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 2018-11-05

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.

© 2015 - 2018 Obi Ike-Nwosu

Also By Obi Ike-Nwosu

Intermediate Python

Contents

1. Introduction

The Python Programming language has been around for quite a while. Development work was started on the first version by Guido Van Rossum in 1989 and it has since grown to become one of the more popular languages that has been used in applications ranging from graphical interfaces to financial[1] and data analysis[2] applications.

This write-up aims to go behind the scene of the Python interpreter and provide a conceptual overview of how a python program is executed. This material targets `CPython` which as of this writing is the most popular implementation of Python and is considered the standard.

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

I like to think of the execution of a python program as split into two or three main phases as listed below depending on how the interpreter is invoked. These are covered in different measures within this write-up:

1. Initialization : This involves the set up of the various data structures needed by the python process. This will probably only counts when a program is being executed non-interactively through the interpreter shell.
2. Compiling : This involves activities such as parsing source code to build syntax trees, creation of abstract syntax trees, building of symbol tables and generation of code objects.
3. Interpreting : This involves the actual execution of generated code objects within some context.

The process of generating parse trees and abstract syntax trees from source code is language agnostic so the same methods that apply to other languages also apply to Python; as a result, not much is on this subject is covered here. On the other hand, the process of building symbol tables and code objects from the *Abstract Syntax tree* is the more interesting part of the compilation phase which is handled in a more or less python specific way and attention is paid to it. The interpreting of compiled code objects and all the data structures that are used in the process is also covered. Topics that will be touched upon include but are not limited to the process of building symbol tables and generating code objects, python objects, frame objects, code objects, function objects, python opcodes, the interpreter loop, generators and user defined classes.

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

This material is aimed at anybody that is interested in gaining some insight into how the CPython virtual machine functions. It is assumed that the user is already familiar with python and understands the fundamentals of the language. As part of this expose on the virtual machine, we go through a considerable amount of `C` code so a user that has a rudimentary understanding of `C` will find it easier to follow. After all said and done, all that is needed to get through this material is a healthy desire to want 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 substantial amount of wisdoms in videos available in Pycon videos[3], school lectures[4] and blog write-ups[5]. This work will not be complete without acknowledging these fantastic sources that have been leveraged in the production of this work.

At the end of this book, a user should be able to understand the intricacies of how the Python interpreter executes a program. This includes the various steps involved in executing the program and the various data structures that are crucial to the execution of such program. We start off with a gentle bird's eye view of what happens when a trivial program is executed by passing the module name to the interpreter at the commandline. The `CPython` executable can be installed from source by following the instructions at the Python Developer's Guide[6].

Python version 3 is used throughout this material.

[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 provides a high level expose on how the interpreter goes about executing a python program. In subsequent chapters, we zoom in on the various pieces of puzzle and provide a more detailed description of such pieces. Regardless of the complexity of a python program, this process is the same. 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.

Given a python module, `test.py`, this module can be executed at the command line by passing it as an argument to the python interpreter program as such `$python test.py`. This is just one of the ways of the invoking the python executable - we could start the interactive interpreter, execute a string as code etc but these other methods of execution are not of interest to us. When the module is passed as an argument to the executable on the command line, figure 2.1 best captures the flow of various activities that are involved in the actual execution of the supplied module.

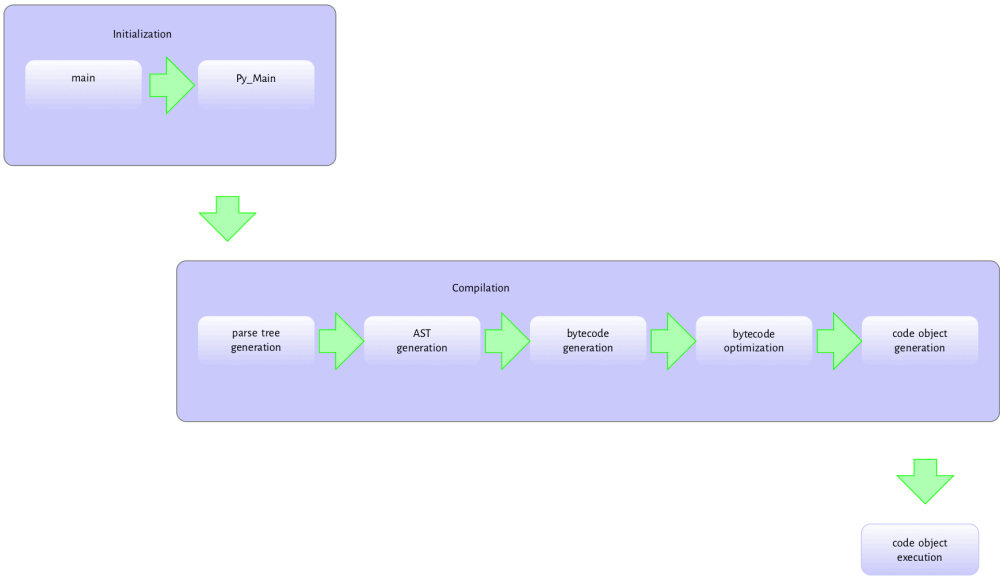


Figure 2.1: Flow during execution of source code

The python executable is a `C` program just 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 the python executable is invoked. Take a moment to grasp this, the python executable is just another program that runs your own *program*. The same argument can be made for the relationship between `C` and `assembly` or `llvm`. The standard process initialization which depends on the platform the executable is running on starts once the python executable is invoked with module name as argument,

[1] https://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/

This writeup assumes a unix based operating system so some speicifics may differ when a windows operating system is being used.

The `C` runtime performs all its initialisation magic - libraries are loaded, environment variables are checked or set then the python executable's main method is run just like any other `C` program. The python executable's `main` program is located in the `./Programs/python.c` file and it handles some initialization such as making copies of program command line arguments that were passed to the module. The `main` function then calls the `Py_Main` function located in the `./Modules/main.c` which handles the interpreter initialization process - parsing commandline arguments and setting program flags[2], reading environment variables, running hooks, carrying out hash randomization etc. As part of the initialization process, `Py_Initialize` from `pylifecycle.c` is called; this handles the initialization of the interpreter and thread state data structures - two very important data structures.

