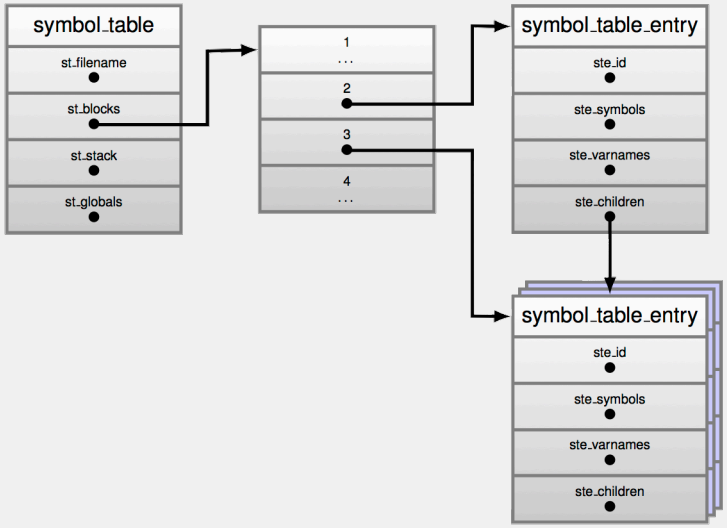


Figure 3.2: A Symbol table and symbol table entries.

Listing 3.14 is the definition of the `_symtable_entry` data structure; this is also in `Include/symtable.h`.

Listing 3.14: The _symtable_entry data structure

```
        typedef struct _symtable_entry {
            PyObject_HEAD
            PyObject *ste_id;               /* int: key in ste_table->st_blocks */
            PyObject *ste_symbols;          /* dict: variable names to flags */
            PyObject *ste_name;             /* string: name of current block */
            PyObject *ste_varnames;         /* list of function parameters */
            PyObject *ste_children;         /* list of child blocks */
            PyObject *ste_directives;       /* locations of global and nonlocal
                                               statements */
            _Py_block_ty ste_type;          /* module, class, or function */
            int ste_nested;                 /* true if block is nested */
            unsigned ste_free : 1;          /*true if block has free variables*/
            unsigned ste_child_free : 1;    /* true if a child block has free
                                               vars including free refs to globals*/
            unsigned ste_generator : 1;     /* true if namespace is a generator */
            unsigned ste_varargs : 1;       /* true if block has varargs */
            unsigned ste_varkeywords : 1;   /* true if block has varkeywords */
            unsigned ste_returns_value : 1; /* true if namespace uses return with
                                               an argument */
            unsigned ste_needs_class_closure : 1; /* for class scopes, true if a
                                                     closure over __class__
                                                     should be created */
            int ste_lineno;          /* first line of block */
            int ste_col_offset;      /* offset of first line of block */
            int ste_opt_lineno;      /* lineno of last exec or import * */
```

```
26              int ste_opt_col_offset;  /* offset of last exec or import * */
27              int ste_tmpname;         /* counter for listcomp temp vars */
28              struct symtable *ste_table;
29          } PySTEntryObject;
```

The comments in Listing 3.14 explain the function of each field. The `ste_symbols` field is a mapping of symbols in the code block to context flags. The flags are numeric values that provide information on the use context of the given symbol. For example, a symbol may be a function argument or a global statement definition. Listing 3.15 are some examples of the flags in `Include/symtable.h`.

Listing 3.15: Flags that specify context of a name definition

```
1               /* Flags for def-use information */
2               #define DEF_GLOBAL 1           /* global stmt */
3               #define DEF_LOCAL 2            /* assignment in code block */
4               #define DEF_PARAM 2<<1         /* formal parameter */
5               #define DEF_NONLOCAL 2<<2      /* nonlocal stmt */
6               #define DEF_FREE 2<<4          /* name used but not defined in
7                                                 nested block */
```

The `PySymtable_BuildObject` function in `Python/compile.c` walks the AST to create the symbol table. This is a two-step process summarized in listing 3.12.

Listing 3.12: Creating a symbol table from an AST

```
For each node in a given AST
    If the node is the start of a code block:
        1. Create a new symbol table entry and set the current symbol table to this \
value.
        2. Push the new symbol table to st_stack.
        3. Add the new symbol table to the list of children of the previous symbol t\
able.
        4. Update the current symbol table to this new symbol table
        5. For all nodes in the code block nodes:
            a. recursively visit each node using the "symtable_visit_XXX"
                functions where "XXX" is a node type.
        6. Exit the code block by removing the current symbol table entry from the s\
tack.
        7. Pop the next symbol table entry from the stack and set the
        current symbol table entry to this popped value
    else:
        recursively visit the node and sub-nodes.
```

First, we visit each node of the `AST` to build a collection of symbols used. After the first pass, the symbol table entries contain all names that have been used within the module, but it does not have contextual information about such names. For example, the interpreter cannot tell if a given variable is a global, local, or free variable. The `symtable_analyze` function in the `Parser/symtable.c` handles the second phase. In this phase, the algorithm assigns scopes (local, global, or free) to the symbols gathered from the first pass. The comments in the `Parser/symtable.c` are quite informative and are paraphrased below to provide some insight into the second phase of the symbol table construction process.

> The symbol table requires two passes to determine the scope of each name. The first pass collects raw facts from the AST via the symtable_visit_* functions while the second pass analyzes these facts during a pass over the PySTEntryObjects created during pass 1. During the second pass, the parent passes the set of all name bindings visible to its children when it enters a function. These bindings determine if nonlocal variables are free or implicit globals. Names which are explicitly declared nonlocal must exist in this set of visible names - if they do not, the interpreter raises a syntax error. After the local analysis, it analyzes each of its child blocks using an updated set of name bindings.
>
> There are also two kinds of global variables, implicit and explicit. An explicit global is declared with the global statement. An implicit global is a free variable for which the compiler has found no binding in an enclosing function scope. The implicit global is either a global or a builtin.
> Python's module and class blocks use the `xxx_NAME` opcodes to handle these names to implement slightly odd semantics. In such a block, the name is treated as global until it is assigned a value; then it is treated like a local.
>
> The children update the free variable set. If a child adds a variable to the set of free variables, then such variable is marked as a cell. The function object defined must provide runtime storage for the variable that may outlive the function's frame. Cell variables are removed from the free set before the analyze function returns to its parent.

For example, a symbol table for a module with content in listing 3.16 will contain three symbol table entries.

Listing 3.16: A simple function

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

The first entry is that of the enclosing module, and it will have `make_counter` defined with a local scope. The next symbol table entry will be that of function `make_counter`, and this will have the

`count` and `counter` names marked as local. The final symbol table entry will be that of the nested `counter` function. This entry will have the count variable marked as `free`. One thing to note is that although `make_counter` has a local scope in the symbol table entry for the module block, it is globally defined in the module code block because the `*st_global` field of the symbol table points to the `*st_top` symbol table entry which is that of the enclosing module.

3.5 From AST To Code Objects

After generating the symbol table, the next step is creating code objects. The functions for this step are in the `Python/compile.c` module. First, they convert the `AST` into basic blocks of Python byte code instructions. Basic blocks are blocks of code that have a single entry but can have multiple exits. The algorithm here uses a pattern similar to that used to generate the symbol table. Functions named `compiler_visit_xx`, where `xx` is the node type, recursively visit each node of the AST emitting bytecode instructions in the process. We see some examples of these functions in the sections that follow. The blocks of bytecode here implicitly represent a graph, the control flow graph. This graph shows the potential code execution paths. In the second step, the algorithm flattens the control flow graph using a post-order depth-first search transversal. After, the jump offsets are calculated and used as instruction arguments for bytecode `jump` instructions. The bytecode instructions are then used to create a code object.

The `compiler` data structure

Figure 3.3 shows the relationship between the primary data structures used in the process of generating the fundamental blocks that make up the control flow graph.

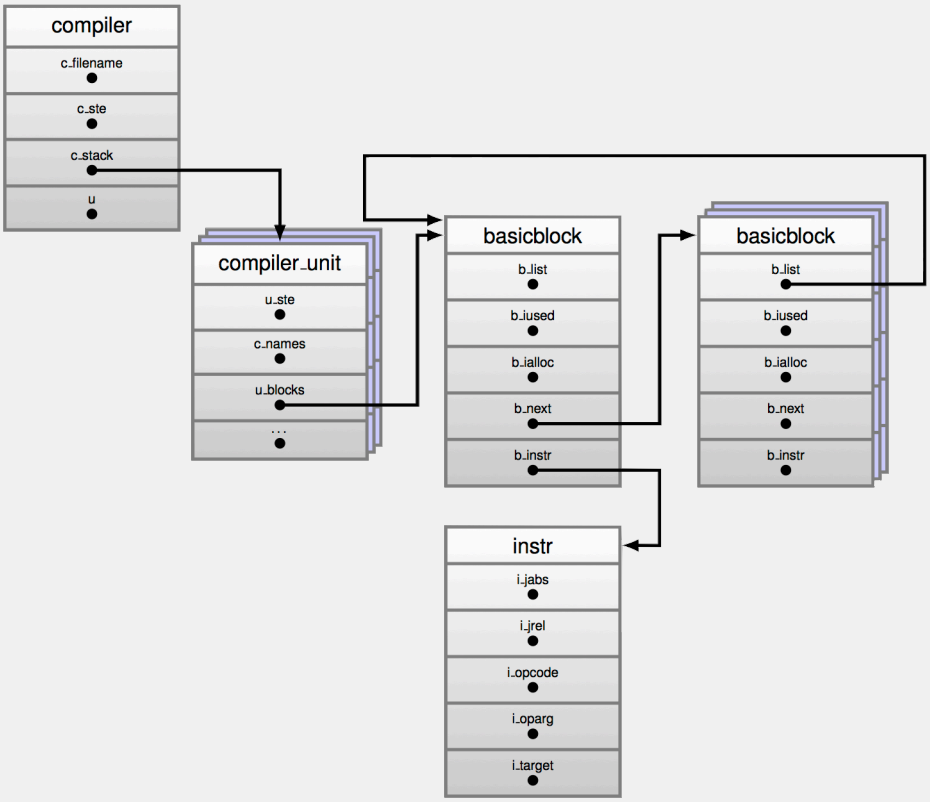


Figure 3.3: The four major data structures used in generating a code object.

At the very top level is the `compiler` data structure, which captures the global compilation process for a module. Listing 3.17 is a definition of this data structure.

Listing 3.17: The `compiler` data structure

```
      struct compiler {
          PyObject *c_filename;
          struct symtable *c_st;
          PyFutureFeatures *c_future; /* pointer to module's __future__ */
          PyCompilerFlags *c_flags;

          int c_optimize;              /* optimization level */
          int c_interactive;           /* true if in interactive mode */
          int c_nestlevel;

          struct compiler_unit *u; /* compiler state for current block */
```

```
12              PyObject *c_stack;          /* Python list holding compiler_unit
13                                         ptrs */
14              PyArena *c_arena;           /* pointer to memory allocation arena */
15          };
```

The fields that are of interest to us here are the following.

1. `*c_st`: a reference to the symbol table generated in the previous section.
2. `*u`: a reference to a compiler unit data structure. This encapsulates information needed for working with a code block. This field points to the compiler unit for the current code block that is being compiled.
3. `*c_stack`: a reference to a stack of `compiler_unit` data structures. When a code block is composed of multiple code blocks, this field manages the saving and restoration of compile_-unit data structures as the compiler encounters new blocks. When the compiler enters a new code block - it creates a new scope - then the `compiler_enter_scope()` pushes the current compiler_unit - `*u`- onto the stack - `*c_stack`, creates a new compiler_unit object, and sets it as the current state for that new block encountered. When the compiler exits the block, the `*c_stack` is popped off the stack accordingly to restore the state.

The compiler initializes a `compiler` data structure to compile every Python module; as the AST generated for the module is walked, the compiler creates a `compiler_unit` data structure for each code block that it encounters within the `AST`.

The `compiler_unit` data structure

The `compiler_unit` data structure shown in Listing 3.18 captures the information needed to generate the required byte code instructions for a code block. We discuss most of the fields defined in the `compiler_unit` when we look at code objects.

Listing 3.18: The `compiler_unit` data strcuture

```
1       struct compiler_unit {
2           PySTEntryObject *u_ste;
3
4           PyObject *u_name;
5           PyObject *u_qualname;  /* dot-separated qualified name (lazy) */
6           int u_scope_type;
7
8           /* The following fields are dicts that map objects to the index of them i\
9  n co_XXX. The index is an argument for opcodes that refer to those collections.
10          */
11          PyObject *u_consts;    /* all constants */
12          PyObject *u_names;     /* all names */
13          PyObject *u_varnames;  /* local variables */
14          PyObject *u_cellvars;  /* cell variables */
15          PyObject *u_freevars;  /* free variables */
16
```

```
17              PyObject *u_private;        /* for private name mangling */
18
19              Py_ssize_t u_argcount;        /* number of arguments for block */
20              Py_ssize_t u_kwonlyargcount; /* number of keyword only arguments
21                                              for block */
22              /* Pointer to the most recently allocated block.  By following b_list
23              members, you can reach all early allocated blocks. */
24              basicblock *u_blocks;
25              basicblock *u_curblock; /* pointer to current block */
26
27              int u_nfblocks;
28              struct fblockinfo u_fblock[CO_MAXBLOCKS];
29
30              int u_firstlineno; /* the first lineno of the block */
31              int u_lineno;          /* the lineno for the current stmt */
32              int u_col_offset;      /* the offset of the current stmt */
33              int u_lineno_set;  /* boolean to indicate whether instr
34                                    has been generated with current lineno */
35      };
```

The `u_blocks` and `u_curblock` fields reference the basic blocks that make up the compiled code block. The `*u_ste` field refers to a symbol table entry for the code block the compiler is processing. The rest of the fields have pretty self-explanatory names. The compiler walks the different nodes that make up the code block and depending on whether a given node type begins a basic block or not, a basic block containing the nodes instructions is created or instructions for the node get added to an existing basic block. Node types that may begin a new basic block include but are not limited to the following.

1. Function nodes.
2. Jump targets.
3. Exception handlers.
4. Boolean operations, etc.

Basic blocks

The basic block is central to generating code objects. A basic block is a sequence of instructions that has one entry point but multiple exit points. Listing 3.19 is the definition of the `basic_block` data structure.

Listing 3.19: The `basicblock_` data strcuture

```
typedef struct basicblock_ {
    /* Each basicblock in a compilation unit is linked via b_list in the
    reverse order that the block are allocated.  b_list points to the next
    block, not to be confused with b_next, which is next by control flow. */
    struct basicblock_ *b_list;
    /* number of instructions used */
    int b_iused;
    /* length of instruction array (b_instr) */
    int b_ialloc;
    /* pointer to an array of instructions, initially NULL */
    struct instr *b_instr;
    /* If b_next is non-NULL, it is a pointer to the next
    block reached by normal control flow. */
    struct basicblock_ *b_next;
    /* b_seen is used to perform a DFS of basicblocks. */
    unsigned b_seen : 1;
    /* b_return is true if a RETURN_VALUE opcode is inserted. */
    unsigned b_return : 1;
    /* depth of stack upon entry of block, computed by stackdepth() */
    int b_startdepth;
    /* instruction offset for block, computed by assemble_jump_offsets() */
    int b_offset;
} basicblock;
```

The interesting fields here are `*b_list` that is a linked list of all basic blocks allocated during the compilation process, `*b_instr` which is an array of instructions within the basic block and `*b_next` which is the next basic flow reached by normal control flow execution. Each instruction has a structure shown in Listing 3.20 that holds a bytecode instruction. These bytecode instructions are in the `Include/opcode.h` header file.

Listing 3.20: The `instr` data strcuture

```
struct instr {
    unsigned i_jabs : 1;
    unsigned i_jrel : 1;
    unsigned char i_opcode;
    int i_oparg;
    struct basicblock_ *i_target; /* target block (if jump instruction) */
    int i_lineno;
};
```

To illustrate how the interpreter creates these basic blocks, we use the function in Listing 3.11. Compiling its AST shown in figure 3.2 into a CFG results in the graph similar to that in figure

3.4 - this shows only blocks with instructions. An inspection of this graph provides some intuition behind the basic blocks. Some basic blocks have a single entry point, but others have multiple exits. These blocks are described in more detail next.

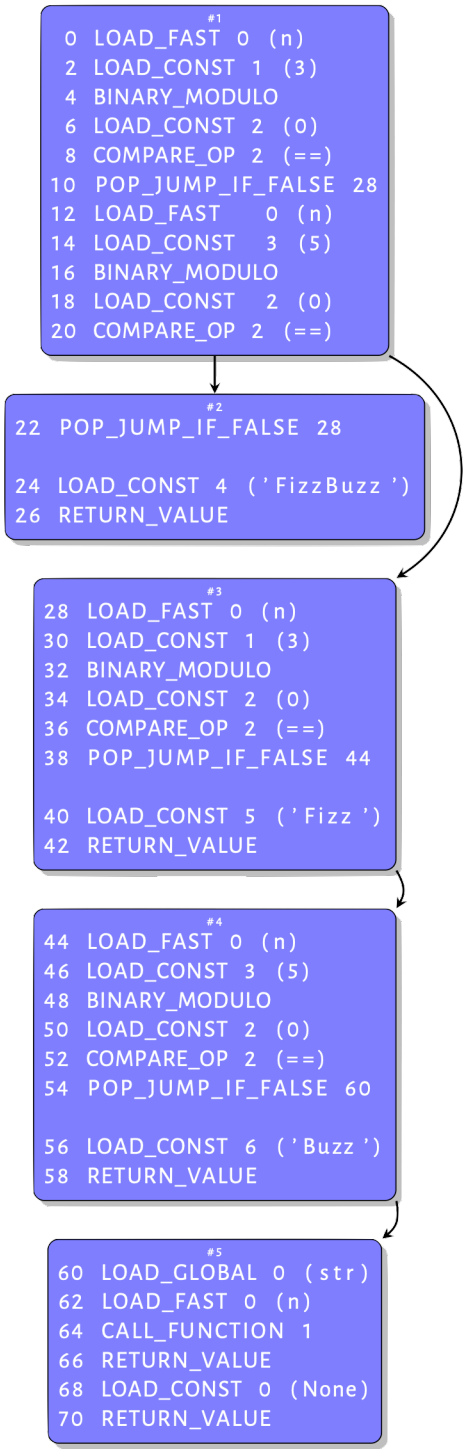


Figure 3.4: Control flow graph for the fizzbuzz function in listing 3.11.

Listing 3.21: Compiling an `if` statement

```
static int
compiler_if(struct compiler *c, stmt_ty s)
{
    basicblock *end, *next;
    int constant;
    assert(s->kind == If_kind);
    end = compiler_new_block(c);
    if (end == NULL)
        return 0;

    constant = expr_constant(c, s->v.If.test);
    /* constant = 0: "if 0"
     * constant = 1: "if 1", "if 2", ...
     * constant = -1: rest */
    if (constant == 0) {
        if (s->v.If.orelse)
            VISIT_SEQ(c, stmt, s->v.If.orelse);
    } else if (constant == 1) {
        VISIT_SEQ(c, stmt, s->v.If.body);
    } else {
        if (asdl_seq_LEN(s->v.If.orelse)) {
            next = compiler_new_block(c);
            if (next == NULL)
                return 0;
        }
        else
            next = end;
        VISIT(c, expr, s->v.If.test);
        ADDOP_JABS(c, POP_JUMP_IF_FALSE, next);
        VISIT_SEQ(c, stmt, s->v.If.body);
        if (asdl_seq_LEN(s->v.If.orelse)) {
            ADDOP_JREL(c, JUMP_FORWARD, end);
            compiler_use_next_block(c, next);
            VISIT_SEQ(c, stmt, s->v.If.orelse);
        }
    }
    compiler_use_next_block(c, end);
    return 1;
}
```

The body of the function in Listing 3.13 is an `if` statement as is visible in figure 3.2. The snippet

in Listing 3.21 is the function that compiles an `if` statement `AST` node into basic blocks. When this function compiles our example `if` statement node, the `else` statement on line 20 is executed. First, it creates a new basic block for an `else` node if such exists. Then, it visits the guard clause of the `if` statement node. What we have in the function in Listing 3.11 is interesting because the guard clause is a boolean expression which can trigger a jump during execution. Listing 3.22 is the function that compiles a boolean expression.

Listing 3.22: Compiling a boolean statement

```
1  static int compiler_boolop(struct compiler *c, expr_ty e)
2  {
3      basicblock *end;
4      int jumpi;
5      Py_ssize_t i, n;
6      asdl_seq *s;
7
8      assert(e->kind == BoolOp_kind);
9      if (e->v.BoolOp.op == And)
10         jumpi = JUMP_IF_FALSE_OR_POP;
11     else
12         jumpi = JUMP_IF_TRUE_OR_POP;
13     end = compiler_new_block(c);
14     if (end == NULL)
15         return 0;
16     s = e->v.BoolOp.values;
17     n = asdl_seq_LEN(s) - 1;
18     assert(n >= 0);
19     for (i = 0; i < n; ++i) {
20         VISIT(c, expr, (expr_ty)asdl_seq_GET(s, i));
21         ADDOP_JABS(c, jumpi, end);
22     }
23     VISIT(c, expr, (expr_ty)asdl_seq_GET(s, n));
24     compiler_use_next_block(c, end);
25     return 1;
26 }
```

The code up to the loop at line 20 is straight forward. In the loop, the compiler visits each expression, and after each visit, it adds a `jump`. This is because of the short circuit evaluation used by Python. It means that when a boolean operation such as an `AND` evaluates to `false`, the interpreter ignores the other expressions and performs a jump to continue execution. The compiler knows where to jump to if need be because the following instructions go into a new basic block - the use of `compiler_use_next_block` enforces this. So we have two blocks now. After visiting the test, the `compiler_if` function adds a jump instruction for the `if` statement, then compiles the body of the `if` statement. Recall that after visiting the boolean expression, the compiler created a new basic block. This block

contains the jump and instructions for body of the `if` statement, a simple return in this case. The target of this jump is the next block that will hold the `elif` arm of the if statement. The next step is to compile the `elif` component of the if statement but before this, the compiler calls the `compiler_use_next_block` function to activate the `next` block. The `orElse` arm is just another `if` statement, so the `compiler_if` function gets called again. This time around the test of the `if` is a compare operation. This is a single comparison, so there are no jumps involved and no new blocks, so the interpreter emits byte code for comparing values and returns to compile the body of the `if` statement. The same process continues for the last `orElse` arm resulting in the CFG in figure 3.3.

Figure 3.3 shows that the `fizzbuzz` function can exit block 1 in two ways. The first is via serial execution of all the instructions in block 1 then continuing in block 2. The other is via the `jump` instruction after the first compare operation. The target of this jump is block 3, but an executing code object knows nothing of basic blocks – the code object has a stream of bytecodes that are indexed with offsets. We have to provide the jump instructions with the offset into the bytecode instruction stream of the jump targets.

Assembling the basic blocks

The `assemble` function in `Python/compile.c` linearizes the CFG and creates the code object from the linearized CFG. It does so by computing the instruction offset for jump targets and using these as arguments to the jump instructions.