A look at the data structures definitions for the interpreter and thread states provides some context into the functions of these data structures. The interpreter and thread states are just structures with pointers to fields that hold information that is required for the execution of a program. The interpreter state `typedef` (just assume that this is `C` lingo for type definition though this is not entirely true) is provided in listing 2.1.

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)*.

[2] https://docs.python.org/3.6/using/cmdline.html

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 - in the event that the python program is multithreaded then the interpreter is shared by all threads created by the program - the structure of a thread state is discussed 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 for all python objects in the virtual machine world. Python objects are covered in more detail in the chapters that will follow.
4. The `codec*` related fields hold information that help with the location and loading of encodings. These are very important for decoding bytes.

The execution of a program must occur within a thread. The thread state structure contains all the information that is needed by a thread to execute python some code object - a part of the thread data structure is shown in listing 2.2.

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;

```

```
27          PyObject *dict;  /* Stores per-thread state */
28          int gilstate_counter;
29
30          ...
31      } PyThreadState;
```

The interpreter and the thread state data structures are discussed in more details in subsequent chapters. The initialization process also sets up the import mechanisms as well as rudimentary stdio.

Once all the initialization is complete, the `Py_Main` function invokes the `run_file` function also located 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 if a `pyc` version of the file exists - the `pyc` file is just a file containing the compiled version of the file being executed. In the case that the file has a `pyc` version then an attempt will be made to read it in as binary and then run it. In this case, there is no `pyc` file so the `PyRun_FileExFlags` is called and so on. The `PyParser_ASTFromFileObject` function calls the `PyParser_ParseFileObject` which reads the module content and builds a parse tree from it. The parse tree created is then passed to the `PyParser_ASTFromNodeObject` which then goes ahead to create an abstract syntax tree from the parse tree.

If you have been following the actual `C` source code by now 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 details. CPython manages the object life cycle using reference counting; whenever a new reference to an object is made the reference is increased with the `Py_INCREF` while whenever a reference goes out of scope the reference is reduced with the `Py_DECREF` functions.

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 the code objects are created. The `run_mod` function then invokes `PyEval_EvalCode` from the `ceval.c file` on the code object. This results in another series of function call: `PyEval_EvalCode->PyEval_EvalCode->_PyEval_EvalCodeWithName->_PyEval_EvalFrameEx` function calls. The code object is passed as an argument into most each of these functions in one form or another. The `_PyEval_EvalFrameEx` is the atual interpreter loop that handles the execution of code objects. It is however not just invoked with a code object as argument rather a *frame object* with has a field that references a code object is one of its arguments. This *frame object* provides the context for the execution of the code object. A very simplified version of what happens here is that the interpreter loop continuously reads the next instruction pointed to by the instruction counter from an array of instructions. It then executes this instruction - adding or removing objects from the value stack in the process (*where is this value*

stack), till there are no more instructions to be executed in the array 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 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

```
1         >>> def square(x):
2         ...     return x*x
3         ...
4
5         >>> dis(square)
6         2           0 LOAD_FAST                0 (x)
7                     2 LOAD_FAST                0 (x)
8                     4 BINARY_MULTIPLY
9                     6 RETURN_VALUE
```

The `./Include/opcodes.h` header file contains a full listing of all the instruction/opcodes for the python virtual machine. The opcodes are pretty straight forward conceptually. Take our example from listing 2.3 that has a set of four instructions - the LOAD_FAST loads the value of the its argument (`x` in this case) onto an evaluation (value) stack. The python virtual machine is a stack based virtual machine so this means that values for evaluations by an opcode are gotten from a stack and results of an evaluation are placed back on the stack for further use by other opcodes. The `BINARY_MULTIPLY` opcode then pops two items from the value stack, performs binary multiplication on both values and places the result of the binary multiplication back on the value stack. The `RETURN VALUE` instruction pops a value from the stack, sets the return value to 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 number of details that will be discussed in subsequent chapters. 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 that are used as part of instructions come from ?
3. How are nested function and method calls managed ? 4 How does the interpreter loop handle exceptions ?

After all the instructions have been the executed, the `Py_Main` function continues its execution but this time around it starts the clean up process. Just as `Py_Initialize` was called to perform

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 and also freeing up any memory allocated by the interpreter that is still in use.

The above description is a high-level description of the processes the python executable goes through to execute a *python* program. As noted previously, alot of details are stil left to be answered and in the chapters that will follow, we will dig into each of the stages that have been covered and try to provide details on each of these stages. We get into action starting with a description of the compilation process in the next chapter.

3. Compiling Python Source Code

Although python is not popularly regarded as a compiled language, it actually is one. During compilation, some python source code is transformed into bytecode that is executable by the virtual machine. This compilation process in python is however a straightforward process that does not involve *lots* of complicated steps. The compilation process of a python program involves the following steps in the given order.

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

Parsing source code into a parse tree and transforming that parse tree into an AST 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 a much more indepth *tour de force* of both topics.

3.1 From Source To Parse Tree

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

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

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: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^='
                | '<<=' | '>>=' | '**=' | '//=')

        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]

        ...
```

When executing a module passed to the interpreter on the command line, a call to the `PyParser_ParseFileObject` function initiates the parsing of the module. This function calls the tokenization function, `PyTokenizer_FromFile`, passing the module file name as argument. The tokenization function breaks up the content of the module into legal python tokens or throws an exception when an illegal value is found.

3.2 Python tokens

Python source code is made up of tokens. For example, the `return` word is a keyword token, the literal `- 2` is a numeric literal token and so on. The first task while parsing python source is to tokenize the source that is break it into its component tokens. Python has a number of tokens such as the following.

1. identifiers : These are names that defined by a programmer. They include function names, variable names, class names etc. These must conform to the rules of identifiers as specified in the python documentation.
2. operators: these are special symboles 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 special token that denotes the end of a logical line.
7. INDENT and DEDENT: These token are used to represent indentation levels which group compound statements.

A group of tokens delineated by the **NEWLINE** token make up a logical line hence we could say that a python program is made up of a sequence of *logical lines* with each logical line delineated by the **NEWLINE** token. These logical lines maps to python statements. Each of these logical lines are made up of a number of physical lines that are each terminated by an end-of-line sequence. In python, most times logical lines map to physical line so we have logical line delimited by end-of-line characters. Compound statements can span multiple physical lines such as shown in figure 3.0. Logical lines can be joined together implicitly when expression are in parentheses, square brackets or curly braces or explicitly by the use of the backslash character. Indentation also plays a central role in grouping python statements. One of lines in the python grammar is thus `suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT` so one of the major task of the python tokenizer is to generate indent and dedent tokens that go into the parse tree. The tokenizer uses a stack to keep track of indents and uses the algorithm in listing 3.1 to generate 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 line's indentation report an error.
                2.  For each value at the top of the stack that is unequal to
                the current line'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/parsetok.c` module scans the python source file from left to right and top to bottom tokenizing the content of the file. Whitespaces characters other than terminators serve to delimit tokens but are not compulsory. Where there is some ambiguity such as in `2+2`, a token comprises the the longest possible string that forms a legal token reading from right to left; in this example the tokens are the literal `2`, the operator `+` and the literal `2`.

The tokens generated from the tokenizer are passed to the parser which attempts to build a parse tree according to the python grammar of which a subset is specified in listing 3.0. When the parser encounters a token that violates the grammar, a `SyntaxError` exception is thrown. The output from the parser is a parse tree. The `parser`[3] python module provides limited access to the parse tree of a block of python code and it is used in listing 3.2 to get a concrete demonstration of parse trees.

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,
```

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

```
        [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,
                        [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 above returns a parse tree `(ST)` object, a python intermediate representation for a parse tree, from the supplied source code so long as the source code is syntactically correct. The call to `parser.st2list` returns the actual parse tree represented in form of a python list. The first items in the lists, the integer, identifies the production rule in the python grammar.

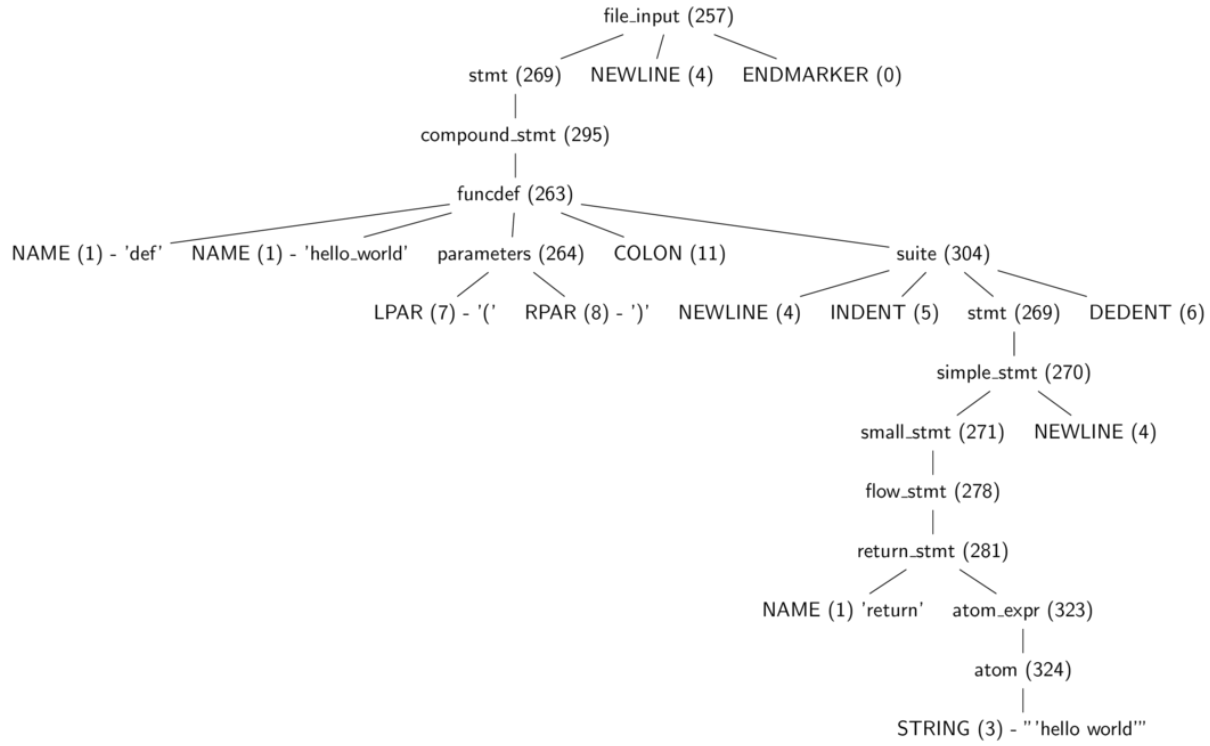


Figure 3.0: A parse tree for listing 3.2 (function that returns the 'hello world' string)

Figure 3.0 is a tree diagram showing the same parse tree from listing 3.2 with some tokens stripped away and one can see 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.

In the CPython virtual machine, a tree data structure is used to represent the parse tree. Each production rule is a *node* on the tree data structure. This node data structure is shown in listing 3.3 from the `Include/node.h`.

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

```
1       typedef struct _node {
2           short               n_type;
3           char               *n_str;
4           int                       n_lineno;
5           int                       n_col_offset;
6           int                       n_nchildren;
7           struct _node       *n_child;
8       } node;
```

As the parse tree is walked, the nodes can be queried for their type, their children if any, the line number that led to the creation of the given node and so on. The macros for interacting with parse

tree nodes are also defined in the `Include/node.h` file.

3.3 From Parse Tree To Abstract Syntax Tree

The next stage of the compilation process is the transformation of a python parse trees to an *Abstract Syntax Tree* (AST). The abstract syntax tree is a representation of the code that is independent of the niceties of python's syntax. For example, a parse tree contains the colon node as shown in figure 3.0 because it is a syntax construct but the AST will not contain such syntax construct as shown in listing 3.4.