The `assembler` data structure in listing 3.23 plays a vital role during assembly of bytecode.

Listing 3.23: Calculating bytecode offsets

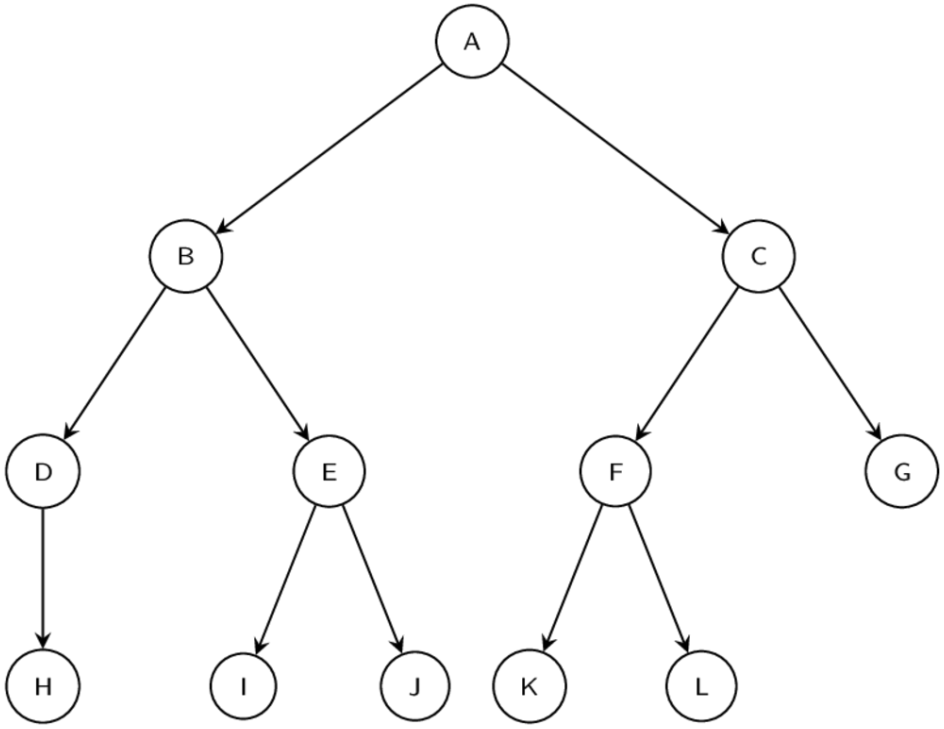
```
struct assembler {
   PyObject *a_bytecode;  /* string containing bytecode */
   int a_offset;              /* offset into bytecode */
   int a_nblocks;             /* number of reachable blocks */
   basicblock **a_postorder; /* list of blocks in dfs postorder */
   PyObject *a_lnotab;    /* string containing lnotab */
   int a_lnotab_off;      /* offset into lnotab */
   int a_lineno;              /* last lineno of emitted instruction */
   int a_lineno_off;      /* bytecode offset of last lineno */
};
```

The assembler data structure holds among other things the `a_postorder` array of basic blocks that contains the basic blocks in post order, `a_bytecode` contains the bytecode string generated and `a_nblocks` is the number of basic blocks within the code block. As we go through this stage of the process, you will see how these all come into play.

First, the assemble function, in this case, adds instructions for a `return None` statement since the last

statement of the function is not a `RETURN` statement - now you know why you can define methods without adding a `RETURN` statement. Next, it flattens the CFG using a *post-order* depth-first traversal - the post-order traversal visits the children of a node before visiting the node itself. The assembler data structure holds the flattened graph, `[block 5, block 4, block 3, block 2, block 1]`, in the `a_postorder` array for further processing. Next, it computes the instruction offsets and uses those as targets for the bytecode jump instructions. The `assemble_jump_offsets` function in listing 3.24 handles this.

Graph Traversal



In a post-order depth-first traversal of a graph, we recursively visit the left child node of the graph followed by the right child node of the graph then the node itself. In our graph in figure 3.45 when the graph is flattened using a post-order traversal, the order of the nodes is `H -> D -> I -> J -> E -> B -> K -> L -> F -> G -> C -> A`. This is in contrast to an pre-order traversal that produces `A -> B -> D -> H -> E -> I -> J -> C -> F -> K -> L -> G` or a in-order traversal that produces `H -> D -> B -> I -> E -> J -> A -> K -> L -> F -> C -> G`

The `assemble_jump_offsets` function in Listing 3.24 is relatively straightforward. In the `for . . . loop` at line 10, it computes the offset into the instruction stream for every instruction (akin to an array index). In the next `for . . . loop` at line 17, it uses the computed offsets as arguments to the jump instructions distinguishing between absolute and relative jumps.

Listing 3.24: Calculating bytecode offsets

```
static void assemble_jump_offsets(struct assembler *a, struct compiler *c){
    basicblock *b;
    int bsize, totsize, extended_arg_recompile;
    int i;

    /* Compute the size of each block and fixup jump args.
       Replace block pointer with position in bytecode. */
    do {
        totsize = 0;
        for (i = a->a_nblocks - 1; i >= 0; i--) {
            b = a->a_postorder[i];
            bsize = blocksize(b);
            b->b_offset = totsize;
            totsize += bsize;
        }
        extended_arg_recompile = 0;
        for (b = c->u->u_blocks; b != NULL; b = b->b_list) {
            bsize = b->b_offset;
            for (i = 0; i < b->b_iused; i++) {
                struct instr *instr = &b->b_instr[i];
                int isize = instrsize(instr->i_oparg);
                /* Relative jumps are computed relative to
                   the instruction pointer after fetching
                   the jump instruction.
                */
                bsize += isize;
                if (instr->i_jabs || instr->i_jrel) {
                    instr->i_oparg = instr->i_target->b_offset;
                    if (instr->i_jrel) {
                        instr->i_oparg -= bsize;
                    }
                    instr->i_oparg *= sizeof(_Py_CODEUNIT);
                    if (instrsize(instr->i_oparg) != isize) {
                        extended_arg_recompile = 1;
                    }
                }
            }
        }
    } while (extended_arg_recompile);
}
```

With instructions offset calculated and jump offsets assembled, the compiler emits instructions contained in the flattened graph in reverse post-order from the traversal. The reverse post order is a topological sorting of the CFG. This means for every edge from vertex u to vertex v, u comes before v in the sorting order. This is obvious; we want a node that jumps to another node to always come before that jump target. After emitting the bytecode instructions, the compiler creates code objects for each code block using the emitted bytecode and information contained in the symbol table. The generated code object is returned to the calling function marking the end of the compilation process.

4. Python Objects

In this chapter, we look at the Python objects and their implementation in the CPython virtual machine. This is central to understanding the Python virtual machine's internals. Most of the source referenced in this chapter is available in the `Include/` and `Objects/` directories. Unsurprisingly, the implementation of the Python object system is quite complex, so we try to avoid getting bogged down in the gory details of the `C` implementation. To kick this off, we start by looking at the `PyObject` structure - the workhorse of the Python object system.

4.1 PyObject

A cursory inspection of the CPython source code reveals the ubiquity of the `PyObject` structure. As we will see later on in this treatise, all the value stack objects used by the interpreter during evaluation are `PyObject`*s*. For want of a better term, we refer to this as the *superclass* of all Python objects. Values are never declared as `PyObject` but a pointer to any object can be cast to a `PyObject`. In layman's term, any object can be treated as a `PyObject` structure because the initial segment of all objects is a `PyObject` structure.

A word on `C` structs.

When we say "values are never declared as a `PyObject` but a pointer to any object can be cast to a `PyObject`", we refer to an implementation detail of the C programming language and how it interprets data at memory locations. C structs used to represent Python objects are just groups of bytes which we can interpret in any manner which we choose to. For example, a struct, `test`, may be composed of 5 `short` values each 2 bytes in size and summing up to 10 bytes. In C, given a reference to ten bytes, we can interpret those ten bytes as `test` struct composed of 5 short values regardless of whether the 10 bytes were defined as a `test` struct - however the output when you try to access the fields of the `struct` maybe gibberish. This means that given `n` bytes of data that represent a Python object where `n` is greater than the size of a PyObject, we can interpret the first n bytes as a PyObject.

Listing 4.0 is a definition of the `PyObject` structure. This structure is composed of several fields that must be filled for a value to be treated as an object.

Listing 4.0: PyObject definition

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

The `_PyObject_HEAD_EXTRA` when present is a `C` macro that defines fields that point to the previously allocated object and the next object, thus forming an implicit doubly-linked list of all live objects. The `ob_refcnt` field is for memory management, while the `*ob_type` is a pointer to the *type object* for the given object. This type determines what the data represents, what kind of data it contains, and the kind of operations the object supports. Take the snippet in Listing 4.1 for example, the name, `name`, points to a string object, and the type of the object is "str".

Listing 4.1: Variable declaration in python

```
>>> name = 'obi'
>>> type(name)
<class 'str'>
```

A valid question from here is *if the type field points to a `type` object then what does the `*ob_type` field of that `type` object point to?* The `ob_type` for a type object recursively refers to itself hence the saying that the type of a `type` is `type`.

A word on reference counting.

CPython uses reference counting for memory management. Reference counting is a simple method in which whenever a new reference to an object, such as binding a name to an object as in listing 4.1, is created the reference count of the object goes up by 1. The converse is true - whenever a reference to an object goes away (for example, using a `del` on name deletes the reference), the reference count is decremented by 1. When the reference count of an object hits zero, the VM can then deallocate that object. In the VM world, the `Py_INCREF` and `Py_DECREF` are used to increase and decrease the reference count of objects and they show up in a lot of code snippets that we discuss.

Types in the VM are implemented using the `_typeobject` data structure defined in the `Objects/Object.h` module. This is a C `struct` with fields for mostly functions or collections of functions filled in by each type. We look at this data structure next.

4.2 Dissecting Types

The `_typeobject` structure defined in `Include/Object.h` serves as the base structure of **all** Python types. The data structure defines a large number of fields that are mostly pointers to C functions that implement some functionality for a given type. Listing 4.2 is the `_typeobject` structure definition.

Listing 4.2: PyTypeObject definition

```
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 */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_asyn;

    reprfunc tp_repr;

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

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

    PyBufferProcs *tp_as_buffer;
    unsigned long tp_flags;
    const char *tp_doc; /* Documentation string */

    traverseproc tp_traverse;

    inquiry tp_clear;
    richcmpfunc tp_richcompare;
    Py_ssize_t tp_weaklistoffset;

    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    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;
    inquiry tp_is_gc;
    PyObject *tp_bases;
    PyObject *tp_mro;
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    unsigned int tp_version_tag;
    destructor tp_finalize;
} PyTypeObject;
```

The `PyObject_VAR_HEAD` field is an extension of the `PyObject` field discussed in the previous section; this extension adds an `ob_size` field for objects that have the notion of length. The Python C API documentation[1] contains a description of each of the fields in this object structure. The critical thing to note is that the fields in the structure each implement a part of the type's behavior. Most of these fields are part of what we can call an object interface or protocol; the types implement these functions but in a type-specific way. For example, `tp_hash` field is a reference to a hash function for a given type - this field could be left without a value in the case where instances of the type are not hashable; whatever function is in the `tp_hash` field gets invoked when the `hash` method is called on an instance of that type. The type object also has the field - `tp_methods` that references methods unique to that type. The `tp_new` slot refers to a function that creates new instances of the type and so on. Some of these fields, such as `tp_init`, are optional - not every type needs to run an initialization function, especially when the type is immutable, such as tuples. In contrast, other fields, such as `tp_new`, are compulsory.

Also, among these fields are fields for other Python protocols, such as the following.

1. Number protocol - A type implementing this protocol will have implementations for the `PyNumberMethods *tp_as_number` field. This field is a reference to a set of functions that implement arithmetic operations (this does not necessarily have to be on numbers). A type will support arithmetic operations with their corresponding implementations included in the `tp_as_number` set in the type's specific way. For example, the non-numeric `set` type has an entry into this field because it supports arithmetic operations such as `-`, `<=`, and so on.

[1] https://docs.python.org/3.6/c-api/typeobj.html

2. Sequence protocol - A type that implements this protocol will have a value in the `PySequenceMethods *tp_as_sequence` field. This value should provide that type with support for some sequence operations[2] such as `len`, `in` etc.
3. Mapping protocol - A type that implements this protocol will have a value in the `PyMappingMethods *tp_as_mapping`. This value enables such type to be treated like Python dictionaries using the dictionary syntax for setting and accessing key-value mappings.
4. Iterator protocol - A type that implements this protocol will have a value in the `getiterfunc tp_iter` and possibly the `iternextfunc tp_iternext` fields enabling instances of the type to be used like Python iterators.
5. Buffer protocol - A type implementing this protocol will have a value in the `PyBufferProcs *tp_as_buffer` field. These functions will enable access to the instances of the type as input/output buffers.

Next, we look at a few type objects as case studies of how the type object fields are populated.

4.3 Type Object Case Studies

The `tuple` type

We look at the `tuple` type to get a feel for how the fields of a type object are populated. We choose this because it is relatively easy to grok given the small size of the implementation - roughly a thousand plus lines of `C` including documentation strings. Listing 4.3 shows the implementation of the `tuple` type.

Listing 4.3: Tuple type definition

```
PyTypeObject PyTuple_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "tuple",
    sizeof(PyTupleObject) - sizeof(PyObject *),
    sizeof(PyObject *),
    (destructor)tupledealloc,                   /* tp_dealloc */
    0,                                          /* tp_print */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_reserved */
    (reprfunc)tuplerepr,                        /* tp_repr */
    0,                                          /* tp_as_number */
    &tuple_as_sequence,                         /* tp_as_sequence */
    &tuple_as_mapping,                          /* tp_as_mapping */
    (hashfunc)tuplehash,                        /* tp_hash */
```

[2] https://docs.python.org/3.6/library/stdtypes.html#sequence-types-list-tuple-range

```
    0,                                          /* tp_call */
    0,                                          /* tp_str */
    PyObject_GenericGetAttr,                    /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC |
        Py_TPFLAGS_BASETYPE | Py_TPFLAGS_TUPLE_SUBCLASS, /* tp_flags */
    tuple_doc,                                  /* tp_doc */
    (traverseproc)tupletraverse,                /* tp_traverse */
    0,                                          /* tp_clear */
    tuplerichcompare,                           /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    tuple_iter,                                 /* tp_iter */
    0,                                          /* tp_iternext */
    tuple_methods,                              /* tp_methods */
    0,                                          /* tp_members */
    0,                                          /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    0,                                          /* tp_dictoffset */
    0,                                          /* tp_init */
    0,                                          /* tp_alloc */
    tuple_new,                                  /* tp_new */
    PyObject_GC_Del,                            /* tp_free */
};
```

We look at the fields that are populated in this type.

1. `PyObject_VAR_HEAD` has been initialized with a type object - PyType_Type as the type. Recall that the type of a type object is Type. A look at the PyType_Type type object shows that the type of PyType_Type is itself.
2. `tp_name` is initialized to the name of the type - tuple.
3. `tp_basicsize` and `tp_itemsize` refer to the size of the tuple object and items contained in the tuple object, and this is filled in accordingly.
4. `tupledealloc` is a memory management function that handles the deallocation of memory when a tuple object is destroyed.
5. `tuplerepr` is the function invoked when the `repr` function is called with a tuple instance as an argument.
6. `tuple_as_sequence` is the set of sequence methods that the tuple implements. Recall that a tuple support `in`, `len` etc. sequence methods.

7. `tuple_as_mapping` is the set of mapping methods supported by a tuple - in this case, the keys are integer indexes only.
8. `tuplehash` is the function that is invoked whenever the hash of a tuple object is required. This comes into play when tuples are used as dictionary keys or in sets.
9. `PyObject_GenericGetAttr` is the generic function invoked when referencing attributes of a tuple object. We look at attribute referencing in subsequent sections.
10. `tuple_doc` is the documentation string for a tuple object.
11. `tupletraverse` is the traversal function for garbage collection of a tuple object. This function is used by the garbage collector to help in the detection of reference cycle[3].
12. `tuple_iter` is a method that gets invoked when the `iter` function is called on a tuple object. In this case, a completely different `tuple_iterator` type is returned so there is no implementation for the `tp_iternext` method.
13. `tuple_methods` are the actual methods of a tuple type.
14. `tuple_new` is the function invoked to create new instances of tuple type.
15. `PyObject_GC_Del` is another field that references a memory management function.

The additional fields with `0` values are empty because tuples do not require those functionalities. Take the `tp_init` field, for example, a tuple is an immutable type, so once created it cannot be changed, so there is no need for any initialization beyond what happens in the function referenced by `tp_new`; hence this field is left empty.

The `type` type

The other type we look at is the `type` type. This is the *metatype* for all built-in types and vanilla user-defined types (a user can define a new metatype) - notice how this type is used when initializing the `tuple` object in `PyVarObject_HEAD_INIT`. When discussing types, it is essential to distinguish between objects that have `type` as their type and objects with a user-defined type. This distinction comes very much to the fore when dealing with attribute referencing in objects.

This type defines methods used when working with other types, and the fields are similar to those from the previous section. When creating new types, as we will see in subsequent sections, this type is used.

The `object` type

Another necessary type is the `object` type; this is very similar to the `type` type. The object type is the root type for all user-defined types and provides some default values that fill in the type fields of a user-defined type. This is because user-defined types behave differently compared to types that have `type` as their type. As we will see in subsequent section, functions such as that for the attribute resolution algorithm provided by the `object` type differ significantly from those offered by the `type` type.

[3] https://docs.python.org/3.6/c-api/typeobj.html#c.PyTypeObject.tp_traverse.

4.4 Minting type instances

With an assumed firm understanding of the rudiments of types, we can progress onto one of the most fundamental functions of types, which is the creation of new instances. To fully understand the process of creating new type instances, it is important to remember that just as we differentiate between built-in types and user-defined types [4], the internal structure of both differs. The `tp_new` field is the cookie cutter for new type instances in Python. The documentation[5] for the `tp_new` slot as reproduced below gives a brilliant description of the function that should fill this slot.

> An optional pointer to an instance creation function. If this function is NULL for a particular type, that type cannot be called to create new instances; presumably, there is some other way to create instances, like a factory function. The function signature is
>
> ```
> PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
> ```
>
> The subtype argument is the type of the object being created; the `args` and `kwds` arguments are the positional and keyword arguments of the call to the type. Note that subtype doesn't have to equal the type whose tp_new function is called; it may be a subtype of that type (but not an unrelated type). The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.
>
> This field is inherited by subtypes but not by static types whose `tp_base` is `NULL` or `&PyBaseObject_Type`.

We will use the `tuple` type from the previous section as an example of a builtin type. The `tp_new` field of the tuple type references the - `tuple_new` method shown in Listing 4.4, which handles the creation of new tuple objects. A new tuple object is created by dereferencing and then invoking this function.

[4] Type is used rather than class for uniformity
[5] https://docs.python.org/3/c-api/typeobj.html#c.PyTypeObject.tp_new

Listing 4.4: tuple_new function for creating new tuple instances

```
static PyObject * tuple_new(PyTypeObject *type, PyObject *args,
                            PyObject *kwds){
    PyObject *arg = NULL;
    static char *kwlist[] = {"sequence", 0};

    if (type != &PyTuple_Type)
        return tuple_subtype_new(type, args, kwds);
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|O:tuple", kwlist, &arg))
        return NULL;

    if (arg == NULL)
        return PyTuple_New(0);
    else
        return PySequence_Tuple(arg);
}
```

Ignoring the first and second conditions for creating a tuple in Listing 4.4, we follow the third condition, `if (arg==NULL) return PyTuple_New(0)` down the rabbit hole to find out how this works. Overlooking the optimizations abound in the `PyTuple_New` function, the segment of the function that creates a new tuple object is the `op = PyObject_GC_NewVar(PyTupleObject, &PyTuple_Type, size)` call which allocates memory for an instance of the `PyTuple_Object` structure on the heap. This is where a difference between internal representation of built-in types and user-defined types comes to the fore - instances of built-ins like tuple are actually `C` structures. So what does this C struct backing a tuple object look like? It is found in the `Include/tupleobject.h` as the `PyTupleObject` typedef, and this is shown in Listing 4.5 for convenience.