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)])')
```

The definitions of the AST nodes for the various Python are found in the file `Parser/Python.asdl` file. Most definitions in the AST correspond to a particular source construct such as an `if` statement or an attribute lookup. 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. In the CPython implementation, the `AST` nodes are represented by `C` structures as defined in the `Include/Python-ast.h`. These structures are actually generated by python code; the `Parser/asdl_c.py` module generates this file from the `AST` asdl definition. For example a cross-section of the definition of a `statement` node is shown in listing 3.5.

Listing 3.5: A cross-section of an AST statement node data structure

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

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

```
                } 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 the listing 3.5 is a `C` type that is used to represent a type that can take one any of the types listed in the union. The `PyAST_FromNode` function in the `Python/ast.c` module handles the generation of the AST from a given parse tree. With the AST generated, it is now time for the generation of bytecode from the AST.

3.4 Building The Symbol Table

With the `AST` generated, the next step of the process is the generation of a symbol table. The symbol table just like the name suggest is a collection of the names within a code block and the context in whic such names are used. The process of building the symbol table involves the analysis of the names contained within a code block and the assignment of the correct scoping to such names. Before discussing the intricacies of the symbol tables generation, it maybe worth going over names and bindings in python.

Names and Binding

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

```
>>> 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 a number of 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 just references to objects that actually have types**.

Code Blocks

Code blocks are central to python program and understanding them for what they are is paramount to understanding the internals of the python virtual machine. A code block is a piece of program code that is executed 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 a number of name-spaces associated with it. For example, a module code block has access to the `global` name-space while a function code block has access to the `local` as well as the `global` name-spaces.

Namespaces

A *namespace* as the name implies is a context in which a given set of names is bound to objects. Namespaces are implemented as dictionary mappings in python. The *builtin* namespace is an example of a name-space 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 name-space*, *the builtin namespace* and *the local namespace*. These namespaces are created at different times and have different lifetimes. For example, a new *local* namespace is created at the invocation of a function and forgotten 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 name-spaces 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. Inner most 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
4. The scope containing the builtin namespace.

When a name is used in python, the interpreter searches the namespaces of the scopes in ascending order as listed above and if the name is not found in any of the namespaces, an exception is raised. 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 prevents a reference to an object in the *global* scope from being modified in 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 has to be used with the object name before modification is attempted. The following example illustrates this.

Listing A3.0: 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
```

In order to modify the object from the global scope, the `global` statement is used as shown in the following snippet.

Listing A3.1: 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 that is used when there is a need to modify a variable bound in an outer non-global scope from an inner scope. This proves very handy when working with nested functions (also referred to as closures). A very trivial illustration of the `nonlocal` keyword in action is shown in the following snippet that defines a simple counter object that counts in ascending order.

Listing A3.2: 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
```

The series of function calls `run_mod -> PyAST_CompileObject -> PySymtable_BuildObject` triggers the process of building a symbol table. Two of the arguments to the `PySymtable_BuildObject` function are the AST previously generated and the file name of the module. The algorithm for building the symbol table is split into two parts. During the first part of the process, each node of the `AST` that is passed as argument to the `PySymtable_BuildObject` is visited in order to build up a collection of symbols used within the `AST`. A very simple description of this process is given in listing 3.6 and the terms used in this will be more obvious when we discuss the data structures used in building a symbol table.

Listing 3.6: Creating a symbol table from an AST

```
    For each node in a given AST
        If node is start of a code block:
            1. create new symbol table entry and set the current symbol table
            to this value.
            2. push new symbol table to st_stack.
            3. add new symbol table to the list of children of previous symbol
            table.
            4. update 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 current symbol table entry from
            the stack.
            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.
```

After the first pass of algorithm on the `AST`, the symbol table entries contain all names that have been used within the module but they do not have contextual information about such names. For example, the interpreter cannot tell if a given variable is a global, local or free variable. A call to the `symtable_analyze` function in the `Parser/symtable.c` initiates the second phase of the symbol table generation. This phase of 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 an 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.
>
> When a function is entered during the second pass, the parent passes the set of all name bindings visible to its children. These bindings are used to determine if non-local variables are free or implicit globals. Names which are explicitly declared nonlocal must exist in this set of visible names - if they do not, a syntax error is raised. After doing 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 to; then it is treated as a local.
>
> The children update the free variable set. If a local variable is added to the free variable set by the child, the variable is marked as a cell. The function object being 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.

Although the comments try to explain the process in clear language, there are some confusing aspects such *the parent passes the set of all name bindings visible to its children* - what parent and what children are being referred to? To get an understanding of such terminology, we have to look at the data structures that are used to in the process of creating a symbol table.

Symbol table data structures

Two data structures that are central to the generation of a symbol table are:

1. The symbol table data structure.
2. The symbol table entry data structure.

The symbol table data structure is shown in listing 3.7. One can think of this as a table that is composed of entries that hold information on the names used in different code blocks of a given module.

Listing 3.7: 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 */
```

```
16              int recursion_depth;            /* current recursion depth */
17              int recursion_limit;            /* recursion limit */
18          };
```

A python module may contain multiple code blocks - such as multiple function definitions - and the `st_blocks` field is a mapping of all these existing code blocks to a symbol table entry. The `st_top` is a symbol table entry for the module being compiled (recall that a module is also a code block) so this will contain the names defined in the modules's global namespace. The `st_cur` refers to the symbol table entry for the code block that is currently being processed. Each code block within the module code block has it own symbol table entry that holds symbols defined within that code block.

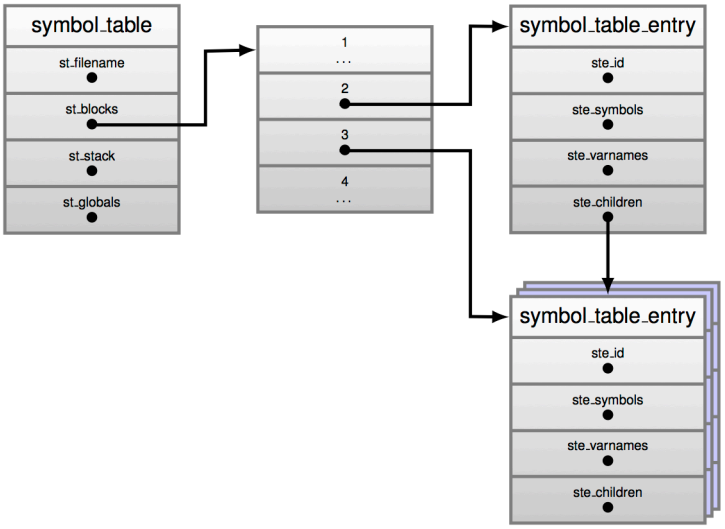


Figure 3.1: A Symbol table and symbol table entries.

Once again, going through the `_symtable_entry` data structure in `Include/symtable.h` is very helpful in getting an understanding for what this data structure does. This data structure is shown in listing 3.8.

Listing 3.8: The _symtable_entry data structure

```
1           typedef struct _symtable_entry {
2               PyObject_HEAD
3               PyObject *ste_id;               /* int: key in ste_table->st_blocks */
4               PyObject *ste_symbols;          /* dict: variable names to flags */
5               PyObject *ste_name;             /* string: name of current block */
6               PyObject *ste_varnames;         /* list of function parameters */
7               PyObject *ste_children;         /* list of child blocks */
8               PyObject *ste_directives;       /* locations of global and nonlocal
9                                                  statements */
10              _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 * */
        int ste_opt_col_offset;  /* offset of last exec or import * */
        int ste_tmpname;         /* counter for listcomp temp vars */
        struct symtable *ste_table;
    } PySTEntryObject;
```

The comments in the source code explain what each field does. The `ste_symbols` field is a mapping that contains the symbols/names that are encountered during the analysis of the code block; the flags which the symbols map to are numeric values that provide information on the context in which symbol/name is being used. For example, a symbol may be a function argument or a global statement definition. Some examples of these flags as defined in the `Include/symtable.h` module are shown in listing 3.9.

Listing 3.9: Flags that specify context of a name definition

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

Returning to our discussion on symbol tables, assuming a module that contains the code shown in listing 3.10 is being compiled. After the symbol table is built, there are three symbol table entries.

Listing 3.10: A simple python 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 scope of local. 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 will have the count variable marked as `free`. One thing to note is that although `make_counter` is defined as local in the symbol table entry for the module block, it is regarded as globally defined within the module code block as the `*st_global` points to the the `*st_top` symbols which in this case is that of the enclosing module.

3.5 From AST To Code Objects

With the symbol table generated, the next step for the compiler is to generate code objects from the `AST` incorporating information contained in the symbol table; the functions that handle this step are implemented in the `Python/compile.c` module. This process of generating code objects is also a multi-step process. In the first step, the `AST` is converted into basic blocks of python byte code instructions. The algorithm for this is similar to that used in generating symbol tables - functions named `compiler_visit_xx` where `xx` is the node type are used to recursively visit each node type emitting basic blocks of python bytecode instructions during the visit process. The basic blocks and paths between them implicitly represent a graph - the control flow graph. This graph shows the code paths that can be taken during the execution of a program. In the second step, the generated control flow graph is flattened using a post-order depth first search transversal. After the graph is flattened, the jump offsets are then calculated and used as instruction argument for byte code `jump` instructions. The code object is emitted from the this set of instructions. To get a good understanding of this process, consider the the `fizzbuzz` function in listing 3.11.

Listing 3.11: A simple python function

```
1    def fizzbuzz(n):
2        if n % 3 == 0 and n % 5 == 0:
3            return 'FizzBuzz'
4        elif n % 3 == 0:
5            return 'Fizz'
6        elif n % 5 == 0:
7            return 'Buzz'
8        else:
9            return str(n)
```

The AST for the function for this function is shown in figure 3.2.