Listing 4.5: PyTuple_Object definition

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject *ob_item[1];

    /* ob_item contains space for 'ob_size' elements.
    * Items must typically not be NULL, except during construction when
    * the tuple is not yet visible outside the function that builds it.
    */
} PyTupleObject;
```

The `PyTupleObject` is defined as a struct having a `PyObject_VAR_HEAD` and an array of PyObject pointers - `ob_items`. This leads to a very efficient implementation as opposed to representing the instance using Python data structures.

Recall that an object is a collection of methods and data. The `PyTupleObject` in this case provides space to hold the actual data that each tuple object contains so we can have multiple instances of `PyTupleObject` allocated on the heap but these will all reference the single `PyTuple_Type` type that provides the methods that can operate on this data.

Now consider a user-defined class such as in LIsting 4.6.

Listing 4.6: User defined class

```
class Test:
    pass
```

The `Test` type, as you would expect, is an object of instance `Type`. To create an instance of the `Test` type, the `Test` type is called as such - `Test()`. As always, we can go down the rabbit hole to convince ourselves of what happens when the type object is called. The `Type` type has a function reference - `type_call` that fills the `tp_call` field, and this is dereferenced whenever the *call* notation is used on an instance of `Type`. A snippet of the `type_call` the function implementation is shown in listing 4.7.

Listing 4.7: A snippet of type_call function definition

```
    ...
    obj = type->tp_new(type, args, kwds);
    obj = _Py_CheckFunctionResult((PyObject*)type, obj, NULL);
    if (obj == NULL)
        return NULL;

    /* Ugly exception: when the call was type(something),
       don't call tp_init on the result. */
    if (type == &PyType_Type &&
        PyTuple_Check(args) && PyTuple_GET_SIZE(args) == 1 &&
        (kwds == NULL ||
         (PyDict_Check(kwds) && PyDict_Size(kwds) == 0)))
        return obj;

    /* If the returned object is not an instance of type,
       it won't be initialized. */
    if (!PyType_IsSubtype(Py_TYPE(obj), type))
        return obj;

    type = Py_TYPE(obj);
    if (type->tp_init != NULL) {
        int res = type->tp_init(obj, args, kwds);
        if (res < 0) {
            assert(PyErr_Occurred());
            Py_DECREF(obj);
```

```
26              obj = NULL;
27          }
28          else {
29              assert(!PyErr_Occurred());
30          }
31      }
32      return obj;
```

In Listing 4.7, we see that when a `Type` object instance is called, the function referenced by the `tp_new` field is invoked to create a new instance of that type. The `tp_init` function is also called if it exists to initialize the new instance. This process explains builtin types because, after all, they have their own `tp_new` and `tp_init` functions defined already, but what about user-defined types? Most times, a user does not define a `__new__` function for a new type (when defined, this will go into the `tp_new` field during class creation). The answer to this also lies with the `type_new` function that fills the `tp_new` field of the `Type`. When creating a user-defined type, such as `Test`, the `type_new` function checks for the presence of base types (supertypes/classes) and when there are none, the `PyBaseObject_Type` type is added as a default base type, as shown in listing 4.8.

Listing 4.8: Snippet showing how the `PyBaseObject_Type` is added to list of bases

```
    ...
    if (nbases == 0) {
    bases = PyTuple_Pack(1, &PyBaseObject_Type);
    if (bases == NULL)
        goto error;
    nbases = 1;
}
...
```

This default base type defined in the `Objects/typeobject.c` module contains some default values for the various fields. Among these defaults are values for the `tp_new` and `tp_init` fields. These are the values that get called by the interpreter for a user-defined type. In the case where the user-defined type implements its methods such as `__init__`, `__new__` etc., these values are called rather than those of the `PyBaseObject_Type` type.

One may notice that we have not mentioned any object structures like the tuple object structure, `tupleobject`, and ask - *if no object structures are defined for a user-defined class, how are object instances handled and where do objects attributes that do not map to slots in the type reside?* This has to do with the `tp_dictoffset` field - a numeric field in type object. Instances are created as *PyObjects*, however, when this offset value is non-zero in the instance type, it specifies the offset of the instance attribute dictionary from the instance (the PyObject) itself as shown in figure 4.0 so for an instance of a `Person` type, the attribute dictionary can be assessed by adding this offset value to the origin of the *PyObject* memory location.

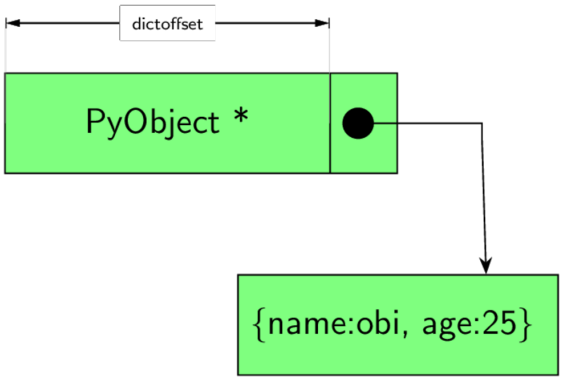


Figure 4.0: How instances of user-defined types are structured.

For example, if an instance *PyObject* is at `0x10` and the offset is `16` then the instance dictionary that contains instance attributes is found at `0x10 + 16`. This is not the only way instances store their attributes, as we will see in the following section.

4.5 Objects and their attributes

Types and their attributes *(variables and methods)* are central to object-oriented programming. Conventionally, types and instances store their attributes using a `dict` data structure, but this is not the full story in cases of instances that have the `__slots__` attribute defined. This `dict` data structure resides in one of two places, depending on the type of the object, as was mentioned in the previous section.

1. For objects with a type of `Type`, the `tp_dict` slot of type structure is a pointer to a `dict` that contains values, variables, and methods for that type. In the more conventional sense, we say the `tp_dict` field of the type object data structure is a pointer to the `type` or `class` `dict`.
2. For objects with user-defined types, that `dict` data structure when present is located just after the `PyObject` structure that represents the object. The `tp_dictoffset` value of the type of the object gives the offset from the start of an object to this instance `dict` contains the instance attributes.

Performing a simple dictionary access to obtain attributes looks simpler than it actually is. Infact, searching for attributes is way more involved than just checking `tp_dict` value for instance of `Type` or the `dict` at `tp_dictoffset` for instances of user-defined types. To get a full understanding, we have to discuss the *descriptor protocol* - a protocol at the heart of attribute referencing in Python.

The Descriptor HowTo Guide[6] is an excellent introduction to descriptors, but the following section provides a cursory description of descriptors. Simply put, a *descriptor* is an **object** that implements the `__get__`, `__set__` or `__delete__` special methods of the descriptor protocol. Listing 4.9 is the signature for each of these methods in Python.

[6] https://docs.python.org/3/howto/descriptor.html

Listing 4.9: The Descriptor protocol methods

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Objects implementing only the `__get__` method are non-data descriptors so they can only be read from after initialization. In contrast, objects implementing the `__get__` and `__set__` are data descriptors meaning that such descriptor objects are writeable. We are interested in the application of descriptors to object attribute representation. The `TypedAttribute` descriptor in listing 4.10 is an example of a descriptor used to represent an object attribute.

Listing 4.10: A simple descriptor for type checking attribute values

```
class TypedAttribute:

    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)

    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")
```

The `TypedAttribute` descriptor class enforces rudimentary type checking for any class' attribute that it represents. It is important to note that descriptors are useful in this kind of case only when defined at the class level rather than instance-level, i.e., in `__init__` method, as shown in listing 4.11.

Listing 4.11: Type checking on instance attributes using TypedAttribute descriptor

```
class Account:
    name = TypedAttribute("name",str)
    balance = TypedAttribute("balance",int, 42)

    def name_balance_str(self):
        return str(self.name) + str(self.balance)

>> acct = Account()
>> acct.name = "obi"
>> acct.balance = 1234
>> acct.balance
1234
>> acct.name
obi
# trying to assign a string to number fails
>> acct.balance = '1234'
TypeError: Must be a <type 'int'>
```

If one thinks carefully about it, it only makes sense for this kind of descriptor to be defined at the type level because if defined at the instance the level, then any assignment to the attribute will overwrite the descriptor.

A review of the Python VM source code shows the importance of descriptors. Descriptors provide the mechanism behind properties, static methods, class methods, and a host of other functionality in Python. Listing 4.12, the algorithm for resolving an attribute from an instance,`b`, of a user-defined type, is a concrete illustration of the importance of descriptors.

Listing 4.12: Algorithm to find a referenced attribute in an instance of a user-defined type

```
1. Search type(b).__dict__ for the attribute by name. If the attribute is present an\
d a data descriptor, then return the result of calling the descriptor's __get__ meth\
od. If the name is absent, then all base classes in the *MRO* of type(b) are searche\
d in the same way.
2. Search b.__dict__, and if the attribute is present, return it.
3. if the name from 1 is a non-data descriptor return the value of calling
__get__ ,
4. If the name is not found,  call __getattr__() if provided by the user-defined typ\
e otherwise raise an `AttributeError`.
```

The algorithm in Listing 4.12 shows that during attribute referencing we first check for descriptor objects; it also illustrates how the `TypedAttribute` descriptor can represent an attribute of an object - whenever an attribute is referenced such as `b.name` the VM searches the `Account` type object for the

attribute, and in this case, it finds a `TypedAttribute` descriptor; the VM then calls `__get__` method of the descriptor accordingly. The `TypedAttribute` example illustrates a descriptor but is somewhat contrived; to get a real touch of how important descriptors are to the core of the language, we consider some examples that show how they are applied.

Do note that the attribute reference algorithm in listing 4.12 differs from the algorithm used when referencing an attribute whose type is `type`. Listing 4.3 shows the algorithm for such.

Listing 4.13: Algorithm to find a referenced attribute in a type

```
1. Search type(type).__dict__  for the attribute by name. If the name is present and\
 it is a data descriptor, return the result of calling the descriptor's __get__ meth\
od. If the name is absent, then all base classes in the *MRO* of type(type) are sear\
ched in the same way.
2. Search type.__dict__ and all its bases for the attribute by name.
If the name present and it is a descriptor, then invoke its `__get__` method, otherw\
ise return the value.
3. If a value was found in (1) and it is a non-data descriptor, then return the valu\
e from invoking its __get__ function.
4. if the value found in (1) is not a descriptor, then return it.
```

Examples of Attribute Referencing with Descriptors inside the VM

Consider the `type` data structure discussed earlier in this chapter. The `tp_descr_get` and `tp_descr_set` fields in a type data structure can be filled in by any type instance to satisfy the descriptor protocol. A function object is a perfect place to show how this works.

Given the type definition, `Account` from listing 4.11, consider what happens when we reference the method, `name_balance_str`, from the class as such - `Account.name_balance_str` and when we reference the same method from an instance as shown in listing 4.14.

Listing 4.14: Illustrating bound and unbound functions

```
>> a = Account()
>> a.name_balance_str
<bound method Account.name_balance_str of <__main__.Account object at
0x102a0ae10>>

>> Account.name_balance_str
<function Account.name_balance_str at 0x102a2b840>
```

Looking at the snippet from listing 4.14, although we seem to reference the same attribute, the actual objects returned are different in value and type. When referenced from the account type, the returned value is a `function` type, but when referenced from an instance of the account type, the result is a `bound method` type. This is possible because functions are descriptors too. Listing 4.15 is the definition of a function object type.

Listing 4.15: Function type object definition

```
PyTypeObject PyFunction_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "function",
    sizeof(PyFunctionObject),
    0,
    (destructor)func_dealloc,                   /* tp_dealloc */
    0,                                          /* tp_print */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_reserved */
    (reprfunc)func_repr,                        /* tp_repr */
    0,                                          /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    0,                                          /* tp_hash */
    function_call,                              /* tp_call */
    0,                                          /* tp_str */
    0,                                          /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC,/* tp_flags */
    func_doc,                                   /* tp_doc */
    (traverseproc)func_traverse,                /* tp_traverse */
    0,                                          /* tp_clear */
    0,                                          /* tp_richcompare */
    offsetof(PyFunctionObject, func_weakreflist), /* tp_weaklistoffset */
    0,                                          /* tp_iter */
    0,                                          /* tp_iternext */
    0,                                          /* tp_methods */
    func_memberlist,                            /* tp_members */
    func_getsetlist,                            /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    func_descr_get,                             /* tp_descr_get */
    0,                                          /* tp_descr_set */
    offsetof(PyFunctionObject, func_dict),      /* tp_dictoffset */
    0,                                          /* tp_init */
    0,                                          /* tp_alloc */
    func_new,                                   /* tp_new */
};
```

The function object fills in the `tp_descr_get` field with a `func_descr_get` function thus instances of the `function` type are non-data descriptors. Listing 4.16 shows the implementation of the `funct_descr_get` method.

Listing 4.16: Function type object definition

```
static PyObject * func_descr_get(PyObject *func, PyObject *obj, PyObject *type){
    if (obj == Py_None || obj == NULL) {
        Py_INCREF(func);
        return func;
    }
    return PyMethod_New(func, obj);
}
```

The `func_descr_get` can be invoked during either type attribute resolution or instance attribute resolution, as described in the previous section. When invoked from a type, the call to the `func_descr_get` is as such `local_get(attribute, (PyObject *)NULL,(PyObject *)type)` while when invoked from an attribute reference of an instance of a user-defined type, the call signature is `f(descr, obj, (PyObject *)Py_TYPE(obj))`. Going over the implementation for `func_descr_get` in listing 4.16, we see that if the instance is `NULL`, then the function itself is returned while if an instance is passed in to the call, a new method object is created using the function and the instance. This sums up how Python can return a different type for the same function reference using a descriptor.

The `self` argument is the first parameter to any instance method definition, but why is it left out when such methods are called? The reality is that instance methods do take the instance (called `self` by convention) as the first argument - this is just transparent to the caller. A call such as `b.name_balance_str()` is the same thing as `type(b).name_balance_str(b)`. The reason we can invoke `b.name_balance_str()` is that the value returned by `b.name_balance_str` is a method object which is a thin wrapper around the `name_balance_str` with the instance already bound to the method instance. So when we make a call such as `b.name_balance_str()` the method uses the bound instance as an argument to the wrapped function hiding this detail from us.

In another instance of the importance of descriptors, consider the snippet in Listing 4.17 which shows the result of accessing the `__dict__` attribute from both an instance of the built-in `type` and an instance of a user-defined type.

Listing 4.17: Accessing the `__dict__` attribute from an instance of the built-in type and an instance of a user-defined type

```
class A:
    pass

>>> A.__dict__
mappingproxy({'__module__': '__main__', '__doc__': None, '__weakref__': <attribute '\
__weakref__' of 'A' objects>, '__dict__': <attribute '__dict__' of 'A' objects>})
>>> i = A()
>>> i.__dict__
{}
>>> A.__dict__['name'] = 1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> i.__dict__['name'] = 2
>>> i.__dict__
{'name': 2}
>>>
```

Observe from listing 4.17 that both objects do not return the vanilla dictionary type when the `__dict__` attribute is referenced. The type object seems to return an immutable mapping proxy that we cannot even assign. In contrast, the instance of `type` returns a vanilla dictionary mapping that supports all the usual dictionary functions. So it seems that attribute referencing is done differently for these objects. Recall the algorithm described for attribute search from a couple of sections back. The first step is to search the `__dict__` of the type of the object for the attribute, so we go ahead and do this for both objects in listing 4.18.

Listing 4.18: Checking for `__dict__` in type of objects

```
>>> type(type.__dict__['__dict__']) # type of A is type
<class 'getset_descriptor'>
    type(A.__dict__['__dict__'])
<class 'getset_descriptor'>
```

We see that the `__dict__` attribute is represented by data descriptors for both objects, and that is why we can get different object types. We would like to find out what happens under the covers for this descriptor, just as we did in the *functions* and *bound methods*. A good place to start is the `Objects/typeobject.c` module and the definition for the `type` type object. An interesting field is the `tp_getset` field that contains an array of `C` structs (`PyGetSetDef` values) shown in listing 4.19. This is the collection of values that will be represented by descriptors in `type's` type `__dict__` attribute - the `__dict__` attribute is the mapping referred to by the `tp_dict` slot of the type object points.

Listing 4.19: Checking for `__dict__` in type of objects

```
static PyGetSetDef type_getsets[] = {
    {"__name__", (getter)type_name, (setter)type_set_name, NULL},
    {"__qualname__", (getter)type_qualname, (setter)type_set_qualname, NULL},
    {"__bases__", (getter)type_get_bases, (setter)type_set_bases, NULL},
    {"__module__", (getter)type_module, (setter)type_set_module, NULL},
    {"__abstractmethods__", (getter)type_abstractmethods,
    (setter)type_set_abstractmethods, NULL},
    {"__dict__",  (getter)type_dict,  NULL, NULL},
    {"__doc__", (getter)type_get_doc, (setter)type_set_doc, NULL},
    {"__text_signature__", (getter)type_get_text_signature, NULL, NULL},
    {0}
};
```

These values are not the only ones represented by descriptors in the type *dict*; there are other values such as the `tp_members` and `tp_methods` values which have descriptors created and insert into the `tp_dict` during type initialization. The insertion of these values into the *dict* happens when the `PyType_Ready` function is called on the type. As part of the `PyType_Ready` function initialization process, descriptor objects are created for each entry in the `type_getsets` and then added into the `tp_dict` mapping - the `add_getset` function in the `Objects/typeobject.c` handles this.

Returning to our `__dict__`, attribute, we know that after initialization of the type, the `__dict__`-attribute exists in the `tp_dict` field of the type, so let's see what the *getter* function of this descriptor does. The *getter* function is the `type_dict` function shown in listing 4.20.

Listing 4.20: Getter function for an instance of `type`

```
static PyObject * type_dict(PyTypeObject *type, void *context){
    if (type->tp_dict == NULL) {
        Py_INCREF(Py_None);
        return Py_None;
    }
    return PyDictProxy_New(type->tp_dict);
}
```

The `tp_getattro` field points to the function that is the first port of call for getting attributes for any object. For the type object, it points to the `type_getattro` function. This method, in turn, implements the attribute search algorithm as described in listing 4.13. The function invoked by the descriptor found in the type dict for the `__dict__` attribute is the `type_dict` function given in listing 4.20, and it is pretty easy to understand. The return value is of interest to us here; it is a dictionary proxy to the actual dictionary that holds the type attribute; this explains the `mappingproxy` type that is returned when we query the `__dict__` attribute of a type object.

So what about the instance of A, a user-defined type, how is the `__dict__` attribute resolved? Now recall that `A` is an object of type `type` so we go hunting in the `Object/typeobject.c` module to see how new type instances are created. The `tp_new` slot of the `PyType_Type` contains the `type_new` function that handles the creation of new type objects. Perusing through all the type creation code in the function, one stumbles on the snippet in listing 4.21.

Listing 4.21: Setting `tp_getset` field for user defined type

```
if (type->tp_weaklistoffset && type->tp_dictoffset)
    type->tp_getset = subtype_getsets_full;
else if (type->tp_weaklistoffset && !type->tp_dictoffset)
    type->tp_getset = subtype_getsets_weakref_only;
else if (!type->tp_weaklistoffset && type->tp_dictoffset)
    type->tp_getset = subtype_getsets_dict_only;
else
    type->tp_getset = NULL;
```

Assuming the first conditional is true as the `tp_getset` field is filled with the value shown in Listing 4.22.

Listing 4.22: The `getset` values for instance of `type`

```
static PyGetSetDef subtype_getsets_full[] = {
    {"__dict__", subtype_dict, subtype_setdict,
    PyDoc_STR("dictionary for instance variables (if defined)")},
    {"__weakref__", subtype_getweakref, NULL,
    PyDoc_STR("list of weak references to the object (if defined)")},
    {0}
};
```

When `(*tp->tp_getattro)(v, name)` is invoked, the `tp_getattro` field which contains a pointer to the `PyObject_GenericGetAttr` is called. This function is responsible for implementing the attribute search algorithm for a user-defined types. In the case of the `__dict__` attribute, the descriptor is found in the object type's `dict` and the `__get__` function of the descriptor is the `subtype_dict` function defined for the `__dict__` attribute from listing 4.21. The `subtype_dict` *getter* function is shown in listing 4.23.

Listing 4.23: The *getter* function for `__dict__` attribute of a user-defined type

```
static PyObject * subtype_dict(PyObject *obj, void *context){
    PyTypeObject *base;

    base = get_builtin_base_with_dict(Py_TYPE(obj));
    if (base != NULL) {
        descrgetfunc func;
        PyObject *descr = get_dict_descriptor(base);
        if (descr == NULL) {
            raise_dict_descr_error(obj);
            return NULL;
        }
        func = Py_TYPE(descr)->tp_descr_get;
        if (func == NULL) {
            raise_dict_descr_error(obj);
            return NULL;
        }
        return func(descr, obj, (PyObject *)(Py_TYPE(obj)));
    }
    return PyObject_GenericGetDict(obj, context);
}
```

The `get_builtin_base_with_dict` returns a value when the object instance is in an inheritance hierarchy, so ignoring that for this instance is appropriate. The `PyObject_GenericGetDict` object is invoked. Listing 4.24 shows the `PyObject_GenericGetDict` and an associated helper that fetches the instance dict. The actual *get the dict* function is the `_PyObject_GetDictPtr` function that queries the object for its `dictoffset` and uses that to compute the address of the instance `dict`. In a situation where this function returns a null value, `PyObject_GenericGetDict` can return a new dict to the calling function.

Listing 4.24: Fetching dict attribute of an instance of a user defined type

```
PyObject * PyObject_GenericGetDict(PyObject *obj, void *context){
    PyObject *dict, **dictptr = _PyObject_GetDictPtr(obj);
    if (dictptr == NULL) {
        PyErr_SetString(PyExc_AttributeError,
                        "This object has no __dict__");
        return NULL;
    }
    dict = *dictptr;
    if (dict == NULL) {
        PyTypeObject *tp = Py_TYPE(obj);
        if ((tp->tp_flags & Py_TPFLAGS_HEAPTYPE) && CACHED_KEYS(tp)) {
```

```
            DK_INCREF(CACHED_KEYS(tp));
            *dictptr = dict = new_dict_with_shared_keys(CACHED_KEYS(tp));
        }
        else {
            *dictptr = dict = PyDict_New();
        }
    }
    Py_XINCREF(dict);
    return dict;
}

PyObject ** _PyObject_GetDictPtr(PyObject *obj){
    Py_ssize_t dictoffset;
    PyTypeObject *tp = Py_TYPE(obj);

    dictoffset = tp->tp_dictoffset;
    if (dictoffset == 0)
        return NULL;
    if (dictoffset < 0) {
        Py_ssize_t tsize;
        size_t size;

        tsize = ((PyVarObject *)obj)->ob_size;
        if (tsize < 0)
            tsize = -tsize;
        size = _PyObject_VAR_SIZE(tp, tsize);

        dictoffset += (long)size;
        assert(dictoffset > 0);
        assert(dictoffset % SIZEOF_VOID_P == 0);
    }
    return (PyObject **) ((char *)obj + dictoffset);
}
```

This explanation succinctly sums up how the Python VM uses descriptors to implement type-dependent custom attribute access depending on types. Descriptors are pervasive in the VM; `__slots__`, static and class methods, properties are just some further examples of language features that are made possible by the use of descriptors.

4.6 Method Resolution Order (MRO)

We have mentioned *MRO* when discussing attribute referencing without discussing it much so in this section, we go into a bit more detail on *MRO*. Types can belong to a multiple inheritance hierarchy, so there is a need for some kind of order defining how to search for methods when a type inherits from multiple classes; this order which is referred to as |*Method Resolution Order (MRO)* is also actually used when searching for other non-method attributes as we saw in the algorithm for attribute reference resolution. The article, Python 2.3 Method Resolution order[7], is an excellent and easy to read documentation of the method resolution algorithm used in Python; a summary of the main points are reproduced here.

Python uses the C3[8] algorithm for building the *method resolution order* (also referred to as *linearization* here) when a type inherits from multiple base types. Listing 4.25 shows some notations used in explaining this algorithm.

```
C1 C2 ... CN denotes the list of classes  [C1, C2, C3 .., CN]

The head of the list is its first element: head = C1

The tail is the rest of the list: tail = C2 ... CN.