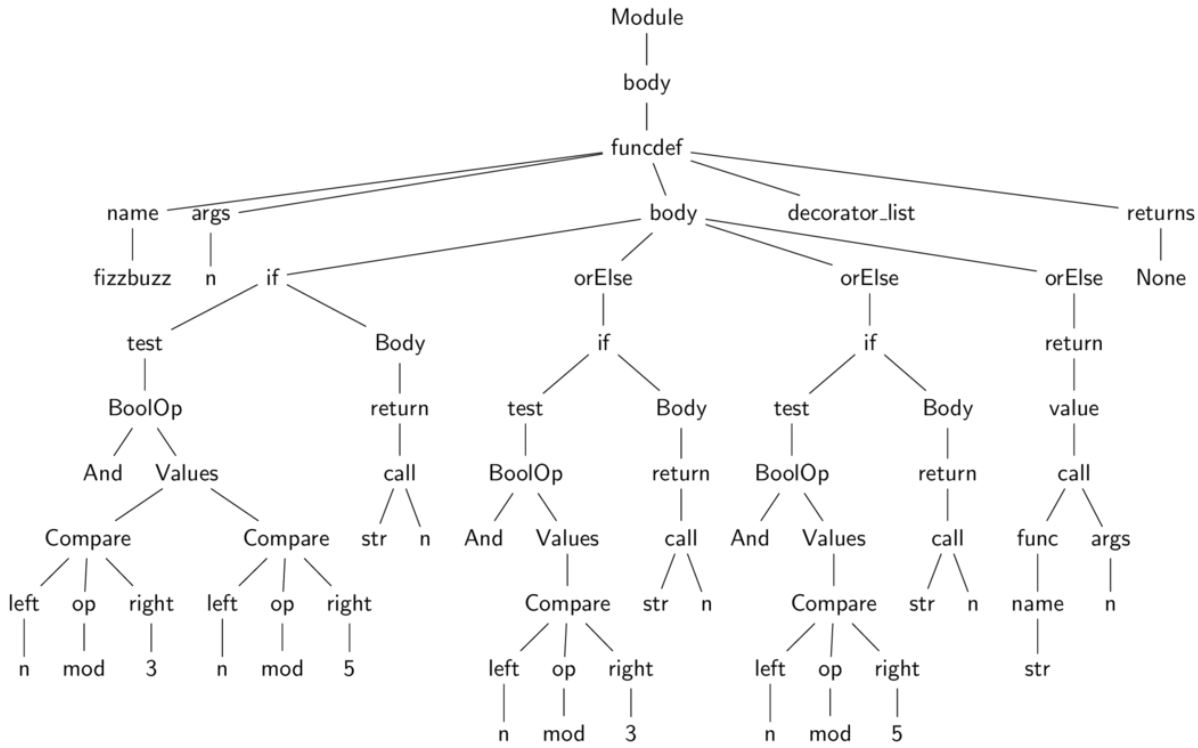


Figure 3.2: A very simple AST for listing 3.2

This AST from figure 3.2 when compiled into a CFG gives a graph that is similar to that shown in figure 3.3; empty blocks have been omitted from the figure. An inspection of this graph provides some intuition behind the basic blocks. The basic blocks have a single entry point but can have multiple exits. These blocks are described in more detail next.

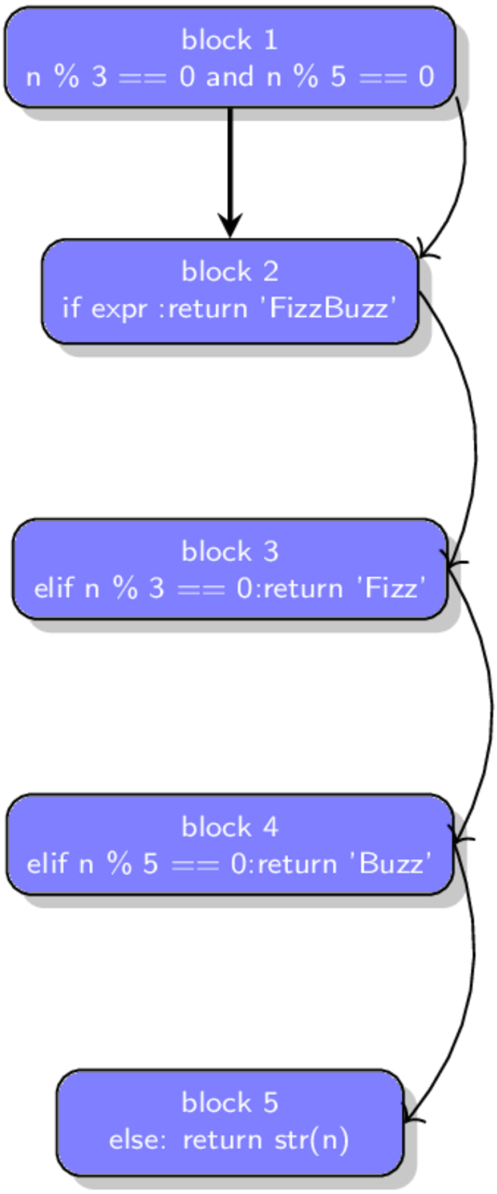


Figure 3.3: Control flow graph for the fizzbuzz function from listing 3.11. The straight line represent normal straight line execution of code while the curved lines represent jumps.

In the following descriptions, only the actual instructions have been included; some instructions need an argument but these have not been included to enable us focus on the main topic at hand.

1. Block 1 - This block contains instructions that map to the `BoolOp` node of the AST in figure 3.2. The instructions in this block implement the operation `n%3==0 and n%5==0` using the following set of eleven instructions.

```
LOAD_FAST
LOAD_CONST
BINARY_MODULO
LOAD_CONST
COMPARE_OP
JUMP_IF_FALSE_OR_POP
LOAD_FAST
LOAD_CONST
BINARY_MODULO
LOAD_CONST
COMPARE_OP
```

Surprisingly, the rest of the `if` node (the actual test that determines if the clause should be executed) is not included within this block. The reason for this will become clearer as the second block is discussed. As the image in figure 3.3 shows, there are two ways to exit this block - either via a straight execution of all opcodes or a jump to block 2 when `JUMP_IF_FALSE_OR_POP` is executed.

2. Block 2 - This block maps to the first `if` node encapsulating the `if` test and subsequent clause . The second block contains the following four instructions.

```
POP_JUMP_IF_FALSE
LOAD_CONST
RETURN_VALUE
JUMP_FORWARD
```

As will be seen in subsequent chapters, when the interpreter is executing the byte code instructions for an `if` statement, it reads an object from the value stack and depending on the truth value of that object, it either executes the next bytecode instruction or jumps to an instruction in a different part of the set of instructions and continues execution from there. The `POP_JUMP_IF_FALSE` is the instruction that handles this; this opcode takes an argument that specifies the destination of such a jump. One might wonder why the instructions for `BoolOp` node and `if` statements are in different blocks. To understand this, recall that python uses short circuit evaluation for boolean operations so in this case, if `n%3==0` evaluates to false then `n%5==0` is not evaluated. Looking at the instructions from the first block, one can notice the `JUMP_IF_FALSE_OR_POP` instruction after the first comparison. This instruction is a variant of a `jump` and hence needs a target. Think about this for a second and the need for the different blocks becomes obvious. The `JUMP_IF_FALSE_OR_POP` needs a target at which to continue the execution of the instructions when the first expression in the boolean expression evaluates to false due to the short circuit operation - the target in this case is the `POP_JUMP_IF_FALSE` instruction for the `if` statement. For a jump to be possible, we need a target to jump to and with the `if` statement instructions in a different block, the offset by which a jump is made can then be calculated. If all the components of the boolean expression are evaluated then execution will continue as normal with instructions in the `if` block executed after all instructions in the `BoolOp` block are executed.

3. Block 3 - This maps to the first `orElse` AST node of the third block and it contains the following 9 instructions.

```
LOAD_FAST
LOAD_CONST
BINARY_MODULO
LOAD_CONST
COMPARE_OP
POP_JUMP_IF_FALSE
LOAD_CONST
RETURN_VALUE
JUMP_FORWARD
```

   Observe that the `elif` statement and the `n%3==0` as well as body of the statement are all in the same block and it is easy to see why this is so. The only entry into this block is through a jump into this block and the node can be exited either through the return instruction or a jump when the `if` test fails.
4. Block 4 is a mirror image of block 3 in terms of instructions but with the arguments to the instructions differing.
5. Block 5 maps to the final `orElse` AST node and contains the following 4 instructions.

```
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

   The `LOAD_GLOBAL` takes the `str` function as argument and loads it into the value stack. `LOAD_FAST` loads the argument `n` onto the stack while RETURN_VALUE returns the value that is on the stack from the execution of the `CALL_FUNCTION` instruction i.e `str(n)`.

Like in the previous section, we look at the data structures that are used in building the basic blocks to get a better grasp of the process.

The `compiler` data structure

Figure 3.4 shows the relationship between the main data structures used in the process of generating the basic blocks that make up the control flow graph.

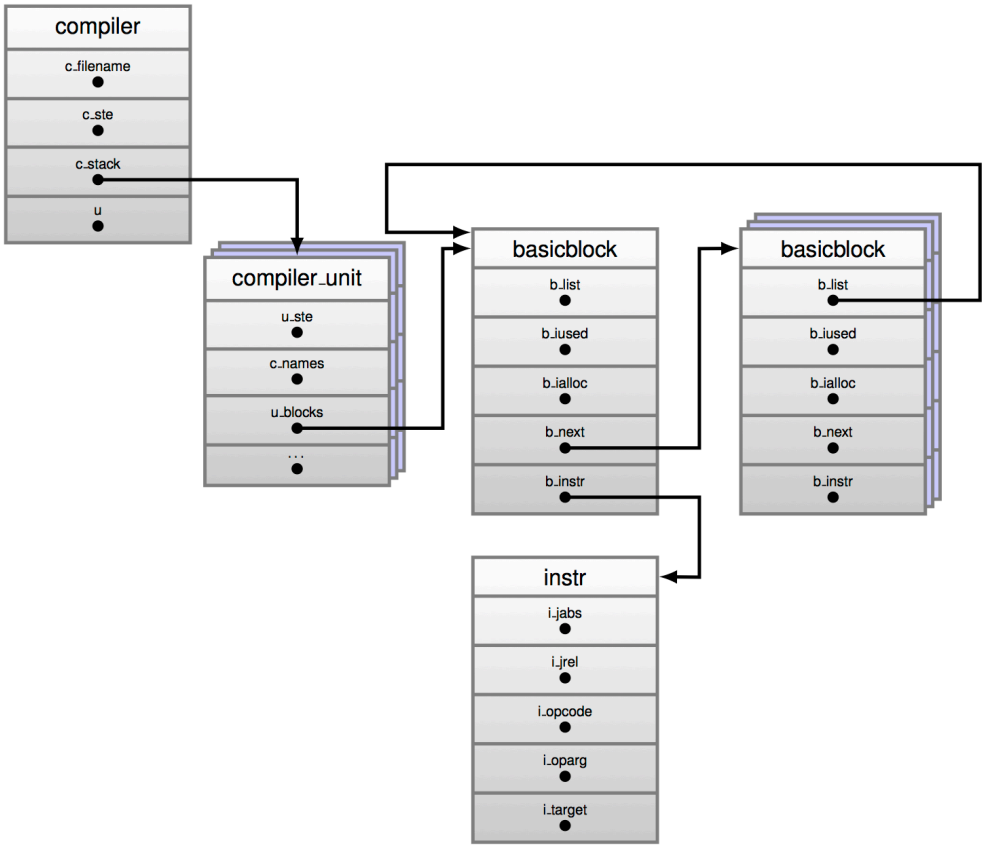


Figure 3.4: The four major data structures used in generating a code object.

At the very top level is the `compiler` data structure which captures the global compilation process for a module. The data structure is defined in listing 3.12.

Listing 3.12: 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 */
            PyObject *c_stack;           /* Python list holding compiler_unit
                                        ptrs */
            PyArena *c_arena;            /* pointer to memory allocation arena */
        };
```

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 operated on.
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 restoring of compile_unit data structures as new blocks are encountered. When a new code block is entered - a new scope is created - 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 block is exited, the `*c_stack` is popped off the stack accordingly to restore state.

For every module being compiled, a `compiler` data structure is initialised; as the AST generated for the module is walked, a `compiler_unit` data structure is generated for each code block that is encountered within the `AST`.

The `compiler_unit` data structure

The `compiler_unit` data structure as shown in listing 3.13 below captures information needed to generate the required byte code instructions for a code block. Most of the fields defined in the `compiler_unit` will be encountered when we look at code objects.

Listing 3.13: The `compiler_unit` data strcuture

```
        struct compiler_unit {
            PySTEntryObject *u_ste;

            PyObject *u_name;
            PyObject *u_qualname;  /* dot-separated qualified name (lazy) */
            int u_scope_type;

            /* The following fields are dicts that map objects to
            the index of them in co_XXX.      The index is used as
            the argument for opcodes that refer to those collections.
            */
            PyObject *u_consts;    /* all constants */
            PyObject *u_names;     /* all names */
            PyObject *u_varnames;  /* local variables */
            PyObject *u_cellvars;  /* cell variables */
```

```
16              PyObject *u_freevars;  /* free variables */
17
18              PyObject *u_private;        /* for private name mangling */
19
20              Py_ssize_t u_argcount;        /* number of arguments for block */
21              Py_ssize_t u_kwonlyargcount; /* number of keyword only arguments
22                                              for block */
23              /* Pointer to the most recently allocated block.  By following b_list
24              members, you can reach all early allocated blocks. */
25              basicblock *u_blocks;
26              basicblock *u_curblock; /* pointer to current block */
27
28              int u_nfblocks;
29              struct fblockinfo u_fblock[CO_MAXBLOCKS];
30
31              int u_firstlineno; /* the first lineno of the block */
32              int u_lineno;          /* the lineno for the current stmt */
33              int u_col_offset;      /* the offset of the current stmt */
34              int u_lineno_set;  /* boolean to indicate whether instr
35                                    has been generated with current lineno */
36          };
```

The `u_blocks` and `u_curblock` fields reference the basic blocks that make up the code block being compiled. The `*u_ste` field is a reference to a symbol table entry for the code block being compiled. The rest of the fields have pretty self explanatory names. The different nodes that make up the code block are walked during the compilation process 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 are 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. Boolen operations. etc.