C + (C1 C2 ... CN) = C C1 C2 ... CN denotes the sum of the lists [C] +
[C1, C2, ... ,CN].
```

Consider a type `C` in a multiple inheritance hierarchy, with `C` inheriting from the base types `B1, B2, ... , BN`, the linearization of `C` is the sum of `C` plus the merge of the linearizations of the parents and the list of the parents - `L[C(B1 ... BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)`. The linearization of the `object` type which has no parents is trivial - `L[object] = object`. The `merge` operation is calculated according to the following algorithm[9]:

> take the head of the first list, i.e., L[B1][0]; if this head is not in the tail of any of the other lists, then add it to the linearization of C and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head. Then repeat the operation until all the classes have been removed, or it is impossible to find good heads. In this case, it is impossible to construct the merge; Python 2.3 will refuse to create the class C and will raise an exception.

Some type hierarchies cannot be linearized using this algorithm, and in such cases, the VM throws an error and does not create such hierarchies.

[7] https://www.python.org/download/releases/2.3/mro/
[8] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.3910&rep=rep1&type=pdf
[9] https://www.python.org/download/releases/2.3/mro/

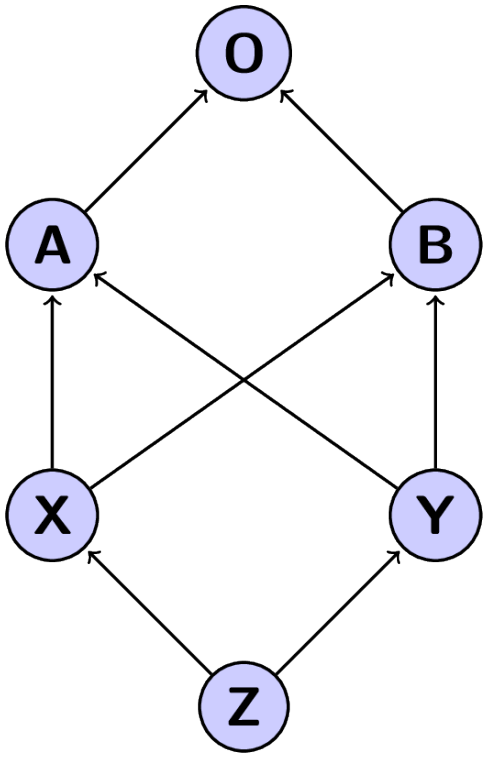


Figure 4.1: A simple multiple inheritance hierarchy.

Assuming we have an inheritance hierarchy such as that shown in figure 4.1, the algorithm for creating the *MRO* would proceed as follows starting from the top of the hierarchy with `O`, `A`, and `B`. The linearizations of `O`, `A` and `B` are trivial:

Listing 4.26: Calculating linearization for types `O`, `A` and `B` from figure 4.1

```
L[O] = O
L[A] = A O
L[B] = B O
```

The linearization of `X` can be computed as `L[X] = X + merge(AO, BO, AB)`

`A` is a good head, so it is added to the linearization, and we are left to compute `merge(O, BO, B)`. `O` is not a good head because it is in the tail of `BO`, so we move to the next sequence. `B` is a good head, so we add it to the linearization, and we are left to compute `merge(O, O)`, which evaluates to `O`. The resulting linearization of `X` - `L[X] = X A B O`.

Like the procedure from above, the linearization for `Y` is computed, as shown in Listing 4.27:

Listing 4.27: Calculating linearization for type `Y` from figure 4.1

```
L[Y] = Y + merge(AO,BO,AB)
     = Y + A + merge(O,BO,B)
     = Y + A + B + merge(O,O)
     = Y A B O
```

With linearizations for `X` and `Y` computed, we can compute that for `Z` as shown in listing 4.28.

Listing 4.28: Calculating linearization for type `Z` from figure 4.1

```
L[Z] = Z + merge(XABO,YABO,XY)
     = Z + X + merge(ABO,YABO,Y)
     = Z + X + Y + merge(ABO,ABO)
     = Z + X + Y + A + merge(BO,BO)
     = Z + X + Y + A + B + merge(O,O)
     = Z X Y A B O
```

5. Code Objects

Code objects are essential building blocks of the Python virtual machine. Code objects encapsulate the Python virtual machine's bytecode; we may call the bytecode the assembly language of the Python virtual machine.

Code objects, as the name suggests, represent compiled executable Python code. We had come across code objects before when we discussed Python source compilation. The compilation process maps each code block to a code object. As described in the brilliant Python documentation[1]:

> A Python program is constructed from code blocks. A block is 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. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified as a command-line argument to the interpreter) is a code block. A script command (a command specified on the interpreter command line with the '-c' option) is a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block.

The code object contains runnable bytecode instructions that alter the state of the Python VM when run. Given a function, we can access its code object using the `__code__` attribute as in the following snippet.

Listing 5.1: Function code objects

```
    def return_author_name():
        return "obi Ike-Nwosu"

    >>> return_author_name.__code__
    <code object return_author_name at 0x102279270, file "<stdin>", line 1>
```

For other code blocks, one can obtain the code objects for that code block by compiling such code. The compile[2] function provides a facility for this in the Python interpreter. The code objects possess several fields that are used by the interpreter loop when executing and we look at some of these in the following sections.

5.1 Exploring code objects

An excellent way to start with code objects is to compile a simple function and inspect the resulting code object. We use the simple `fizzbuzz` function shown in Listing 5.2 as a guinea pig.

[1] https://docs.python.org/3/reference/executionmodel.html
[2] https://docs.python.org/3.6/library/functions.html#compile

Listing 5.2: Function code objects attributes of Fizzbuzz function

```
    co_argcount = 1
    co_cellvars = ()
    co_code = b'|\x00d\x01\x16\x00d\x02k\x02r\x1e|\x00d\x03\x16\x00d\x02k\x02r\x1ed\\
x04S\x00n,|\x00d\x01\x16\x00d\x02k\x02r0d\x05S\x00n\x1a|\x00d\x03\x16\x00d\x02k\x02r\
Bd\x06S\x00n\x08t\x00|\x00\x83\x01S\x00d\x00S\x00'
    co_consts = (None, 3, 0, 5, 'FizzBuzz', 'Fizz', 'Buzz')
    co_filename = /Users/c4obi/projects/python_source/cpython/fizzbuzz.py
    co_firstlineno = 6
    co_flags = 67
    co_freevars = ()
    co_kwonlyargcount = 0
    co_lnotab = b'\x00\x01\x18\x01\x06\x01\x0c\x01\x06\x01\x0c\x01\x06\x02'
    co_name = fizzbuzz
    co_names = ('str',)
    co_nlocals = 1
    co_stacksize = 2
    co_varnames = ('n',)
```

The fields shown are almost self-explanatory except for the `co_lnotab` and `co_code` fields that seem to contain gibberish.

1. *co_argcount*: This is the number of arguments to a code block and has a value only for function code blocks. The value is set to the count of the argument set of the code block's `AST` during the compilation process. The evaluation loop makes use of these variables during the set-up for code evaluation to carry out sanity checks such as checks that all arguments are present and for storing locals.
2. *co_code*: This holds the sequence of bytecode instructions executed by the evaluation loop. Each of these bytecode instruction sequences is composed of an `opcode` and an `oparg` - arguments to the opcode where it exists. For example, `co.co_code[0]` returns the first byte of the instruction, `124` that maps to a Python `LOAD_FAST` opcode.
3. *co_consts*: This field is a list of constants like string literals and numeric values contained within the code object. The example from above shows the content of this field for the `fizzbuzz` function. The values included in this list are integral to code execution as they are the values referenced by the `LOAD_CONST` opcode. The operand argument to a bytecode instruction such as the `LOAD_CONST` is the index into this list of constants. Consider the co_consts value of `(None, 3, 0, 5, 'FizzBuzz', 'Fizz', 'Buzz')` for the `FizzBuzz` function and contrast with the disassembled code object below.

Listing 5.3: Cross section of bytecode instructions for Fizzbuzz function

```
           0 LOAD_BUILD_CLASS
           2 LOAD_CONST               0 (<code object Test at 0x101a02810, file "fiz\
zbuzz.py", line 1>)
           4 LOAD_CONST               1 ('Test')
           6 MAKE_FUNCTION            0
           8 LOAD_CONST               1 ('Test')
          10 CALL_FUNCTION            2
          12 STORE_NAME               0 (Test)

          ...

          66 LOAD_GLOBAL              0 (str)
          68 LOAD_FAST                0 (n)
          70 CALL_FUNCTION            1
          72 RETURN_VALUE
          74 LOAD_CONST               0 (None)
          76 RETURN_VALUE
```

Recall that during the compilation process, a `return None` is added if there is no `return` statement at the end of a function so we can tell that the bytecode instruction at offset 74 is a `LOAD_CONST` for a `None` value. The opcode argument is a 0, and we can see that the `None` value has an index of `0` in the constants list from where the `LOAD_CONST` instruction loads it.

4. *co_filename*: This field, as the name suggests, contains the name of the file that contains the code object's source code from which the code object.
5. *co_firstlineno*: This gives the line number on which the source for the code object begins. This value plays quite an essential role during activities such as debugging code.
6. *co_flags*: This field indicates the kind of code object. For example, if the code object is that of a coroutine, the flag is set to `0x0080`. Other flags such as `CO_NESTED` indicate if a code object is nested within another code block, `CO_VARARGS` indicates if a code block has variable arguments. These flags affect the behaviour of the evaluation loop during bytecode execution.
7. *co_lnotab*: The contains a string of bytes used to compute the source line numbers that correspond to instruction at a bytecode offset. For example, the `dis` the function makes use of this when calculating line numbers for instructions.
8. *co_varnames*: This is the number of locally defined names in a code block. Contrast this with `co_names`.
9. *co_names*: This a collection of non-local names used within the code object. For example, the snippet in listing 5.4 references a non-local variable, `p`.

Listing 5.4: Illustrating local and non-local names

```
def test_non_local():
    x = p + 1
    return x
```

List 5.5 is the result of introspecting on the code object for the function in Listing 5.4.

Listing 5.5: Illustrating local and non-local names

```
co_argcount = 0
co_cellvars = ()
co_code = b't\x00d\x01\x17\x00}\x00|\x00S\x00'
co_consts = (None, 1)
co_filename = /Users/c4obi/projects/python_source/cpython/fizzbuzz.py
co_firstlineno = 18
co_flags = 67
co_freevars = ()
co_kwonlyargcount = 0
co_lnotab = b'\x00\x01\x08\x01'
co_name = test_non_local
co_names = ('p',)
co_nlocals = 1
co_stacksize = 2
co_varnames = ('x',)
```

From this example, the difference between the `c_names` and `co_varnames` is noticeable. `co_varnames` references the locally defined names while `co_names` references non-locally defined names. Do note that it is only during execution of the program that an error is raised when the name `p` is not found. Listing 5.6 shows the bytecode instructions for the function in Listing 5.4, and it is an easy set to understand.

Listing 5.6: Bytecode instructions for test_non_local function

```
 0 LOAD_GLOBAL              0 (0)
 3 LOAD_CONST               1 (1)
 6 BINARY_POWER
 7 STORE_FAST               0 (0)
10 LOAD_FAST                0 (0)
13 RETURN_VALUE
```

Note how rather than a `LOAD_FAST` as was seen in the previous example, we have `LOAD_GLOBAL` instruction. Later, when we discuss the evaluation loop, we will discuss an optimisation that the evaluation loop carries out that makes the use of the `LOAD_FAST` instruction as the name suggests.

10. *co_nlocals*: This is a numeric value that represents the number of local names used by the code object. In the immediate past example from Listing 5.4, the only local variable used is `x` and thus this value is `1` for the code object of that function.
11. *co_stacksize*: The Python virtual machine is stack-based, i.e. values used in evaluation and results of the evaluation are read from and written to an execution stack. This `co_stacksize` value is the maximum number of items that exist on the evaluation stack at any point during the execution of the code block.
12. *co_freevars*: The *co_freevars* field is a collection of free variables defined within the code block. This field is mostly relevant to nested functions that form closures. Free variables are variables that are used within a block but not defined within that block; this does not apply to global variables. The concept of a free variable is best illustrated with an example, as shown in listing 5.7.

Listing 5.7: A simple nested function

```
def f(*args):
        x=1
        def g():
            n = x
```

The `co_freevars` field is empty for the code object of the `f` function while that of the `g` function contains the `x` value. Free variables are strongly interrelated with *cell variables*.

13. *co_cellvars*: The `co_cellvars` field is a collection of names for that require cell storage objects during the execution of a code object. Take the snippet in Listing 5.7, the `co_cellvars` field of the code object for the function - `f`, contains just the name -`x` while that of the nested function's code object is empty; recall from the discussion on free variables that the `co_freevars` collection of the nested function's code object consists of just this name - `x`. This captures the relationship between cell variables, and free variables - a free variable in a nested scope is a cell variable within the enclosing scope. Special cell objects are required to store the values in this cell variable collection during the execution of the code object. This is so because each value in this field is used by nested code objects whose lifetime may exceed that of the enclosing code object. Hence, such values require storage in locations that do not get deallocated after the execution of the code object.

The bytecode - `co_code` in more detail.

The actual virtual machine instructions for a code object, the bytecode, are contained in the `co_code` field of a code object as previously mentioned. The byte code from the `fizzbuzz` function, for example, is the string of bytes shown in listing 5.7.

Listing 5.7: Bytecode string for fizzbuzz function

```
b'|\x00d\x01\x16\x00d\x02k\x02r\x1e|\x00d\x03\x16\x00d\x02k\x02r\x1ed\x04S\x00n,|\x0\
0d\x01\x16\x00d\x02k\x02r0d\x05S\x00n\x1a|\x00d\x03\x16\x00d\x02k\x02rBd\x06S\x00n\x\
08t\x00|\x00\x83\x01S\x00d\x00S\x00'
```

To get a human-readable version of the byte string, we use the `dis` function from the `dis` module to extract a human-readable printout as shown in listing 5.8.

Listing 5.8: Bytecode instruction disassembly for fizzbuzz function

```
  7           0 LOAD_FAST                0 (n)
              2 LOAD_CONST               1 (3)
              4 BINARY_MODULO
              6 LOAD_CONST               2 (0)
              8 COMPARE_OP               2 (==)
             10 POP_JUMP_IF_FALSE       30
             12 LOAD_FAST                0 (n)
             14 LOAD_CONST               3 (5)
             16 BINARY_MODULO
             18 LOAD_CONST               2 (0)
             20 COMPARE_OP               2 (==)
             22 POP_JUMP_IF_FALSE       30

              ...

 14     >>   66 LOAD_GLOBAL              0 (str)
             68 LOAD_FAST                0 (n)
             70 CALL_FUNCTION            1
             72 RETURN_VALUE
        >>   74 LOAD_CONST               0 (None)
             76 RETURN_VALUE
```

The first column of the output shows the line number for that instruction. Multiple instructions may map to the same line number. This value is calculated using information from the `co_lnotab` field of a code object. The second column is the offset of the given instruction from the start of the bytecode. Assuming the bytecode string is contained in an array, then this value is the index of the instruction into the array. The third column is the actual human-readable instruction opcode; the full range of opcodes are found in the `Include/opcode.h` module. The fourth column is the argument to the instruction.

The first `LOAD_FAST` instruction takes the argument `0`. This value is an index into the `co_varnames` array. The last column is the value of the argument - provided by the `dis` function for ease of use. Some arguments do not take explicit arguments. Notice that the `BINARY_MODULO` and `RETURN_VALUE`

instructions take no explicit argument. Recall that the Python virtual machine is stack-based so these instructions read values from the top of the stack.

Bytecode instructions are two bytes in size - one byte for the opcode and the second byte for the argument to the opcode. In the case where the opcode does not take an argument, then the second argument byte is zeroed out. The Python virtual machine uses a little-endian byte encoding on the machine which I am currently typing out this book thus the 16 bits of code are structured as shown in figure 5.0 with the opcode taking up the higher 8 bits and the argument to the opcode taking up the lower 8 bits.

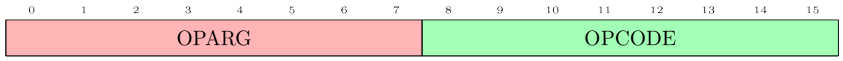


Figure 5.0: Bytecode instruction format showing opcode and oparg

Sometimes, the argument to an opcode may be unable to fit into the default single byte. The Python virtual machine makes use of the `EXTENDED_ARG` opcode for these kinds of arguments. What the Python virtual machine does is to take an argument that is too large to fit into a single byte and split it into two (we assume that it can fit into two bytes here, but this logic is easily extended past two bytes) - the most significant byte is an argument to the `EXTENDED_ARG` opcode while the least significant byte is the argument to its actual opcode. The `EXTENDED_ARG` opcode(s) will come before the actual opcode in the sequence of opcodes, and the argument can then be rebuilt by shifts to the right and **or'ing** with other sections of the argument. For example, if one wanted to pass the value 321 as an argument to the `LOAD_CONST` opcode, this value cannot fit into a single byte, so the `EXTENDED_ARG` opcode is used. The binary representation of this value is 0b101000001, so the actual *do work* opcode (**LOAD_CONST**) takes the first byte (1000001) as argument (65 in decimal) while the `EXTENDED_ARG` opcode takes the next byte (1) as an argument; thus, we have `(144, 1), (100, 65)` as the sequence of instructions that is output.

The documentation for the dis module[3] contains a comprehensive list and explanation of all opcodes currently implemented by the virtual machine.

5.2 Code Objects within other code objects

Another code block code object that is worth looking at is that of a module. Assuming we are compiling a module with the `fizzbuzz` function as content, what would the output, look like? To find out, we use the `compile` function in python to compile a module with the content shown in listing 5.9.

[3] https://docs.python.org/3.6/library/dis.html

Listing 5.9: Nested function to illustrated nested code objects

```
    def f():
        print(c)
        a = 1
        b = 3
        def g():
            print(a+b)
            c=2
            def h():
                print(a+b+c)
```

Listing 5.10 is the result of compiling a module code block.

Listing 5.10: Bytecode instruction disassembly for listing 5.10

```
    0 LOAD_CONST               0 (<code object f at 0x102a028a0, file "fizzbuzz.py",\
 line 1>)
    2 LOAD_CONST               1 ('f')
    4 MAKE_FUNCTION            0
    6 STORE_NAME               0 (f)
    8 LOAD_CONST               2 (None)
   10 RETURN_VALUE
```

The instruction at byte offset 0 loads a code object stored as the name `f` - our function definition using the `MAKE_FUNCTION` Instruction. Listing 5.11 is the content of this code object.

Listing 5.11: Bytecode instruction disassembly for nested function from listing 5.9

```
    co_argcount = 0
    co_cellvars = ()
    co_code = b'd\x00d\x01\x84\x00Z\x00d\x02S\x00'
    co_consts = (<code object f at 0x1022029c0, file "fizzbuzz.py", line 1>, 'f', No\
ne)
    co_filename = fizzbuzz.py
    co_firstlineno = 1
    co_flags = 64
    co_freevars = ()
    co_kwonlyargcount = 0
    co_lnotab = b''
    co_name = <module>
    co_names = ('f',)
    co_nlocals = 0
    co_stacksize = 2
    co_varnames = ()
```

As would be expected in a module, the fields related to code object arguments are all zero - (`co_argcount`, `co_kwonlyargcount`). The `co_code` field contains bytecode instructions, as shown in listing 5.10. The `co_consts` field is an interesting one. The constants in the field are a code object and the names - `f` and `None`. The code object is that of the function, the value 'f' is the name of the function, and None is the return value of the function - recall the python compiler adds a `return None` statement to a code object without one.

Notice that function objects are not created during the module's compilation. What we have are just code objects - it is during the execution of the code objects that the function gets created as seen in Listing 5.10. Inspecting the attributes of the code object will show that it is also composed of other code objects as shown in listing 5.12.

Listing 5.12: Bytecode instruction disassembly for nested function from listing 5.10

```
    co_argcount = 0
    co_cellvars = ('a', 'b')
    co_code = b't\x00t\x01\x83\x01\x01\x00d\x01\x89\x00d\x02\x89\x01\x87\x00\x87\x01\
f\x02d\x03d\x04\x84\x08}\x00d\x00S\x00'
    co_consts = (None, 1, 3, <code object g at 0x101a028a0, file "fizzbuzz.py", line\
 5>, 'f.<locals>.g')
    co_filename = fizzbuzz.py
    co_firstlineno = 1
    co_flags = 3
    co_freevars = ()
    co_kwonlyargcount = 0
    co_lnotab = b'\x00\x01\x08\x01\x04\x01\x04\x01'
    co_name = f
    co_names = ('print', 'c')
    co_nlocals = 1
    co_stacksize = 3
    co_varnames = ('g',)
```

The same logic explained earlier on applies here with the function object created only during the execution of the code object.

5.3 Code Objects in the VM

Like most built-in types, there is the `code` type that defines the code object type and the `PyCodeObject` structure for code objects instances. The code type is similar to other type objects that have been discussed in previous sections, so we do not reproduce it here. Listing 5.13 shows the structures used to represent code objects instances.

Listing 5.13: Code object implementation in C

```
typedef struct {
    PyObject_HEAD
    int co_argcount;         /* #arguments, except *args */
    int co_kwonlyargcount;   /* #keyword only arguments */
    int co_nlocals;          /* #local variables */
    int co_stacksize;        /* #entries needed for evaluation stack */
    int co_flags;            /* CO_..., see below */
    int co_firstlineno;      /* first source line number */
    PyObject *co_code;       /* instruction opcodes */
    PyObject *co_consts;     /* list (constants used) */
    PyObject *co_names;      /* list of strings (names used) */
    PyObject *co_varnames;   /* tuple of strings (local variable names) */
    PyObject *co_freevars;   /* tuple of strings (free variable names) */
    PyObject *co_cellvars;   /* tuple of strings (cell variable names) */

    unsigned char *co_cell2arg; /* Maps cell vars which are arguments. */
    PyObject *co_filename;   /* unicode (where it was loaded from) */
    PyObject *co_name;       /* unicode (name, for reference) */
    PyObject *co_lnotab;     /* string (encoding addr<->lineno mapping) See
                    Objects/lnotab_notes.txt for details. */
    void *co_zombieframe;    /* for optimization only (see frameobject.c) */
    PyObject *co_weakreflist;   /* to support weakrefs to code objects */
    /* Scratch space for extra data relating to the code object.__icc_nan
       Type is a void* to keep the format private in codeobject.c to force
       people to go through the proper APIs. */
    void *co_extra;
} PyCodeObject;
```

The fields are almost all the same as those found in a Python code objects except for the `co_stacksize`, `co_flags`, `co_cell2arg`, `co_zombieframe`, `co_weakreflist` and `co_extra`. `co_weakreflist` and `co_extra` are not really interesting fields at this point. The rest of the fields here pretty much serve the same purpose as those in the code object. The `co_zombieframe` is a field that exists for optimisation purposes. It holds a reference to a frame object that was previously used as a context to execute the code object. This is then used as the execution frame when such code object is being re-executed to prevent the overhead of allocating memory for another frame object.

6. Frame Objects

Frame objects provide the contextual environment for executing bytecode instructions. Take the set of bytecode instructions in listing 6.0 for example, `LOAD_COST` loads values on to a stack, but it has no notion of where or what this stack is. The code object also has no information on the thread or interpreter state that is vital for execution.

Listing 6.0: A set of bytecode instructions

```
    0 LOAD_CONST               0 (<code object f at 0x102a028a0, file "fizzbuzz.py",\
 line 1>)
    2 LOAD_CONST               1 ('f')
    4 MAKE_FUNCTION            0
    6 STORE_NAME               0 (f)
    8 LOAD_CONST               2 (None)
   10 RETURN_VALUE
```

Executing code objects requires another data structure that provides such contextual information, and this is where the frame objects come into play. One can think of the frame object as a container in which the code object is executed - it knows about the code object and has references to data and values required during the execution of some code object. As usual, Python does provide us with some facilities to inspect frame objects using the `sys._getframe()` function, as shown in the Listing 6.1 snippet.

Listing 6.1: Accessing frame objects

```
>>> import sys
>>> f = sys._getframe()
>>> f
<frame object at 0x10073ed48>
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'f_' is not defined
>>> dir(f)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__','__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
'f_back', 'f_builtins', 'f_code', 'f_globals', 'f_lasti', 'f_lineno',
'f_locals', 'f_trace']
```

Before a code object can be executed, a frame object within which the execution of such a code object takes place is created. Such a frame object contains all the namespaces required for the execution of a code object (*local*, *global*, and *builtin*), a reference to the current thread of execution, stacks for evaluating byte code and other housekeeping information that are important for executing byte code. To get a better understanding of the frame object, let us look at the definition of the frame object data structure from the `Include/frame.h` module and reproduced in listing 6.2.

Listing 6.2: Frame object definition in the vm

```
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 (PyDictObject) */
    PyObject *f_globals;        /* global symbol table (PyDictObject) */
    PyObject *f_locals;         /* local symbol table (any mapping) */
    PyObject **f_valuestack;    /* points after the last local */
    PyObject **f_stacktop;
    PyObject *f_trace;          /* Trace function */

    /* fields for handling generators*/
    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback;
    /* Borrowed reference to a generator, or NULL */
    PyObject *f_gen;

    int f_lasti;                /* Last instruction if called */
    int f_lineno;               /* Current line number */
    int f_iblock;               /* index in f_blockstack */
    char f_executing;           /* whether the frame is still executing */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    PyObject *f_localsplus[1];  /* locals+stack, dynamically sized */
} PyFrameObject;
```

The fields coupled with the documentation within the frame are not difficult to understand but we provide a bit more detail about these fields and how they relate to the execution of bytecode.

1. `f_back`: This field is a reference to the frame of the code object that was executing before the current code object. Given a set of frame objects, the `f_back` fields of these frames together form a stack of frames that goes back to the initial frame. This initial frame then has a `NULL` value in this `f_back` field. This implicit stack of frames forms what we refer to as the `call stack`.
2. `f_code`: This field is a reference to a code object. This code object contains the bytecode that is executed within the context of this frame.

3. `f_builtins`: This is a reference to the *builtin* namespace. This namespace contains names such as `print`, `enumerate` etc. and their corresponding values.
4. `f_globals`: This is a reference to the global namespace of a code object.
5. `f_locals`: This is a reference to the local namespace of a code object. As previously mentioned, these names are defined within the scope of a function. When we discuss the `f_localplus` field, we will see an optimization that Python does when working with locally defined names.
6. `f_valuestack`: This is a reference to the evaluation stack for the frame. Recall that the Python virtual machine is a stack-based virtual machine so, during the evaluation of bytecode, values are read from the top of this stack and results of evaluating the byte code are stored on the top of this stack. This field is the stack that is used during code object execution. The `stacksize` of a frame's code object gives the maximum depth to which this data structure can grow.
7. `f_stacktop`: As the name suggests, the field points to the next free slot of the evaluation value stack. When a frame is newly created, this value is set to the value stack - this is the first available space on the stack as there are no items on the stack.
8. `f_trace`: This field references a function that used for tracing the execution of python code.
9. `f_exc_type`, `f_exc_value`, `f_exc_traceback`, `f_gen`: are fields used for bookkeeping to be able to execute generator code cleanly. More on this when we discuss python generators.
10. `f_localplus`: This is a reference to an array that contains enough space for storing *cell* and *local* variables. This field enables the evaluation loop to optimize loading and storing values of names to and from the value stack with the `LOAD_FAST` and `STORE_FAST` instructions. The `LOAD_FAST` and `STORE_FAST` opcodes provide faster name access than their counterpart `LOAD_NAME` and `STORE_NAME` opcodes because they use array indexing for accessing the value of names and this is done in approximately constant time, unlike their counterparts that search a mapping for a given name. When we discuss the evaluation loop, we see how this value is set up during the frame bootstrapping process.
11. `f_blockstack`: This field references a data structure that acts as a stack used to handle loops and exception handling. This is the second stack in addition to the value stack that is of utmost importance to the virtual machine, but this does not receive as much attention as it rightfully should. The relationship between the block stack, exceptions and looping constructs is quite complicated, and we look at that in the coming chapters.

6.1 Allocating Frame Objects

Frame objects are ubiquitous during python code evaluation - every executed code block needs a frame object that provides some context. New frame objects are created by invoking the `PyFrame_New` function in the `Objects/frameobject.c` module. This function is invoked so many times - whenever a code object is executed, that two main optimizations are used to reduce the overhead of invoking this function, and we briefly look at these optimizations.

First, code objects have a field, the `co_zombieframe` which references an *inert* frame object. When a code object is executed, the frame within which it was executed is not immediately deallocated. The frame is rather maintained in the `co_zombieframe` so when next the same code object executed,

time is not spent allocating memory for a new execution frame. The `ob_type`, `ob_size`, `f_code`, `f_valuestack` fields retain their value; `f_locals`, `f_trace`, `f_exc_type`, `f_exc_value`, `f_exc_traceback` are `NULL` and `f_localplus` retains its allocated space but with the local variables nulled out. The remaining fields do not hold a reference to any object. The second optimization that is used by the virtual machine is to maintain a *free list* of pre-allocated frame objects from which frames can be obtained for the execution of code objects.

The source code for frame objects is a gentle read and one can see how the `zombie` frame and *freelist* concepts are implemented by looking at how allocated frames are deallocated after the execution of the enclosed code object. The interesting part of the code for frame deallocation is shown in listing 6.3.

Listing 6.3: Deallocating frame objects

```
        if (co->co_zombieframe == NULL)
            co->co_zombieframe = f;
        else if (numfree < PyFrame_MAXFREELIST) {
            ++numfree;
            f->f_back = free_list;
            free_list = f;
        }
        else
            PyObject_GC_Del(f);
```

Careful observation shows that the *freelist* will only ever grow when a recursive call is made, i.e. a code object tries to execute itself as that is the only time the `zombieframe` field is `NULL`. This small optimization of using the *freelist* helps eliminate to a certain degree, the repeated memory allocations for such recursive calls.

This chapter covers the main points about the frame object without delving into the evaluation loop, which is tightly integrated with the frame objects. A few things left out of this chapter are covered in subsequent chapters. For example,

1. How are values passed on from one frame to the next when code execution hits a `return` statement?
2. What is the thread state, and where does the thread state come from?
3. How are exceptions bubble down the stack of frames when an exception is thrown in the executing frame? Etc.

Most of these question are answered when we look at the interpreter and thread state data structures in the next chapter, and then the evaluation loop in subsequent chapters.

7. Interpreter and Thread States

As mentioned in previous chapters, initializing the interpreter and thread state data structures is one of the steps involved in the bootstrap of the Python interpreter. In this chapter, we provide a detailed look at these data structures.

7.1 The Interpreter state

The `Py_Initialize` function in the `pylifecycle.c` module is one of the bootstrap functions invoked during the initialization of the Python interpreter. This function handles the set-up of the Python runtime as well as the initialization of the interpreter state and thread state data structures among other things.

The interpreter state is a straightforward data structure that captures the global state shared by a set of cooperating threads of execution in a Python process. Listing 7.0 is a cross-section of this data structure's definition.

Listing 7.0: Cross-section of the interpreter state data structure

```
        typedef struct _is {

            struct _is *next;
            struct _ts *tstate_head;

            PyObject *modules;
            PyObject *modules_by_index;
            PyObject *sysdict;
            PyObject *builtins;
            PyObject *importlib;

            PyObject *codec_search_path;
            PyObject *codec_search_cache;
            PyObject *codec_error_registry;
            int codecs_initialized;
            int fscodec_initialized;

            ...

            PyObject *builtins_copy;
```

```
        PyObject *import_func;
    } PyInterpreterState
```

The fields in listing 7.0 should be familiar if one covered the prior materials in this book, and has used Python for a considerable amount of time. We discuss some of the fields of the interpreter state data structure once again.

- `*next`: There can be multiple interpreter states within a single OS process that is running a python executable. This `*next` field references another interpreter state data structure within the python process if such exist, and these form a linked list of interpreter states, as shown in figure 7.0. Each interpreter state has its own set of variables that will be used by a thread of execution that references that interpreter state. However, all interpreter threads in the process share the same memory space and *Global Interpreter Lock*.

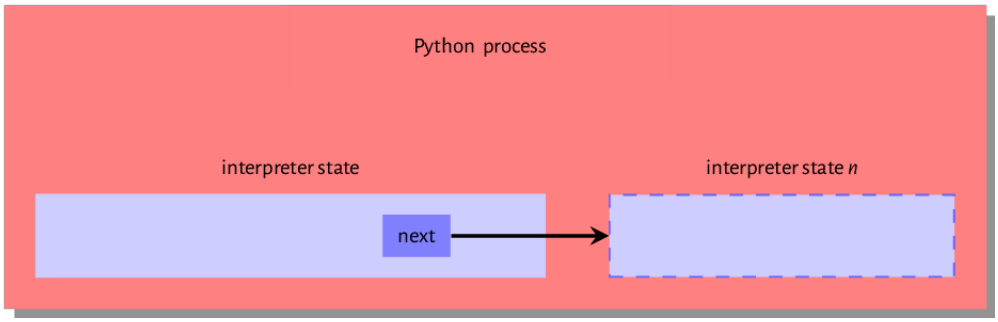


Figure 7.0: Interpreter state within the executing python process

- `*tstate_head`: This field references the thread state of the currently executing thread or in the case of a multithreaded program, the thread that currently holds the Global Interpreter Lock (GIL). This is a data structure that maps to an executing operating system thread.

The remaining fields are variables that are shared by all cooperating threads of the interpreter state. The `modules` field is a table of installed Python modules - we see how the interpreter finds these modules later on when we discuss the import system, the `builtins` field is a reference to the built-in `sys` module. The content of this module is the set of built-in functions such as `len`, `enumerate` etc. and the `Python/bltinmodule.c` module contains implementations for most of the contents of the module. The `importlib` is a field that references the implementation of the import mechanism - we speak a bit more about this when we discuss the import system in detail. The `*codec_search_path`, `*codec_search_cache`, `*codec_error_registry`, `*codecs_initialized` and `*fscodec_initialized` are fields that all relate to codecs that Python uses to encode and decode bytes and text. The interpreter uses values in these fields to locate such codecs as well as handle errors related to using such codecs. An executing Python program is composed of one or more threads of execution. The interpreter has to maintain some state for each thread of execution, and this works by maintaining a thread state data structure for each thread of execution. We look at this data structure next.

7.2 The Thread state

Reviewing the *Thread state* data structure in listing 7.1, one can see that the thread state data structure is a more involved data structure than the interpreter state data structure.

Listing 7.2: Cross-section of the thread state data structure

```
    typedef struct _ts {
        struct _ts *prev;
        struct _ts *next;
        PyInterpreterState *interp;

        struct _frame *frame;
        int recursion_depth;
        char overflowed;
        char recursion_critical;
        int tracing;
        int use_tracing;

        Py_tracefunc c_profilefunc;
        Py_tracefunc c_tracefunc;
        PyObject *c_profileobj;
        PyObject *c_traceobj;

        PyObject *curexc_type;
        PyObject *curexc_value;
        PyObject *curexc_traceback;

        PyObject *exc_type;
        PyObject *exc_value;
        PyObject *exc_traceback;

        ...

    } PyThreadState;
```

A thread state data structure's `previous` and `next` fields reference thread states created before and just after the given thread state. These fields form a doubly-linked list of thread states that share a single interpreter state. The `interp` field references the thread state's interpreter state. The `frame` references the current frame of execution; the value referenced by this field changes when the code object that is executing changes.

The `recursion_depth`, as the name suggests, specifies how deep the stack frame should get during a recursive call. The `overflowed` flag is set when the stack overflows. After a stack overflow, the

thread allows 50 more calls for clean-up operations. The `recursion_critical` flag signals to the thread that the code being executed should not overflow. The `tracing` and `use_tracing` flag are related to functionality for tracing the execution of the thread. The `*curexc_type`, `*currexc_value`, `*curexc_traceback`, `*exc_type`, `*exc_value` and `*curexc_traceback` are fields that are all used in the exception handling process as will be seen in subsequent chapters.

It is essential to understand the difference between the thread state and an actual thread. The thread state is just a data structure that encapsulates some state for an executing thread. Each thread state is associated with a native OS thread within the Python process. Figure 7.1 is an excellent visual illustration of this relationship. We can see that a single python process is home to at least one interpreter state and each interpreter state is home to one or more thread states, and each of these thread states maps to an operating system thread of execution.

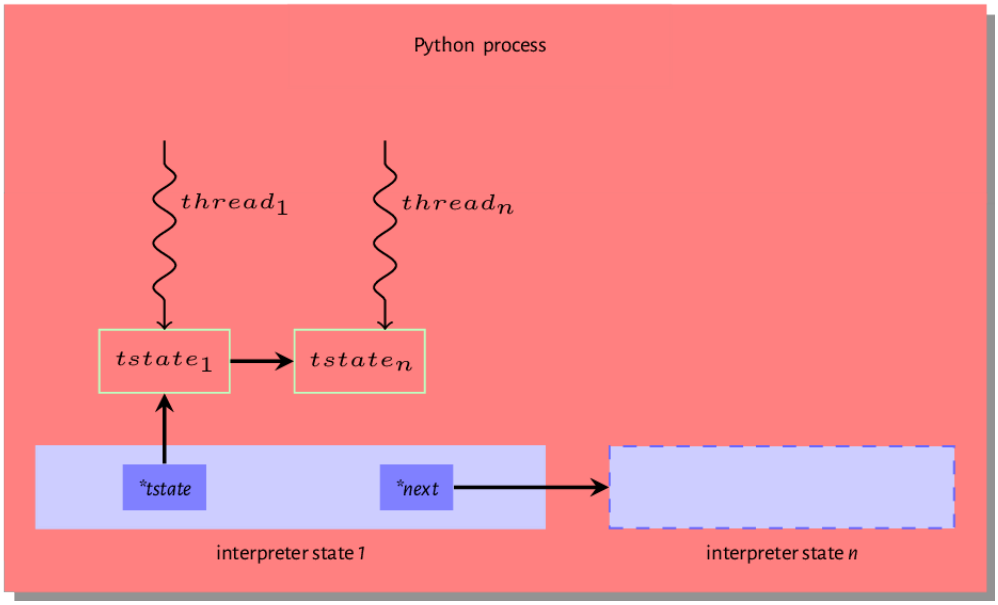


Figure 7.1: Relationship between interpreter state and thread states

Operating System threads and associated Python thread states are created either during the initialization of the interpreters or when invoked by the threading module. Even with multiple threads alive within a Python process, only one thread can actively carry out CPU bound tasks at any given time. This is because an executing thread must hold the GIL to execute byte code within the python virtual machine. This chapter will not be complete without a look at the famous or *infamous* GIL concept so we take this on in the next section

Global Interpreter Lock - `GIL`

Although python threads are operating system threads, a thread cannot execute python bytecode unless such thread holds the `GIL`. The operating system may schedule a thread that does not hold the `GIL` to run but as we will see, all such a thread can do is wait to get the `GIL` and only when it holds the `GIL` is it able to execute bytecode. We take a look at this whole process.

The Need for a GIL

Before we begin any discussion on the `GIL`, it is worthwhile to ask why we need a global lock that will probably adversely affects our threads? The `GIL` is relevant for a myriad of reasons. First of all, however, it is important to understand that the `GIL` is an implementation detail of CPython and not an actual language detail - Jython which is Python implemented on the Java virtual machine has no notion of a `GIL`. The primary reason the `GIL` exist is for ease of implementation of the CPython virtual machine. It is way easier to implement a single global lock than to implement fine-grained locks and the core developers have opted for this. There have however been projects to implement fine-grained locks within the Python virtual machine but these have sometimes slowed down single-threaded programs. A global lock also provides much-needed synchronization when performing specific tasks. Take the reference counting mechanism that is used by CPython for memory management, without the concept of a `GIL`, you may have two thread interleave their increment and decrement of reference count, leading to severe issues with memory handling. Another reason for this lock is that some `C` libraries that CPython calls into are inherently not thread-safe, so some kind of synchronization is required when using them.

When the interpreter startups, a single main thread of execution is created, and there is no contention for the `GIL` as there is no other thread around, so the main thread does not bother to acquire the lock. The `GIL` comes into play after other threads are spawned. The snippet in listing 7.3 is from the `Modules/_threadmodule.c` and provides insight into this process during the creation of a new thread.

Listing 7.3: Cross-section of code for creating new thread

```
    boot->interp = PyThreadState_GET()->interp;
    boot->func = func;
    boot->args = args;
    boot->keyw = keyw;
    boot->tstate = _PyThreadState_Prealloc(boot->interp);
    if (boot->tstate == NULL) {
        PyMem_DEL(boot);
        return PyErr_NoMemory();
    }
    Py_INCREF(func);
    Py_INCREF(args);
    Py_XINCREF(keyw);
    PyEval_InitThreads(); /* Start the interpreter's thread-awareness */
    ident = PyThread_start_new_thread(t_bootstrap, (void*) boot);
```

The snippet in listing 7.3 is from the `thread_PyThread_start_new_thread` function that is invoked to create a new thread. `boot` is a data structure the contains all the information that a new thread needs

to execute. The `tstate` field references the thread state for the new thread, and the `_PyThreadState_Prealloc` function call creates this thread state. The main thread of execution must acquire the `GIL` before creating the new thread; a call to `PyEval_InitThreads` handles this. With the interpreter now thread-aware and the main thread holding the `GIL`, the `PyThread_start_new_thread` is invoked to create the new operating system thread. The `_tbootstrap` function in the `Modules/_threadmodule.c` module is a callback function invoked by new threads when they come alive. A snapshot of this bootstrap function is in listing 7.4.

Listing 7.4: Cross-section of thread bootstrapping function

```
    static void t_bootstrap(void *boot_raw){
        struct bootstate *boot = (struct bootstate *) boot_raw;
        PyThreadState *tstate;
        PyObject *res;

        tstate = boot->tstate;
        tstate->thread_id = PyThread_get_thread_ident();
        _PyThreadState_Init(tstate);
        PyEval_AcquireThread(tstate);
        nb_threads++;
        res = PyEval_CallObjectWithKeywords(
            boot->func, boot->args, boot->keyw);

        ...
```

Notice the call to `PyEval_AcquireThread` function in listing 7.4. The `PyEval_AcquireThread` function is defined in the `Python/ceval.c` module and it invokes the `take_gil` function, which is the function that attempts to get a hold of the `GIL`. A description of this process, as provided in the source file, is quoted in the following text.

> The GIL is just a boolean variable (gil_locked) whose access is protected by a mutex (gil_mutex), and whose changes are signalled by a condition variable (gil_cond). gil_mutex is taken for short periods of time, and therefore mostly uncontended. In the GIL-holding thread, the main loop (PyEval_EvalFrameEx) must be able to release the GIL on demand by another thread. A volatile boolean variable (gil_drop_request) is used for that purpose, which is checked at every turn of the eval loop. That variable is set after a wait of `interval` microseconds on `gil_cond` has timed out. [Actually, another volatile boolean variable (eval_breaker) is used which ORs several conditions into one. Volatile booleans are sufficient as inter-thread signalling means since Python is run on cache-coherent architectures only.] A thread wanting to take the GIL will first let pass a given amount of time (`interval` microseconds) before setting gil_drop_request. This encourages a defined switching period, but does not enforce it since opcodes can take an arbitrary time to execute. The `interval` value is available for the user to read and modify using the

> Python API `sys.{get,set}switchinterval()`. When a thread releases the GIL and gil_drop_request is set, that thread ensures that another GIL-awaiting thread gets scheduled. It does so by waiting on a condition variable (switch_cond) until the value of gil_last_holder is changed to something else than its own thread state pointer, indicating that another thread has taken GIL. This prohibits the latency-adverse behaviour on multi-core machines where one thread would speculatively release the GIL, but still, run and end up being the first to re-acquire it, making the "timeslices" much longer than expected.

What does the above mean for a newly spawned thread? The `t_bootstrap` function in listing 7.4 invokes the `PyEval_AcquireThread` function that handles *requesting* for the `GIL`. A lay explanation for what happens during this request is thus - assume `A` is the main thread of execution holding the `GIL` while `B` is the new thread that has just been spawned.

1. When `B` is spawned, `take_gil` is invoked. This checks if the conditional `gil_cond` variable is set. If it is not set then the thread starts a wait.
2. After wait time elapses, the `gil_drop_request` is set.
3. Thread A, after each trip through the execution loop, checks if any other thread has set the`gil_drop_request` variable.
4. Thread A drops the `GIL` when it detects that the `gil_drop_request` variable is set and also sets the `gil_cond` variable.
5. Thread A also waits on another variable - `switch_cond`, until the value of the `gil_last_holder` is set to a value other than thread A's thread state pointer indicating that another thread has taken the `GIL`.
6. Thread B now has the `GIL` and can go ahead to execute bytecode.
7. Thread A waits a given time, sets the `gil_drop_request` and the cycle continues.

GIL and Performance

The `GIL` is the primary reason why increasing the number of threads working on a CPU bound program in a single processor core setting in in Python most times does not speed up such a program. In fact, at times, adding a thread will adversely affect the performance of a program when compared to that of a single-threaded program; this is because there is a cost associated with all the switches and waits.

To conclude this chapter, we recap the model of the Python virtual machine we have created so far that captures when we run the Python interpreter with a source file. First, the interpreter initializes interpreter and thread states, the source in the file is compiled into a code object. The code object is then passed to the interpreter loop where a frame object is created and attached to the main thread of execution, so the execution of the code object can happen. So we have a Python process that may contain one or more interpreter states, and each interpreter state may have one or more thread

states, and each thread states references a frame that may reference another frame etc., forming a stack of frames. Figure 7.2 provides a visual representation of this order.

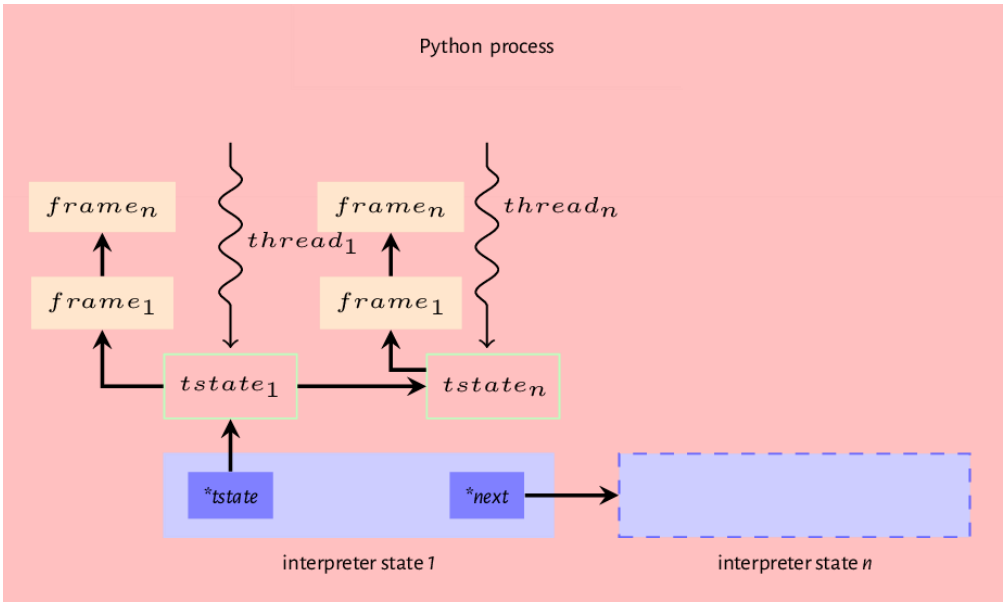


Figure 7.2: Interpreter state, thread state and frame relationship

In the next chapter, we show how all the parts that we have described enable the execution of a Python code object.

8. Intermezzo: The `abstract.c` Module

We have thus far mentioned severally that the Python virtual machine generically treat values for evaluation as `PyObjects`. This begets the obvious question - *How are operations safely carried out on such generic objects ?*. For example, when evaluating the bytecode instruction `BINARY_ADD`, two `PyObject` values are popped from the evaluation stack and used as arguments to an add operation, but how does the virtual machine know if the add operation makes sense for both values?

To understand how a lot of the operations on `PyObjects` work, we only have to look at the `Objects/Abstract.c` module. This module defines several functions that operate on objects that implement a given object protocol. This means that for example, if one were adding two objects, then the add function in this module would expect that both objects implement the `__add__` method of the `tp_numbers` slots. The best way to explain this is to illustrate with an example.

Consider when the `BINARY_ADD` opcode adds two numbers, the function that does the addition is the `PyNumber_Add` function of the `Objects/Abstract.c` module. Listing 8.1 is the definition of the `PyNumber_Add` function.

Listing 8.1: Generic add function from abstract.c module

```
1    PyObject * PyNumber_Add(PyObject *v, PyObject *w){
2        PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
3        if (result == Py_NotImplemented) {
4            PySequenceMethods *m = v->ob_type->tp_as_sequence;
5            Py_DECREF(result);
6            if (m && m->sq_concat) {
7                return (*m->sq_concat)(v, w);
8            }
9            result = binop_type_error(v, w, "+");
10       }
11       return result;
12   }
```

Our interest at this point is in line 2 of the `PyNumber_Add` function defined in listing 8.1 - the call to the `binary_op1` function. The `binary_op1` function is another generic function that takes among its parameters, two numbers or subclass of numbers and applies a binary function to these; the `NB_SLOT` macro returns the offset of a given method into the `PyNumberMethods` structure; recall that this structure is a collection of methods that work on numbers. The definition of this generic `binary_op1` function is in listing 8.2, and an in-depth explanation of this function immediately follows.

Listing 8.2: The generic binary_op1 function

```
1   static PyObject * binary_op1(PyObject *v, PyObject *w, const int op_slot){
2       PyObject *x;
3       binaryfunc slotv = NULL;
4       binaryfunc slotw = NULL;
5
6       if (v->ob_type->tp_as_number != NULL)
7           slotv = NB_BINOP(v->ob_type->tp_as_number, op_slot);
8       if (w->ob_type != v->ob_type &&
9           w->ob_type->tp_as_number != NULL) {
10          slotw = NB_BINOP(w->ob_type->tp_as_number, op_slot);
11          if (slotw == slotv)
12              slotw = NULL;
13      }
14      if (slotv) {
15          if (slotw && PyType_IsSubtype(w->ob_type, v->ob_type)) {
16              x = slotw(v, w);
17              if (x != Py_NotImplemented)
18                  return x;
19              Py_DECREF(x); /* can't do it */
20              slotw = NULL;
21          }
22          x = slotv(v, w);
23          if (x != Py_NotImplemented)
24              return x;
25          Py_DECREF(x); /* can't do it */
26      }
27      if (slotw) {
28          x = slotw(v, w);
29          if (x != Py_NotImplemented)
30              return x;
31          Py_DECREF(x); /* can't do it */
32      }
33      Py_RETURN_NOTIMPLEMENTED;
34  }
```

1. The function takes three values, two `PyObject *` - `v` and `w` and an integer value, operation slot, which is the offset of that operation into the `PyNumberMethods` structure.
2. Lines 3 and 4 define two values `slotv` and `slotw`, structures that represent a binary function as their types suggest.

3. From line 3 to line 13, we attempt to dereference the function given by op_slot argument for both v and w. On line 8, there is an equality check of both values' types, and if they are of the same type, there is no need to dereference the second value's function in the op_slot. If both values are not of the same type, but the functions dereferenced from both are equal, then the slotw value is nulled out.
4. With the binary functions dereferenced, if slotv is not NULL then on line 15 we check that slotw is not NULL and the type of w is a subtype of the type of v. If that is true, slotw's function is applied to both v and w. This happens because if you pause to think about it for a second, the method further down the inheritance tree is what we want to use not one further up. If w is not a subtype, then slotv is applied to both values at line 22.
5. Getting to line 27 means that the slotv function is NULL so we apply whatever slotw references to both v and w so long as it is not NULL.
6. In the case where both slotv and slotw both do not contain a function, then a Py_-NotImplemented is returned. Py_RETURN_NOTIMPLEMENTED is just a macro that increments the reference count of the Py_NotImplemented value before returning it.

The idea captured by the explanation given above is a blueprint for how the interpreter performs operations on values. We have simplified things a bit here by ignoring that opcodes that can be overloaded. For example, the + symbol maps to the BINARY_ADD opcode and applies to strings, numbers and some sequences, but we have only looked at it in the context of numbers and subclasses of numbers. The BINARY_ADD implementation shown in Listing 8.3 can handle the other cases by looking at the type of values it is operating on and calling the corresponding functions. First, if both values are Unicode characters, the interpreter calls the function for concatenating Unicode characters. Otherwise, thePyNumber_Add function is invoked. This function's implementation shows how it checks for numeric and then sequence types applying corresponding addition functions to the different types.

Listing 8.3: ceval implementation of binary add

```
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    if (PyUnicode_CheckExact(left) &&
        PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to left */
    }
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
```

Ignore lines 1 and 2 as we discuss them when we talk about the interpreter loop. What we see from the rest of the snippet, is that when we encounter the BINARY_ADD, the first port of call is a check that

both values are strings to apply string concatenation to the values. The `PyNumber_Add` function from `Objects/Abstract.c` is then applied to both values if they are not strings. Although the code seems a bit messy with the string check done in `Python/ceval.c` and the number and sequence checks done in `Objects/Abstract.c`, it is pretty clear what is happening when we have an overloaded opcode.

This explanation provided above is the way the interpreter handles most opcode operations - check the types of the values then dereference the method as required and apply to the argument values.

9. The evaluation loop, `ceval.c`

We have finally arrived at the gut of the virtual machine where the virtual machine iterates over a code object's bytecode instructions and executes such instructions. The essence of this is a `for` loop that iterates over opcodes, switching on each opcode type to run the desired code. The `Python/ceval.c` module, about 5411 lines long, implements most of the functionality required - at the heart of this function is the `PyEval_EvalFrameEx` function, an approximately 3000 line long function that contains the actual evaluation loop. It is this `PyEval_EvalFrameEx` function that is the main thrust of our focus in the chapter.

The `Python/ceval.c` module provides platform-specific optimizations such as *threaded gotos* as well as Python virtual machine optimizations such as opcode prediction. In this write-up, we are more concerned with the virtual machine processes and optimizations, so we conveniently disregard any platform-specific optimizations or process introduced here so long as it does not take away from our explanation of the evaluation loop. We go into more detail than usual here to provide a solid explanation for how the heart of the virtual machine is structured and works. It is important to note that the opcodes and their implementations are constantly in flux so this description here may be inaccurate at a later time.

Before any bytecode execution happens, several housekeeping operations such as creating and initializing frames, setting up variables and initializing the virtual machine variables such as instruction pointers are carried out. We look at some of these operations next.

9.1 Putting names in place

As mentioned above, the heart of the virtual machine is the `PyEval_EvalFrameEx` function that executes the Python bytecode but before this happens, a lot of setups - error checking, frame creation and initialization etc. - need to take place to prepare the evaluation context. This is where the `_PyEval_EvalCodeWithName` function also within the `Python/ceval.c` the module comes in. For illustration purposes, we assume that we are working with a module that has the content shown in listing 9.0.

Listing 9.0: Content of a simple module

```
    def test(arg, defarg="test", *args,  defkwd=2, **kwd):
        local_arg = 2
        print(arg)
        print(defarg)
        print(args)
        print(defkwd)
        print(kwd)

    test()
```

Recall that code blocks have code objects; these code blocks could either be functions, modules etc. so for a module with the above content, we can safely assume that we are dealing with two code objects - one for the module and one for the function `test` defined within the module.

After the generation of the code object for the module in listing 9.0, the generated code object is executed via a chain of function calls from the `Python/pythonrun.c` module - `run_mod -> PyEval_EvalCode->PyEval_EvalCodeEx->_PyEval_EvalCodeWithName->PyEval_EvalFrameEx`. At this moment, our interest lies with the `_PyEval_EvalCodeWithName` function with its signature shown in listing 9.1. It handles the required name setup before bytecode evaluation in `PyEval_EvalFrameEx`. However, by looking at the function signature for the `_PyEval_EvalCodeWithName` as shown in listing 9.1, one is probably left asking how this is related to executing a module object rather than an actual function.

Listing 9.1: _PyEval_EvalCodeWithName function signature

```
    static PyObject * _PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyO\
bject *locals,
           PyObject **args, int argcount, PyObject **kws, int kwcount,
           PyObject **defs, int defcount, PyObject *kwdefs, PyObject *closure,
           PyObject *name, PyObject *qualname)
```

To wrap one's head around this, one must think more generally in terms of code blocks and code objects, not functions or modules. Code blocks can have any or none of those arguments specified in the `_PyEval_EvalCodeWithName` function signature - a function just happens to be a more specific type of code block which has most if not all those values supplied. This means that the case of executing `_PyEval_EvalCodeWithName` for a module code object is not very interesting as most of those arguments are without value. The interesting instance occurs when a Python function call is made via the `CALL_FUNCTION` opcode. This results in a call to the `fast_function` function also in the `Python/ceval.c` module. This function extracts function arguments from the function object before delegating to the `_PyEval_EvalCodeWithName` function to carry out all the sanity checks that are needed - this is not the full story, but we will look at the `CALL_FUNCTION` opcode in more detail in a later section of this chapter.

The _PyEval_EvalCodeWithName is quite a big function, so we do not include it here, but most of the setup process that it goes through is pretty straightforward. For example, recall we mentioned that the fastlocals field of a frame object provides some optimization for the local namespace and that non-positional function arguments are known fully only at runtime. This means that we cannot populate this fastlocals data structure without careful error checking. It is during this setup by the _PyEval_EvalCodeWithName function that the array referenced by the fastlocals field of a frame is populated with the full range of local values. The steps involved in the setup process that the _PyEval_EvalCodeWithName goes through when called involves the steps shown in listing 9.1.

Listing 9.2: _PyEval_EvalCodeWithName setup steps

```
1. Initialize a frame object that provides context for the code object execution.
3. Add the keyword *dict* to the frame fast locals.
4. Add positional arguments to `fastlocals`.
5. Add the variable sequence of non-positional, non-keyword arguments to the
`fastlocals` array (`*args` from our example module). These values are held together\
 in a tuple data structure.
6. Check that any keyword argument supplied to a code block is expected and has not \
been provided twice.
7. Check for missing positional arguments and throw an error if any are found.
8. Add the default arguments to the `fastlocals` array (`defarg` in our example modu\
le).
9. Add keyword defaults to `fastlocals` (`defkwd` in our example module).
10. Initialize storage for cell variables and copy free variables array into the
 frame.
11. Do some generator related housekeeping - we look at this in more detail when
we discuss generators.
```

9.2 The parts of the machine

With all the names in place, PyEval_EvalFrameEx is invoked with a frame object as one of its arguments. A cursory look at this function shows that the function is composed of quite a few C macros and variables. The *macros* are an integral part of the execution loop - they provide a means to abstract away repetitive code without incurring the cost of a function call and as such we describe a few of them. In this section, we assume that the virtual machine is not running with C optimizations such as computed gotos enabled so we conveniently ignore macros related to such optimizations.

We begin with a description of some of the variables that are crucial to the execution of the evaluation loop.

1. **stack_pointer: refers to the next free slot in the value stack of the execution frame.

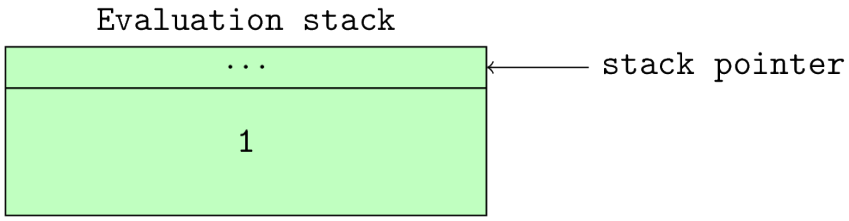


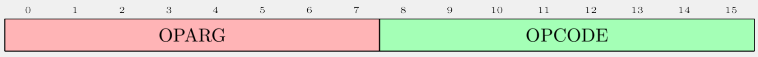
Figure 9.0: Stack pointer after a single value is pushed onto the stack

1. `*next_instr`: refers to the next instruction to be executed by the evaluation loop. One can think of this as the *program counter* for the virtual machine. Python 3.6 changes the type of this value to an `unsigned short` which is 2 bytes in size to handle the new bytecode instruction size.
2. `opcode`: refers to the currently executing python opcode or the opcode that is about to be executed.
3. `oparg`: refers to the argument of the presently executing opcode or opcode that is about to be executed if it takes an argument.
4. `why`: The evaluation loop is an infinite loop implemented by the infinite `for` loop - `for(;;)` so the loop needs a mechanism to break out of the loop and specify why the break occurred. This value refers to the reason for an exit from the evaluation loop. For example if the code block exited the loop due to a return statement, then this value will contain a `WHY_RETURN` status.
5. `fastlocals`: refers to an array of locally defined names.
6. `freevars`: refers to a list of names that are used within a code block but not defined in that code block.
7. `retval`: refers to the return value from executing the code block.
8. `co`: References the code object that contains the bytecode that will be executed by the evaluation loop.
9. `names`: This references the names of all values in the code block of the executing frame.
10. `consts`: This references the constants used by the code objects.

Bytecode instruction

We have discussed the format of bytecode instructions in the chapter on code objects, but it is very relevant to our discussion here so we repeat our description of the bytecode instruction format here.

Assuming we are working with python 3.6 bytecodes, all bytecodes are 16 bit long. The Python VM uses a little-endian byte encoding on the machine which I am currently typing out this book thus the 16 bits of code are structured as shown in the following image with the opcode taking up 1 byte and the argument to the opcode taking up the second byte.



Bytecode instruction format showing opcode and oparg

> Extracting the opcodes and arguments involves some bit manipulation as we will see in the following sections. It is important to note that since the opcode is now two bytes and not one, then the manipulation of instruction pointers subscribes to pointer manipulation[a].
>
> [a] https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/pointer.html

The following macros play a vital role in the evaluation loop.

1. `TARGET(op)`: expands to the `case op` statement. This matches the current opcode with the block of code that implements the opcode.
2. `DISPATCH`: expands to `continue`. This together with the next macro - `FAST_DISPATCH`, handle the flow of control of the evaluation loop after an opcode is executed.
3. `FAST_DISPATCH`: expands to a jump to the `fast_next_opcode` label within the evaluation `for` loop.

With the introduction of the standard 2 bytes opcode in Python 3.6, the following set of macros are used to handle code access.

1. `INSTR_OFFSET()`: This macro provides the byte offset of the current instruction into the array of instructions. This expands to `(2*(int)(next_instr - first_instr))`.
2. `NEXTOPARG()`: This updates the `opcode` and `oparg` variable to the value of the opcode and argument of the next bytecode instruction to be executed. This macro expands to the following snippet.

Listing 9.3: Expansion of the `NEXTOPARG` macro

```
    do { \
        unsigned short word = *next_instr; \
        opcode = OPCODE(word); \
        oparg = OPARG(word); \
        next_instr++; \
    } while (0)
```

The `OPCODE` and `OPARG` macros handle the bit manipulation for extracting opcode and arguments. Figure 9.0 shows the structure of a bytecode instruction with the argument to the opcode taking lower eight bits and the opcode itself taking the upper eight bits hence `OPCODE` expands to `((word) & 255)` thus extracting the most significant byte from the bytecode instruction while `OPARG` which expands to `((word) >> 8)` extracts the least significant byte.

1. `JUMPTO(x)`: This macro expands to `(next_instr = first_instr + (x)/2)` and performs an absolute jump to a particular offset in the bytecode stream.

2. JUMPBY(x): This macro expands to (next_instr += (x)/2) and performs a relative jump from the current instruction offset to another point in the bytecode instruction stream.
3. PREDICT(op): This opcode together with the PREDICTED(op) opcode implement the Python evaluation loop opcode prediction. This opcode expands to the following snippet.

Listing 9.4: Expansion of the PREDICT(op) macro

```
do{ \
    unsigned short word = *next_instr; \
    opcode = OPCODE(word); \
    if (opcode == op){ \
        oparg = OPARG(word); \
        next_instr++; \
        goto PRED_##op; \
    } \
} while(0)
```

4. PREDICTED(op): This macro expands to PRED_##op:.

The last two macros defined above handle opcode prediction. When the evaluation loop encounters a PREDICT(op) macro; the interpreter assumes that the next instruction to be executed is op. The macros check that this is indeed valid and if valid fetches the actual opcode and argument then jumps to the label PRED_##op where the ## is a placeholder for the actual opcode. For example, if we had encountered a prediction such as PREDICT(LOAD_CONST) then the goto statement argument would be PRED_LOAD_CONSTop if that prediction was valid. An inspection of the source code for the PyEval_EvalFrameEx function finds the PREDICTED(LOAD_CONST) label that expands to PRED_LOAD_CONSTop so on a successful prediction of this instruction; there is a jump to this label otherwise normal execution continues. This prediction saves the cost involved with the extra traversal of the switch statement that would otherwise happen with normal code execution.

The next set of macros that we are interested in are the stack manipulation macros that handle placing and fetching of values from the value stack of a frame object. These macros are pretty similar, and the following snippet shows a few examples.

1. STACK_LEVEL(): This returns the number of items on the stack. The macro expands to ((int)(stack_pointer - f->f_valuestack)).
2. TOP(): The returns the last item on the stack. This expands to (stack_pointer[-1]).
3. SECOND(): This returns the penultimate item on the stack. This expands to (stack_pointer[-2]).
4. BASIC_PUSH(v): This places the item, v, on the stack. It expands to (*stack_pointer++ = (v)). A current alias for this macro is the PUSH(v).
5. BASIC_POP(): This removes and returns an item from the stack. This expands to (*--stack_pointer). A current alias for this is the POP() macro.

The last set of macros of concern to us are those that handle local variable manipulation. These macros, GETLOCAL and SETLOCAL are used to get and set values in the fastlocals array.

1. GETLOCAL(i): This expands to (fastlocals[i]). This handles the fetching of locally defined names from the local array.
2. SETLOCAL(i, value): This expands to the snippet in listing 9.5. This macro sets the ith element of the local array to the supplied value.

Listing 9.5: Expansion of the SETLOCAL(i, value) macro

```
        do { PyObject *tmp = GETLOCAL(i); \
             GETLOCAL(i) = value; \
             Py_XDECREF(tmp);
           } while (0)
```

The UNWIND_BLOCK and UNWIND_EXCEPT_HANDLER are related to exception handling, and we look at them in subsequent sections.

9.3 The Evaluation loop

We have finally come to the heart of the virtual machine - the loop where the opcodes are evaluated. The implementation is pretty anti-climatic as there is nothing special here, just a *never-ending* for loop and a massive switch statement that matches on opcodes. To get a concrete understanding of this statement, we look at the execution of the simple hello world function in listing 9.6.

Listing 9.6: Simple hello world python function

```
def hello_world():
    print("hello world")
```

Listing 9.7 shows the disassembly of the function from Listing 9.6, and we illustrate the evaluation of this set of instructions.

Listing 9.7: Disassembly of function in listing 9.6

```
        LOAD_GLOBAL              0 (0)
        LOAD_CONST               1 (1)
        CALL_FUNCTION            1 (1 positional, 0 keyword pair)
        POP_TOP
        LOAD_CONST               0 (0)
        RETURN_VALUE
```

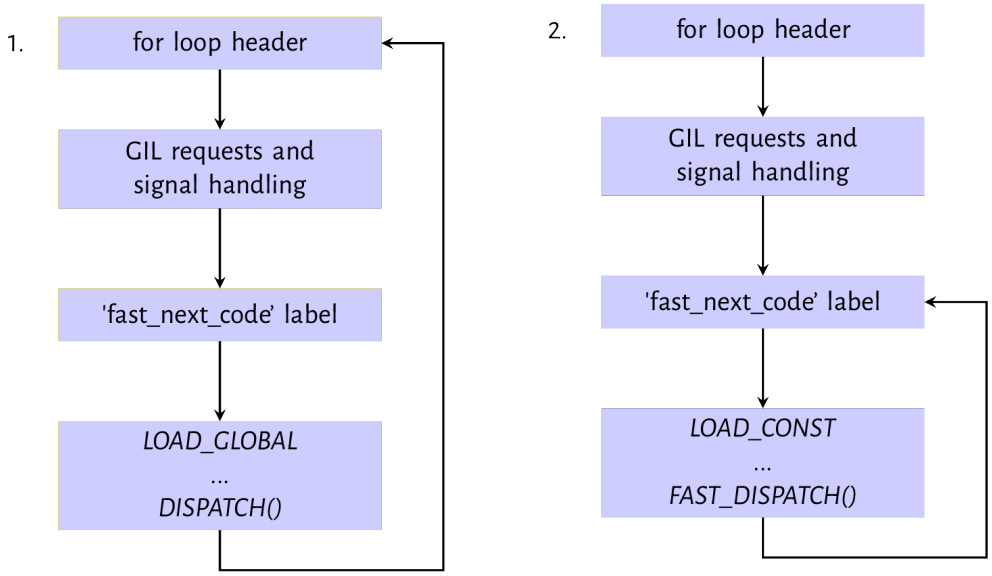


Figure 9.1: Evaluation path for `LOAD_GLOBAL` and `LOAD_CONST` instructions

Figure 9.1 shows the evaluation path for the `LOAD_GLOBAL` and `LOAD_CONST` instructions. The second and third blocks in both images of figure 9.2 represent the housekeeping tasks performed on every iteration of the evaluation loop. The`GIL` and signal handling checks were discussed in the the previous chapter on interpreter and thread states - it is during these checks that a thread executing may give up control of the `GIL` for another thread to execute. The `fast_next_opcode` is a code label just after the `GIL` and signal handling code that exists to serve as a jump destination when the loop wishes to skip the previous checks as we will see when we look at the `LOAD_CONST` instruction.

The first instruction - `LOAD_GLOBAL` is evaluated by the *LOAD_GLOBAL* case statement of the `switch` statement. The implementation of this opcode like other opcodes is a series of C statements and function calls that are surprisingly involved, as shown in listing 9.8. The implementation of the opcode loads the value identified by the given name from the global or builtin namespace onto the evaluation stack. The `oparg` is the index into the tuple which contains all names used within the code block - `co_names`.

Listing 9.8: LOAD_GLOBAL implementation

```
    PyObject *name = GETITEM(names, oparg);
    PyObject *v;
    if (PyDict_CheckExact(f->f_globals)
        && PyDict_CheckExact(f->f_builtins)){
        v = _PyDict_LoadGlobal((PyDictObject *)f->f_globals,
                               (PyDictObject *)f->f_builtins,
                               name);
        if (v == NULL) {
            if (!_PyErr_OCCURRED()) {
                /* _PyDict_LoadGlobal() returns NULL without raising
                 * an exception if the key doesn't exist */
```

```
                format_exc_check_arg(PyExc_NameError,
                                     NAME_ERROR_MSG, name);
            }
            goto error;
        }
        Py_INCREF(v);
    }
    else {
        /* Slow-path if globals or builtins is not a dict */
        /* namespace 1: globals */
        v = PyObject_GetItem(f->f_globals, name);
        if (v == NULL) {
            if (!PyErr_ExceptionMatches(PyExc_KeyError))
                goto error;
            PyErr_Clear();

            /* namespace 2: builtins */
            v = PyObject_GetItem(f->f_builtins, name);
            if (v == NULL) {
                if (PyErr_ExceptionMatches(PyExc_KeyError))
                    format_exc_check_arg(
                                PyExc_NameError,
                                NAME_ERROR_MSG, name);
                goto error;
            }
        }
    }
```

The look-up algorithm for the `LOAD_GLOBAL` opcode first attempts to load the name from the `f_globals` and `f_builtins` fields if they are *dict* objects otherwise it attempts to fetch the value associated with the name from the `f_globals` or `f_builtins` object with the assumption that they implement some protocol for fetching the value associated with a given name. The value, if found, is loaded on to the evaluation stack using the *PUSH(v)* instruction otherwise, an error is set, and the execution jumps to the label for handling that error code.

As the flow chart shows, the *DISPATCH()* macro, an alias for the `continue` statement, is called after the value is pushed onto the evaluation stack.

The second diagram, labelled 2 in figure 9.1, shows the execution of the `LOAD_CONST`. Listing 9.9 is an implementation of the `LOAD_CONST` opcode.

Listing 9.9: LOAD_CONST opcode implementation

```
        PyObject *value = GETITEM(consts, oparg);
           Py_INCREF(value);
           PUSH(value);
           FAST_DISPATCH();
```

This goes through the standard setup as LOAD_GLOBAL but after execution, FAST_DISPATCH() is called rather than DISPATCH(). This causes a jump to the fast_next_opcode code label from where the loop execution continues skipping the signal and GIL checks on the next iteration. Opcodes that have implementations that make C function calls make of the DISPATCH macro while opcodes like the LOAD_GLOBAL that does not make C function calls in their implementation make use of the FAST_-DISPATCH macro. This means that the thread of execution can only yield the GIL after executing opcodes that make C function calls.

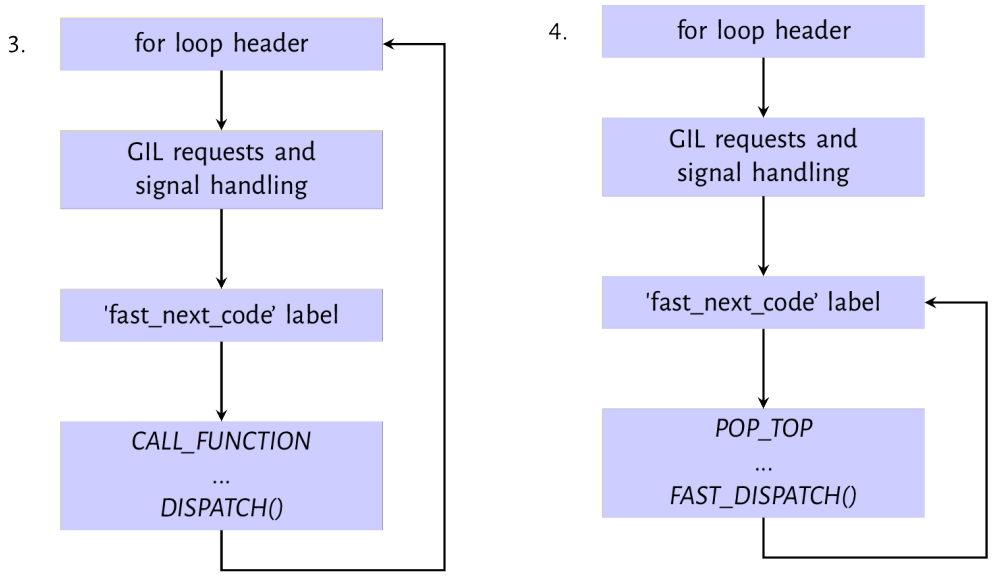


Figure 9.2: Evaluation path for CALL_FUNCTION and POP_TOP instruction

The next instruction is the CALL_FUNCTION opcode, as shown in the first image from figure 9.2. The compile emits this opcode for a function call with only positional arguments. Listing 9.10 is the implementation for this opcode. At the heart of the opcode implementation is the call_function(&sp, oparg, NULL). oparg is the number of arguments passed to the function, and the call_function function reads that number of values from the evaluation stack.

Listing 9.10: CALL_FUNCTION opcode implementation

```
    PyObject **sp, *res;
    PCALL(PCALL_ALL);
    sp = stack_pointer;
    res = call_function(&sp, oparg, NULL);
    stack_pointer = sp;
    PUSH(res);
    if (res == NULL) {
        goto error;
    }
    DISPATCH();
```

The next instruction shown in diagram 4 of figure 9.2 is the POP_TOP instruction that removes a single value from the top of the evaluation stack - this clears any value placed on the stack by the previous function call.

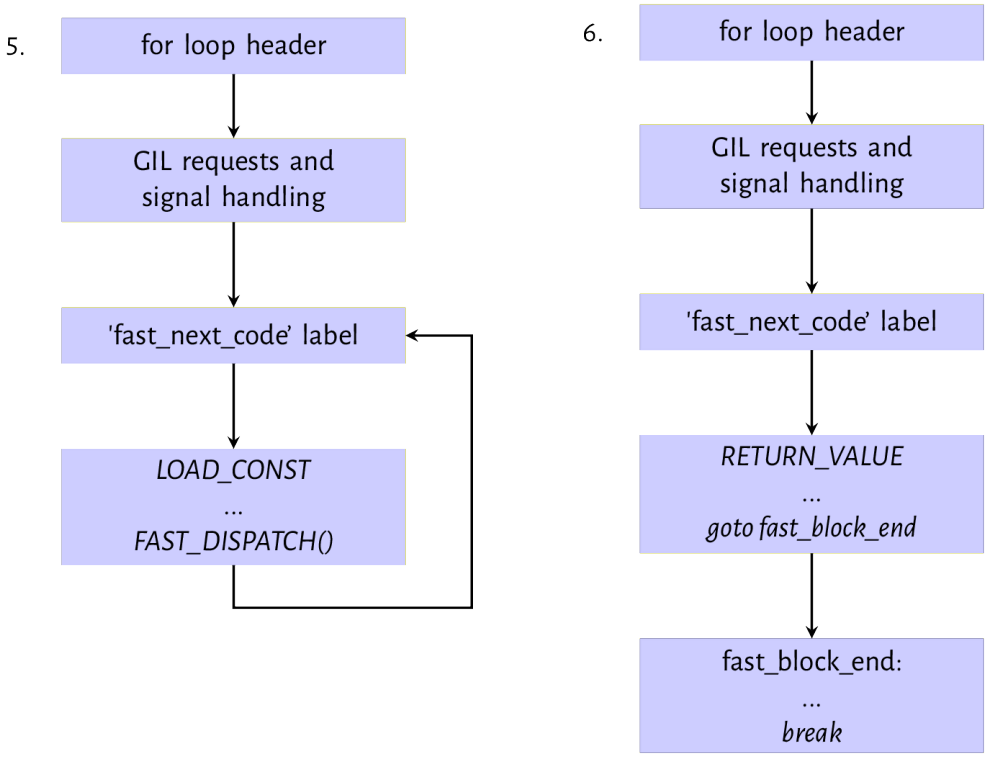


Figure 9.3: Evaluation path for LOAD_CONST and RETURN_VALUE instruction

The next set of instructions are the LOAD_CONST and RETURN_VALUE pair shown in diagrams 5 and 6 of figure 9.3. The LOAD_CONST opcode loads a None value onto the evaluation stack; this is the value that the RETURN_VALUE will return. These two always go together when a function does not explicitly return any value. We have already looked at the mechanics of the LOAD_CONST instruction. The RETURN_VALUE instruction pops the top of the stack into the retval variable, sets the WHY status

code to WHY_RETURN and then performs a jump to the fast_block_end code label where the execution continues. If there has been no exception, the execution breaks out of the for loop and returns the retval.

Notice that a lot of the code snippets that we have looked at have the goto error jump, but we have intentionally discussing errors and exceptions out so far. We will look at exception handling in the next chapter. Although the function's bytecode looked at in this section is relatively trivial, it encapsulates the vanilla behaviour of the evaluation loop. Other opcodes may have more complicated implementations, but the essence of the execution is the same as described above.

Next, we look at some other interesting opcodes supported by the python virtual machine.

9.4 A sampling of opcodes

The python virtual machine has about 157 opcodes, so we randomly pick a few opcodes and deconstruct to get more of a feel for how these opcodes function. Some examples of these opcodes include:

1. MAKE_FUNCTION: As the name suggests, the opcode creates a function object from values on the evaluation stack. Consider a module containing the functions shown in listing 9.11.

   Listing 9.11: Function definitions in a module

   ```
   def test_non_local(arg, *args, defarg="test",  defkwd=2, **kwd):
       local_arg = 2
       print(arg)
       print(defarg)
       print(args)
       print(defkwd)
       print(kwd)

   def hello_world():
       print("Hello world!")
   ```

   Disassembly of the code object from the module's compilation gives the set of bytecode instructions shown in listing 9.12

Listing 9.11: Disassembly of code object from listing 9.11

```
 17           0 LOAD_CONST               8 (('test',))
              2 LOAD_CONST               1 (2)
              4 LOAD_CONST               2 (('defkwd',))
              6 BUILD_CONST_KEY_MAP      1
              8 LOAD_CONST               3 (<code object test_non_local at 0x109eead0\
0, file "string", line 17>)
             10 LOAD_CONST               4 ('test_non_local')
             12 MAKE_FUNCTION            3
             14 STORE_NAME               0 (test_non_local)

 45          16 LOAD_CONST               5 (<code object hello_world at 0x109eeae80,\
 file "string", line 45>)
             18 LOAD_CONST               6 ('hello_world')
             20 MAKE_FUNCTION            0
             22 STORE_NAME               1 (hello_world)
             24 LOAD_CONST               7 (None)
             26 RETURN_VALUE
```

We can see that the `MAKE_FUNCTION` opcode appears twice in the series of bytecode instructions - one for each function definition within the module. The implementation of the `MAKE_FUNCTION` creates a function object and then stores the function in the `local` namespace using the function definition name. Notice that default arguments are pushed on the stack when such arguments are defined. The `MAKE_FUNCTION` implementation consumes these values by *and'ing* the `oparg` with a bitmask and popping values from the stack accordingly.

Listing 9.12: `MAKE_FUNCTION` opcode implementation

```
    TARGET(MAKE_FUNCTION) {
        PyObject *qualname = POP();
        PyObject *codeobj = POP();
        PyFunctionObject *func = (PyFunctionObject *)
            PyFunction_NewWithQualName(codeobj, f->f_globals, qualname);

        Py_DECREF(codeobj);
        Py_DECREF(qualname);
        if (func == NULL) {
            goto error;
        }

        if (oparg & 0x08) {
            assert(PyTuple_CheckExact(TOP()));
            func ->func_closure = POP();
        }
```

```
        if (oparg & 0x04) {
            assert(PyDict_CheckExact(TOP()));
            func->func_annotations = POP();
        }
        if (oparg & 0x02) {
            assert(PyDict_CheckExact(TOP()));
            func->func_kwdefaults = POP();
        }
        if (oparg & 0x01) {
            assert(PyTuple_CheckExact(TOP()));
            func->func_defaults = POP();
        }

        PUSH((PyObject *)func);
        DISPATCH();
    }
```

The flags above denote the following.

1. `0x01`: a tuple of default argument objects in positional order is on the stack.
2. `0x02`: a dictionary of keyword-only parameters default values is on the stack.
3. `0x04`: an annotation dictionary is on the stack.
4. `0x08`: a tuple containing cells for free variables, making a closure is on the stack.

The `PyFunction_NewWithQualName` function that actually creates a function object is implemented in the `Objects/funcobject.c` module and its implementation is pretty simple. The function initializes a function object and sets values on the function object.

2. `LOAD_ATTR`: This opcode handles attribute references such as `x.y`. Assuming we have an instance object `x`, an attribute reference such as `x.name` translates to the set of opcodes shown in listing 9.13.

Listing 9.13: Opcodes for an attribute reference

```
        24 LOAD_NAME                1 (x)
        26 LOAD_ATTR                2 (name)
        28 POP_TOP
        30 LOAD_CONST               4 (None)
        32 RETURN_VALUE
```

The `LOAD_ATTR` opcode implementation is pretty simple and shown in listing 9.14.

Listing 9.14: LOAD_ATTR opcode implementation

```
TARGET(LOAD_ATTR) {
    PyObject *name = GETITEM(names, oparg);
    PyObject *owner = TOP();
    PyObject *res = PyObject_GetAttr(owner, name);
    Py_DECREF(owner);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

We have looked at the PyObject_GetAttr function in the chapter on objects. This function returns the value of an object's attribute which is then loaded on to the stack. One can review the chapter on objects to get the details on how this function works.

3. CALL_FUNCTION_KW: This opcode very similar in functionality to the CALL_FUNCTION opcode that was discussed previously but is used for function calls with keyword arguments. Listing 9.15 is the implementation for this opcode. Notice how one of the significant change from the implementation of the CALL_FUNCTION opcode is that a tuple of names is now passed as one of the arguments when call_function is invoked.

Listing 9.15: CALL_FUNCTION_KW opcode implementation

```
PyObject **sp, *res, *names;

names = POP();
assert(PyTuple_CheckExact(names) && PyTuple_GET_SIZE(names) <= oparg);
PCALL(PCALL_ALL);
sp = stack_pointer;
res = call_function(&sp, oparg, names);
stack_pointer = sp;
PUSH(res);
Py_DECREF(names);

if (res == NULL) {
    goto error;
}
```

The names are the keyword arguments of the function call, and they are used in the _PyEval_EvalCodeWithName to handle the setup before the code object for the function is executed.

This caps our explanation of the evaluation loop. As we have seen, the concepts behind the evaluation loop are not complicated - opcodes each have implementations that in C. These implementations are the actual do work functions. Two critical areas that we have not touched are exception handling and the block stack, two intimately related concepts that we look at in the following chapter.

10. The Block Stack

One of the data structures that does not get as much coverage as it should is the block stack, the other stack within a frame object. Most discussions of the Python VM mention the block stack passingly but then focus on the evaluation stack. However, the block stack is critical for exception handling. There are probably other methods of implementing exception handling but as will become evident as we progress through this chapter, using a stack, the block stack, makes it incredibly simple to implement exception handling. The block stack and exception handling are so intertwined that one will not fully understand the need for the block stack without actually considering exception handling. Loops also make use of the block stack, but it is difficult to see a reason for block stacks with loops until one looks at how loop constructs like `break` interact with exception handlers. The block stack makes the implementation of such interactions a straightforward affair.

The block stack is a stack data structure field within a frame object. Just like the evaluation stack, values are pushed to and popped from the block stack during the execution of a frame's code. However, the block stack is used only for handling loops and exceptions. The best way to explain the block stack is with an example, so we illustrate with a simple `try...finally` construct within a loop as shown in the snippet in listing 10.0.

Listing 10.0: Simple python function with exception handling

```
    def test():
        for i in range(4):
            try:
                break
            finally:
                print("Exiting loop")
```

Listing 10.1 is the disassembly of code from Listing 10.0.

Listing 10.1: Disassembly of function in listing 10.0

```
  2           0 SETUP_LOOP              34 (to 36)
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_CONST               1 (4)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                22 (to 34)
             12 STORE_FAST               0 (i)

  3          14 SETUP_FINALLY            6 (to 22)
```

```
 4           16 BREAK_LOOP
             18 POP_BLOCK
             20 LOAD_CONST               0 (None)

 6     >>    22 LOAD_GLOBAL              1 (print)
             24 LOAD_CONST               2 ('Exiting loop')
             26 CALL_FUNCTION            1
             28 POP_TOP
             30 END_FINALLY
             32 JUMP_ABSOLUTE           10
       >>    34 POP_BLOCK
       >>    36 LOAD_CONST               0 (None)
             38 RETURN_VALUE
```

The combination of a `for` loop and `try ... finally` leads to a lot of instructions even for such a simple function as listing 10.1 shows. The opcodes of interest here are the `SETUP_LOOP` and `SETUP_FINALLY` opcodes so we look at the implementations of these to get the gist of what it does (all `SETUP_*` opcodes map to the same implementation).

The implementation for the `SETUP_LOOP` opcode is a simple function call - `PyFrame_BlockSetup(f, opcode, INSTR_OFFSET() + oparg, STACK_LEVEL());`. The arguments are pretty self-explanatory - `f` is the frame, `opcode` is the currently executing opcode, `INSTR_OFFSET() + oparg` is the instruction delta for the next instruction after that block (for the above code the delta is 50 for the SETUP_LOOP), and the `STACK_LEVEL` denotes how many items are on the value stack of the frame. The function call creates a new block and pushes it on the block stack. The information contained in this block is enough for the virtual machine to continue execution should something happen while in that block. Listing 10.2 is the implementation of this function.

Listing 10.2: Block setup code

```
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;
}
```

The handler in Listing 10.2 is the pointer to the next instruction that should be executed after the `SETUP_*` block. A visual representation of the example from above is illustrative - figure 10.0 shows this using a subset of the bytecode.

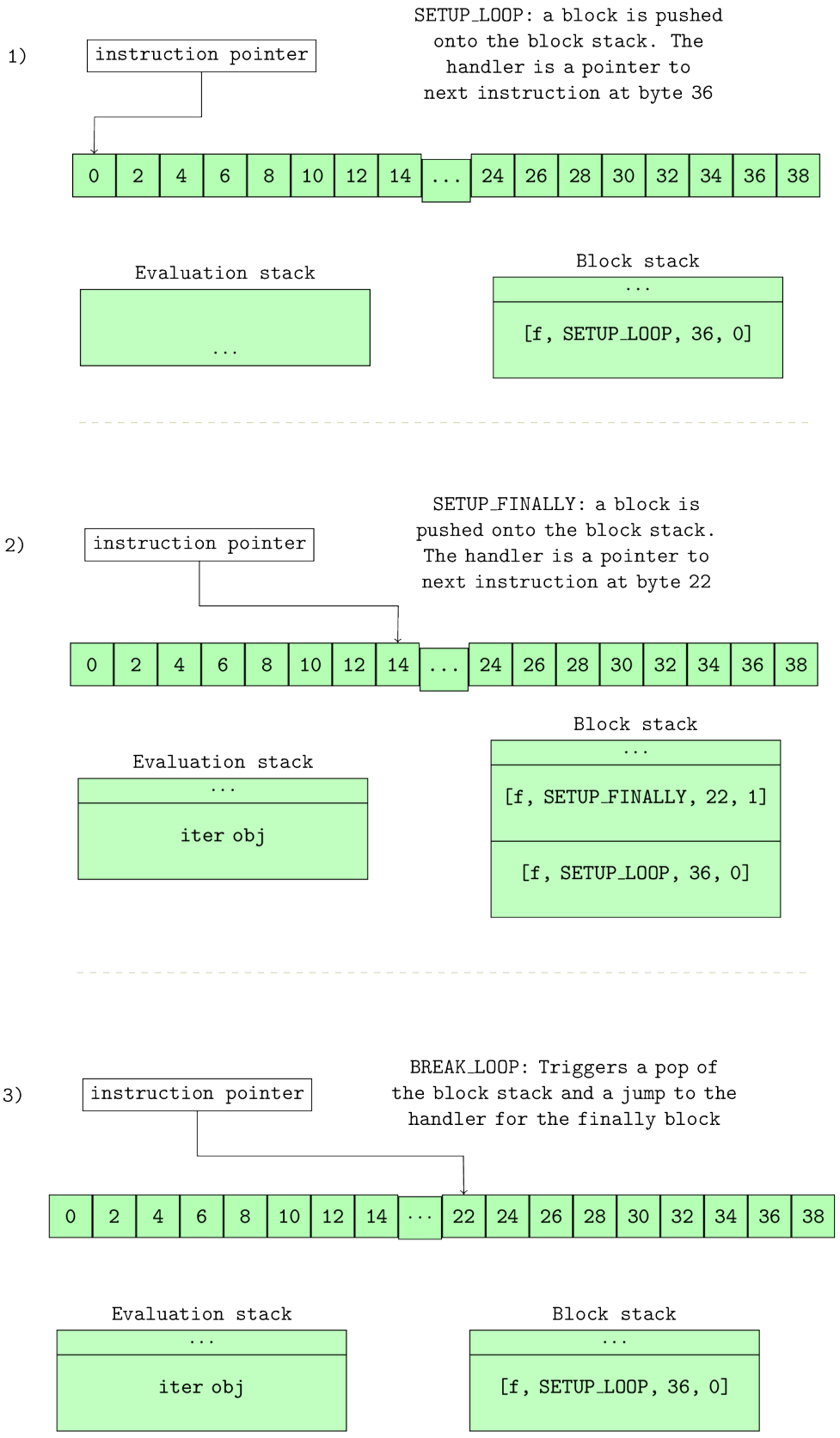


Figure 10.0: How the block stack changes with SETUP_* instructions

Figure 10.0 shows how the block stack varies during the execution of instructions.
The first diagram of figure 10.0 shows a single `SETUP_LOOP` block placed on the block stack after the execution of the `SETUP_LOOP` opcode. The instruction at offset 36 is the handler for this block, so when this stack is popped during normal execution, the interpreter will jump to that offset and continue execution. Another block is pushed onto the block stack during the execution of the `SETUP_FINALLY` opcode. We can see that as the stack is *Last In First Out* data structure, the `finally` block will be the first out - recall that finally must be executed regardless of the `break` statement.

A simple illustration of the use of a block stack is when the break statement is encountered while inside the exception handler within the loop. The `why` variable is set to `WHY_BREAK` during the execution of the `BREAK_LOOP` and a jump to the `fast_block_end` code label, where the block stack unwinding is handled, is performed.

The second diagram of figure 10.0 shows this. Unwinding is just a fancy name for popping blocks on the stack and executing their handler. So in this case, the `SETUP_FINALLY` block is popped off the stack, and the interpreter jumps to its handler at bytecode offset 22. Execution continues from that offset till the `END_FINALLY` statement. Since the `why` code is a `WHY_BREAK`; a jump is executed to the `fast_block_end` code label once again where more stack unwinding happens - the loop block is left on the stack. This time around (not shown in figure 10.0), the block popped from the stack has a handler at byte offset 36, so the execution continues at that bytecode offset, thus exiting the loop.

The use of the block stack dramatically simplifies the implementation of the virtual machine implementation. If loops are not implemented with a block stack, an opcode such as the `BREAK_LOOP` would need a jump destination. If we then throw in a `try..finally` construct with that `break` statement we get a complex implementation where we would have to keep track of optional jump destinations within `finally` blocks and so on.

10.1 A Short Note on Exception Handling

With this basic understanding of the block stack, it is not difficult to fathom the implementation of exceptions and exception handling. Take the snippet in Listing 10.3 that tries to add a number to a string.

Listing 10.3: Simple python function with exception handling

```
    def test1():
        try:
            2 + 's'
        except Exception:
            print("Caught exception")
```

Listing 10.4 shows the opcodes generated from this simple function.

Listing 10.4: Disassembly of function in listing 10.3

```
  2           0 SETUP_EXCEPT            12 (to 14)

  3           2 LOAD_CONST               1 (2)
              4 LOAD_CONST               2 ('s')
              6 BINARY_ADD
              8 POP_TOP
             10 POP_BLOCK
             12 JUMP_FORWARD            28 (to 42)

  4     >>   14 DUP_TOP
             16 LOAD_GLOBAL              0 (Exception)
             18 COMPARE_OP              10 (exception match)
             20 POP_JUMP_IF_FALSE       40
             22 POP_TOP
             24 POP_TOP
             26 POP_TOP

  5          28 LOAD_GLOBAL              1 (print)
             30 LOAD_CONST               3 ('Caught exception')
             32 CALL_FUNCTION            1
             34 POP_TOP
             36 POP_EXCEPT
             38 JUMP_FORWARD             2 (to 42)
        >>   40 END_FINALLY
        >>   42 LOAD_CONST               0 (None)
             44 RETURN_VALUE
```

We should have a conceptual idea of how this code block will execute if an exception should occur given the previous explanation. In summary, we expect the `PyNumber_Add` function from the `Objects/abstract.c` module to return a `NULL` for the `BINARY_ADD` opcode. In addition to returning a `NULL` value, the function also sets exception values on the currently executing thread's thread state. Recall, that the thread state has the `curexc_type`, `curexc_value` and `curexc_traceback` fields for holding the current exception in the thread of execution; these fields prove very useful while unwinding the block stack in search of exception handlers. You can follow the chain of function calls from the `binop_type_error` function in the `Objects/abstract.c` module all the way to the `PyErr_SetObject` and `PyErr_Restore` functions in the `Python/errors.c` module where the exception gets set on the thread state.

With the exception values set on the currently executing thread's thread state and a `NULL` value returned from the function call, the interpreter loop executes a jump to the error label where all the magic of exception handling occurs or not. For our trivial example above, we have only one block on the block stack, the `SETUP_EXCEPT` block with a handler at bytecode offset 14. The stack unwinding

begins after the jump to error handler label. The exception values - traceback, exception value and exception type, are pushed on top of the value stack, the `SETUP_EXCEPT` handler gets popped from the block stack, and then there is a jump to the handler - byte offset 14 in this case where the execution continues. Observe the bytecodes from offset 16 to offset 20 in listing 10.4; here the `Exception` class is loaded onto the stack, and then compared with the raised exception value present on the stack. If the exceptions match then normal execution can continue with the popping of exception and tracebacks from value stack and execution of any of the error handler code. When there is no exception match, the `END_FINALLY` instruction is executed, and since there is still an exception on the stack, there is a break from the exception loop.

In the case where there is no exception handling mechanism, the opcodes for the `test` function are more straightforward as shown in listing 10.5.

Listing 10.5: Disassembly of function in listing 10.3 when there is no exception handling

```
  2           0 LOAD_CONST               1 (2)
              2 LOAD_CONST               2 ('s')
              4 BINARY_ADD
              6 POP_TOP
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

The opcodes do not place any value on the block stack so when an exception followed by a jump to the error handling label occurs, there is no block to be unwound from the stack causing the loop to be exit and the error to be dumped to the user.

Although some implementation details are left out, this chapter covers the fundamentals of the interaction between the block stack and error handling in the Python virtual machine. There are other details such as handling exceptions within exceptions, nested exception handlers and so on. However, we conclude this short intermezzo at this point.

11. From Class code to bytecode

We have covered a lot of ground discussing the nuts and bolts of how the Python virtual machine or interpreter (whichever you want to call it) executes your code but for an object-oriented language like Python, we have left out one of the essential parts - the nuts and bolts of how a user-defined class gets compiled down to bytecode and executed.

Our discussions on Python objects have provided us with a rough idea of how new classes *may* be created; however, this intuition may not fully capture the whole process from the moment a user defines a `class` in source to the bytecode resulting from compiling that class. This chapter aims to bridge that gap and provide an exposition on this process.

As usual, we start with the straightforward user-defined class listing 11.0.

Listing 11.0: A simple class definition

```
    class Person:

        def __init__(self, name, age):
            self.name = name
            self.age = age
```

Listing 11.1 is the disassembly of the class from Listing 11.0.

Listing 11.1: A simple class definition

```
          0 LOAD_BUILD_CLASS
          2 LOAD_CONST               0 (<code object Person at 0x102298b70, file "str\
ing", line 2>)
          4 LOAD_CONST               1 ('Person')
          6 MAKE_FUNCTION            0
          8 LOAD_CONST               1 ('Person')
         10 CALL_FUNCTION            2
         12 STORE_NAME               0 (Person)
         14 LOAD_CONST               2 (None)
         16 RETURN_VALUE
```

We are interested in bytes 0 to 12, the actual opcodes that create the new class object and store it for future reference by its name (*Person* in our example). Before, we expand on the opcodes above; we look at the process of class creation as specified by the Python documentation[1]. The description of

[1] https://docs.python.org/3.6/reference/datamodel.html#customizing-class-creation

the process in the documentation, though done at a very high level, is pretty straightforward. We infer from Python documentation[2] that the class creation process involves the following steps.

1. The body of the class statement is isolated into a code object.
2. The appropriate metaclass for class instantiation is determined.
3. A class dictionary representing the namespace for the class is prepared.
4. The code object representing the class' body is executed within this namespace.
5. The class object is created.

During the final step, the class object is created by instantiating the `type` class, passing in the class name, base classes and class dictionary as arguments. Any `__prepare__` hooks are run before instantiating the class object. The metaclass used in the class object creation can be explicitly specified by supplying the `metaclass` keyword argument in the `class` definition. If a `metaclass` is not provided, the interpreter examines the first entry in the *tuple* of any base classes. If base classes are not used, the interpreter searches for the global variable `__metaclass__` and if not found, Python uses the default meta-class.

The whole class creation process starts with a load of the `__build_class` function onto the value stack. This function is responsible for all the type creation heavy lifting. Next, type's body code object, already compiled into a code object, is loaded on the stack by the instruction at offset 2 - `LOAD_CONST`. This code object is then wrapped into a function object by the `MAKE_FUNCTION` opcode and placed back on the stack; it will soon become clear why this happens. By offset 10; the evaluation stack looks similar to that in Figure 11.0.

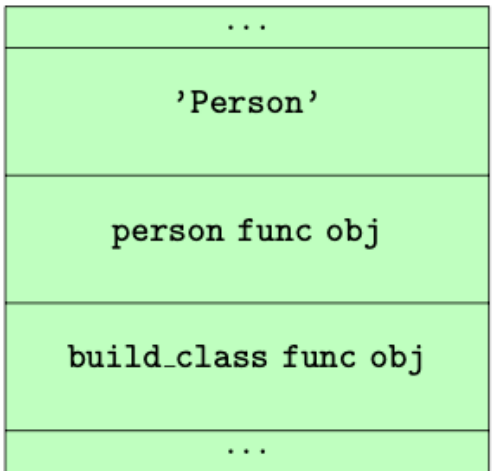


Figure 11.0: State of evaluation stack just before `CALL_FUNCTION`

At offset 10, `CALL_FUNCTION` handles invokes the `__build_class__` function with the two values above it on the evaluation stack as argument (the argument to `CALL_FUNCTION` is two). The `__build_class__` function in the `Python/bltinmodule.c` module is the workhorse that creates our class. A significant part of the function is devoted to sanity checks - check for the right arguments,

[2] https://docs.python.org/3.6/reference/datamodel.html#customizing-class-creation

checks for correct type etc. After these sanity checks, the function then has to decide on the right metaclass. The rules for determining the correct `metaclass` are reproduced verbatim from the Python documentation[3].

1. if no bases and no explicit metaclass are given, then `type()` is used
2. if an explicit metaclass is given and it is not an instance of `type()`, then it is used directly as the metaclass
3. if an instance of type() is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used

> The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(cls)`) of all specified base classes. The most derived metaclass is one which is a subtype of all of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

The actual snippet from the `__build_class` function that handles the metaclass resolution is in listing 11.2, and it has been annotated a bit more to provide some more clarity.

Listing 11.2: A simple class definition

```
    ...
        /* kwds are values passed into brackets that follow class name
        e.g class(metalcass=blah)*/
        if (kwds == NULL) {
            meta = NULL;
            mkw = NULL;
        }
        else {
            mkw = PyDict_Copy(kwds); /* Don't modify kwds passed in! */
            if (mkw == NULL) {
                Py_DECREF(bases);
                return NULL;
            }
            /* for all intent and purposes &PyId_metaclass references the string "me\
taclass"
            but the &PyId_* macro handles static allocation of such strings */

            meta = _PyDict_GetItemId(mkw, &PyId_metaclass);
            if (meta != NULL) {
                Py_INCREF(meta);
                if (_PyDict_DelItemId(mkw, &PyId_metaclass) < 0) {
                    Py_DECREF(meta);
                    Py_DECREF(mkw);
```

[3] https://docs.python.org/3.5/reference/datamodel.html#determining-the-appropriate-metaclass

```
                Py_DECREF(bases);
                return NULL;
            }
            /* metaclass is explicitly given, check if it's indeed a class */
            isclass = PyType_Check(meta);
        }
    }
    if (meta == NULL) {
        /* if there are no bases, use type: */
        if (PyTuple_GET_SIZE(bases) == 0) {
            meta = (PyObject *) (&PyType_Type);
        }
        /* else get the type of the first base */
        else {
            PyObject *base0 = PyTuple_GET_ITEM(bases, 0);
            meta = (PyObject *) (base0->ob_type);
        }
        Py_INCREF(meta);
        isclass = 1;  /* meta is really a class */
    }
    ...
```

With the metaclass found, `__build_class` then proceeds to check if any `__prepare__` attribute exists on the metaclass; if any such attribute exists the class namespace is *prepared* by executing the `__prepare__` hook passing the class name, class bases and any additional keyword arguments from the class definition. This hook is used to customize class behaviour. The following example in listing 11.3 which is taken from the example on metaclass definition and use of the python documentation[4] shows an example of how the `__prepare__` hook can be used to implement a class with attribute ordering.

Listing 11.3: A simple meta-class definition

```
class OrderedClass(type):

    @classmethod
    def __prepare__(metacls, name, bases, **kwds):
        return collections.OrderedDict()

    def __new__(cls, name, bases, namespace, **kwds):
        result = type.__new__(cls, name, bases, dict(namespace))
        result.members = tuple(namespace)
        return result
```

[4] https://docs.python.org/3.6/reference/datamodel.html#metaclass-example

```
    class A(metaclass=OrderedClass):
        def one(self): pass
        def two(self): pass
        def three(self): pass
        def four(self): pass

    >>> A.members
    ('__module__', 'one', 'two', 'three', 'four')
```

The __build_class function returns an empty new dict if there is no __prepare__ attribute defined on the metaclass but if there is one, the namespace that is used is the result of executing the __prepare__ attribute like the snippet in listing 11.4.

Listing 11.4: Preparing for a new class

```
    ...
        // get the __prepare__ attribute
        prep = _PyObject_GetAttrId(meta, &PyId___prepare__);
        if (prep == NULL) {
            if (PyErr_ExceptionMatches(PyExc_AttributeError)) {
                PyErr_Clear();
                ns = PyDict_New(); // namespace is a new dict if __prepare__ is not \
defined
            }
            else {
                Py_DECREF(meta);
                Py_XDECREF(mkw);
                Py_DECREF(bases);
                return NULL;
            }
        }
        else {
            /** where __prepare__ is defined, the namespace is the result of executi\
ng
            the __prepare__ attribute **/
            PyObject *pargs[2] = {name, bases};
            ns = _PyObject_FastCallDict(prep, pargs, 2, mkw);
            Py_DECREF(prep);
        }
        if (ns == NULL) {
            Py_DECREF(meta);
            Py_XDECREF(mkw);
            Py_DECREF(bases);
```

```
            return NULL;
        }
        ...
```

After handling the __prepare__ hook, it is now time for the actual class object to be created. First, the execution of the class body's code object happens within the namespace from the previous paragraph. To understand why take a look at this code object's bytecode in listing 11.5.

Listing 11.5: Disassembly of code object for class body from listing 11.0

```
 1           0 LOAD_NAME                0 (__name__)
             2 STORE_NAME               1 (__module__)
             4 LOAD_CONST               0 ('test')
             6 STORE_NAME               2 (__qualname__)

 2           8 LOAD_CONST               1 (<code object __init__ at 0x102a80660, fi\
le "string", line 2>)
            10 LOAD_CONST               2 ('test.__init__')
            12 MAKE_FUNCTION            0
            14 STORE_NAME               3 (__init__)
            16 LOAD_CONST               3 (None)
            18 RETURN_VALUE
```

After executing this code object, the namespace will contain all the attributes of the class, i.e. class attributes, methods etc. This namespace subsequently used as an argument for a function call to the metaclass as shown in listing 11.6.

Listing 11.6: Invoking a metaclass to create a new class instance

```
    // evaluate code object for body within namespace
     none = PyEval_EvalCodeEx(PyFunction_GET_CODE(func), PyFunction_GET_GLOBALS(func\
), ns,
                              NULL, 0, NULL, 0, NULL, 0, NULL,
                              PyFunction_GET_CLOSURE(func));
    if (none != NULL) {
        PyObject *margs[3] = {name, bases, ns};
        /**
         * this will 'call' the metaclass creating a new class object
         **/
        cls = _PyObject_FastCallDict(meta, margs, 3, mkw);
        Py_DECREF(none);
    }
```

Assuming we are using the `type` metaclass, calling the type means dereferencing the attribute in the `tp_call` slot of the class. The `tp_call` function then, in turn, dereferences the attribute in the `tp_new` slot which creates and returns our brand new class object. The `cls` value returned is then placed back on the stack and stored to the `Person` variable. There we have it, the process of creating a new class and this is all there is to it in Python.

12. Generators: Behind the scenes.

Generators are one of the beautiful concepts in Python. A generator function is one that contains a `yield` statement, and when called, it returns a generator. A simple use of generators in Python is as an iterator that produces values for an iteration on demand. Listing 12.0 is a simple example of a generator function that produces values from `0` up to `n`.

Listing 12.0: A simple generator

```
def firstn(n):
    num = 0
    while num < n:
        v = yield num
        print(v)
        num += 1
```

`firstn` contains the `yield` statement, so calling it will not return a simple value as a conventional function would do. Instead, it will return a generator object which captures the **continuation** of the computation. We can then use the `next` function to get successive values from the returned generator object or send values into the generator using the `send` method of the generator object.

In this chapter, we are not interested in the semantics of the generators objects or the right way to use them. Our interest is in how generators are implemented under the covers in CPython. We are interested in how it is possible to suspend a computation and then subsequently resume such computation. We look at the data structures and ideas behind this concept, and surprisingly, they are not too complicated. First, we look at the C implementation of a generator object.

12.1 The Generator object

Listing 12.1 is the definition of a generator object, and going through this definition provides some intuition into how a generator execution can be suspended or resumed. We can see that a generator object contains a *frame* object and a code object, two objects that are essential to the execution of Python bytecode.

Listing 12.1: Generator object definition

```
/* _PyGenObject_HEAD defines the initial segment of generator
and coroutine objects. */
#define _PyGenObject_HEAD(prefix)                                           \
    PyObject_HEAD                                                           \
    /* Note: gi_frame can be NULL if the generator is "finished" */         \
    struct _frame *prefix##_frame;                                          \
    /* True if generator is being executed. */                              \
    char prefix##_running;                                                  \
    /* The code object backing the generator */                             \
    PyObject *prefix##_code;                                                \
    /* List of weak reference. */                                           \
    PyObject *prefix##_weakreflist;                                         \
    /* Name of the generator. */                                            \
    PyObject *prefix##_name;                                                \
    /* Qualified name of the generator. */                                  \
    PyObject *prefix##_qualname;

typedef struct {
    /* The gi_ prefix is intended to remind of generator-iterator. */
    _PyGenObject_HEAD(gi)
} PyGenObject;
```

The following comprise the main attributes of a generator object.

1. `prefix##_frame`: This field references a frame object. This frame object contains the code object of a generator and it is within this frame that the execution of the generator object's code object takes place.
2. `prefix##_running`: This is a boolean field that indicates whether the generator is running.
3. `prefix##_code`: This field references the code object associated with the generator. This is the code object that executes whenever the generator is running.
4. `prefix##_name`: This is the name of the generator - in listing 12.0, the value is `firstn`.
5. `prefix##_qualname`: This is the fully qualified name of the generator. Most times this value is the same as that of `prefix##_name`.

Creating generators

When we call a generator function, the generator function does not run to completion and return a value; instead, it produces a generator object. This is due to the `CO_GENERATOR` flag that gets set when compiling a generator function. This flag comes in very useful during the setup process that happens just before the code object execution.

During the execution of the code object for the function, recall the `_PyEval_EvalCodeWithName` is invoked to perform some setup. During this setup process, the interpreter checks if the `CO_GENERATOR` flag; if set, it creates and returns a generator object rather than call the evaluation loop function. The magic happens at the last code block of the `_PyEval_EvalCodeWithName` as shown in listing 12.2.

Listing 12.2: _PyEval_EvalCodeWithName returns a generator object when processing a code object with generator flag

```
/* Handle generator/coroutine/asynchronous generator */
    if (co->co_flags & (CO_GENERATOR | CO_COROUTINE | CO_ASYNC_GENERATOR)) {
        PyObject *gen;
        PyObject *coro_wrapper = tstate->coroutine_wrapper;
        int is_coro = co->co_flags & CO_COROUTINE;

        if (is_coro && tstate->in_coroutine_wrapper) {
            assert(coro_wrapper != NULL);
            PyErr_Format(PyExc_RuntimeError,
                         "coroutine wrapper %.200R attempted "
                         "to recursively wrap %.200R",
                         coro_wrapper,
                         co);
            goto fail;
        }

        /* Don't need to keep the reference to f_back; it will be set
         * when the generator is resumed. */
        Py_CLEAR(f->f_back);

        PCALL(PCALL_GENERATOR);

        /* Create a new generator that owns the ready to run frame
         * and return that as the value. */
        if (is_coro) {
            gen = PyCoro_New(f, name, qualname);
        } else if (co->co_flags & CO_ASYNC_GENERATOR) {
            gen = PyAsyncGen_New(f, name, qualname);
        } else {
            gen = PyGen_NewWithQualName(f, name, qualname);
        }
        if (gen == NULL)
            return NULL;

        if (is_coro && coro_wrapper != NULL) {
            PyObject *wrapped;
            tstate->in_coroutine_wrapper = 1;
```

```
            wrapped = PyObject_CallFunction(coro_wrapper, "N", gen);
            tstate->in_coroutine_wrapper = 0;
            return wrapped;
        }

        return gen;
    }
```

We can see from Listing 12.2 that bytecode for a generator function code object is never executed at the point of the function call - the execution of bytecode only happens when the returned generator object is running, and we look at this next.

12.2 Running a generator

We can run a generator object by passing it as an argument to the `next` builtin function. This will cause the generator to execute until it hits a `yield` expression then it suspends execution. The critical question here is how the generators can capture the execution state and update those at will.

Looking back at the generator object definition in Listing 12.1, we see that generators have a field that references a frame object, and this gets filled when the generator is created as shown in listing 12.2. The frame object as we recall has all the state that is required to execute a code object so by having a reference to that execution frame, the generator object can capture all the state required for its execution.

Now that we know how a generator object captures execution state, we move to the question of how the execution of a suspended generator object is resumed, and this is not too hard to figure out given the information that we have already. When the `next` builtin function is called with a generator as an argument, the `next` function dereferences the `tp_iternext` field of the generator type and invokes whatever function that field references. In the case of a generator object, that field references a function, `gen_iternext`, which calls the `gen_send_ex` function, that does the actual work of resuming the execution of the generator object. Before the generator object was created, the initial setup of the frame object and variables was carried out by the `_PyEval_EvalCodeWithName` function, so the execution of the generator object involves calling the `PyEval_EvalFrameEx` with the frame object contained within the generator object as the frame argument. The execution of the code object contained within the frame then proceeds as explained the chapter on the evaluation loop.

To get a more in-depth look at a generator function, we look at the generator function in listing 12.0. The disassembly of the generator function in listing 12.0 results in the set of bytecode shown in listing 12.3.

Listing 12.3: Disassembly of generator function from listing 12.0

```
 4           0 LOAD_CONST               1 (0)
             2 STORE_FAST               1 (num)

 5           4 SETUP_LOOP              34 (to 40)
       >>    6 LOAD_FAST                1 (num)
             8 LOAD_FAST                0 (n)
            10 COMPARE_OP               0 (<)
            12 POP_JUMP_IF_FALSE       38

 6          14 LOAD_FAST                1 (num)
            16 YIELD_VALUE
            18 STORE_FAST               2 (v)

 7          20 LOAD_GLOBAL              0 (print)
            22 LOAD_FAST                2 (v)
            24 CALL_FUNCTION            1
            26 POP_TOP

 8          28 LOAD_FAST                1 (num)
            30 LOAD_CONST               2 (1)
            32 INPLACE_ADD
            34 STORE_FAST               1 (num)
            36 JUMP_ABSOLUTE            6
       >>   38 POP_BLOCK
       >>   40 LOAD_CONST               0 (None)
            42 RETURN_VALUE
```

When the execution of the bytecode shown in listing 12.3 for the generator function gets to the `YIELD_VALUE` opcode at byte offset 16, that opcode causes the evaluation to suspend and return the value on the top of the stack to the caller. By suspend, we mean the evaluation loop for the currently executing frame is exited however this frame is not deallocated because it is still referenced by the generator object so the execution of the frame can continue again when `PyEval_EvalFrameEx` is invoked with the frame as one of its arguments.

Python generators do more than just **generate** values; they can also consume values by using the generator `send` method. This is possible because `yield` is an expression that evaluates to a value. When the send method is called on a generator with a value, the `gen_send_ex` method places the value onto the evaluation stack of the generator object frame before the evaluation of the frame object resumes. Listing 12.3 shows the `STORE_FAST` instruction comes after `YIELD_VALUE`; this stores the value at the top of the stack to the provided name. In the case where there is no `send` function call, then the None value is placed on the top of the stack.