The `basic_block` and `instruction` data structures

The basic block data structure is the rather interesting data structure in the process of generating a control flow graph. A basic block is a sequence of instructions that has one entry point but multiple exit points. The definition for the `basic_block` data structure as used in the python virtual machine is shown in listing 3.14.

Listing 3.14: 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;
```

As previously mentioned, the `CFG` is basically composed of basic blocks and connections between these basic blocks. The `*b_instr` field references an array of instruction data structures and each of these data structure holds a bytecode instruction. These bytecode can be found in `Include/opcode.h` header file. The instruction data structure is a shown in listing 3.15.

Listing 3.15: 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;
        };
```

Take a look at the CFG for the fizzbuzz function, we can see that there are actually two ways, execution can get from block 1 to block 2. The first is through normal execution - all the instructions

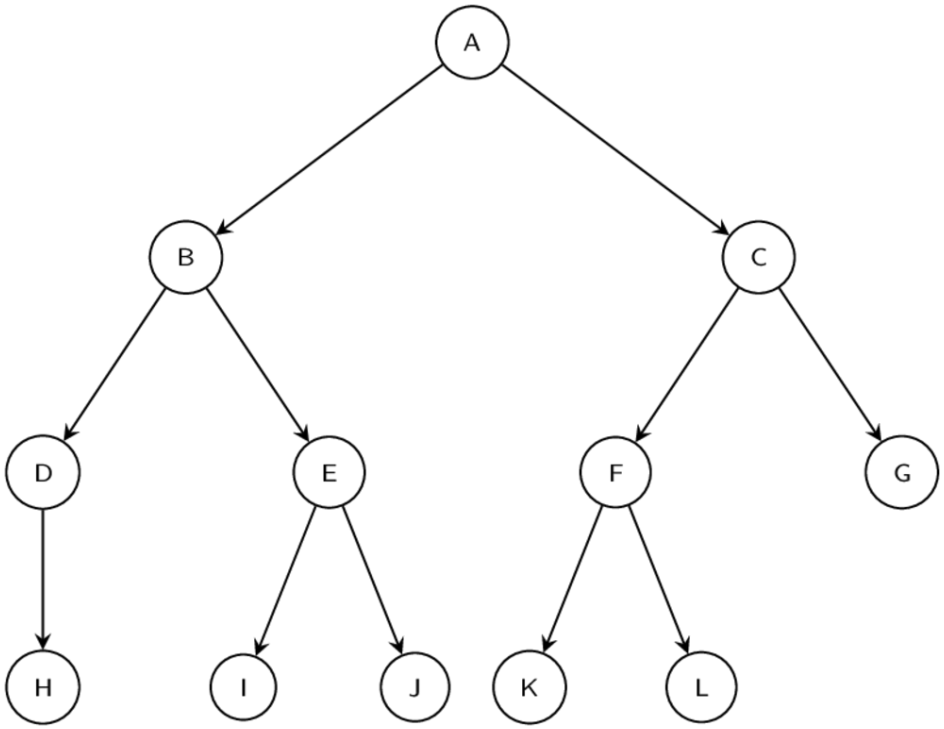
in block 1 are executed so the flow of execution continues in block 2. The other is by a way of the jump instruction that exist just after the first comparison operation. Now, the target of this jump is a basic block but the code objects that are actually executed know nothing of basic blocks – the code block only has a stream of bytecodes and we can only index such stream using offset. We have to take the implicit graph created with blocks as jump targets and somehow replace such blocks with offsets into an array of instructions. This is what the process of assembling the basic blocks does.

Assembling basic blocks

Once the CFG is generated, the basic blocks now contain bytecode instructions that represent the `AST` but the blocks are not ordered linearly and in the case of jump statements, the instructions still have basic blocks as jump targets rather than relative or absolute offsets into the instruction stream. The `assemble` function handles the linearization of the CFG and creation of code object from the CFG.

First, the assemble function adds instructions for a `return None` statement to any block that ends without a `RETURN` statement - now you know why you can define methods without adding a `RETURN` statement. This is followed by a *post-order* depth first traversal of the implicit `CFG` graph in order to flatten out the blocks - the post order traversal visits the children of a node before visiting the node itself.

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 3.5, 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 CFG of the `fizzbuzz` function given in listing 3.3 is a relatively simple graph - the result of the *post-order* traversal for the `fizzbuzz` is `block 5 -> block 4 -> block 3 -> block 2 -> block 1`. Once this graph has been linearized (i.e flattened), the offsets for instructions jumps can then be calculated by calling the `assemble_jump_offsets` function on the flattened graph.

Assembling the jump offsets takes place in two phases. In the first phase, the offset of every instruction into the instruction array is calculated as shown in the snippet from listing 3.16. This is a simple loop that works from the end of the flattened array building up the offset from `0`.

Listing 3.16: Calculating bytecode offsets

```
        ...
        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;
        }
        ...
```

In the second phase of the assembling of jump offsets, the jump targets for jump instructions is then calculated as shown in listing 3.17. This involves computing relative jumps for relative jumps and replacing the targets of absolute jumps with instruction offsets.

Listing 3.17: Assembling jump offsets

```
        ...
         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;
                    }
                }
            }
        }
        ...
```

With jump offsets calculated, the instructions contained in the flattened graph are emitted 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. The reason for this is obvious; we want a node that jumps to another node to always come before that jump target. With the emission of bytecode complete, the code objects can then be assembled 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 python objects and their implementation in the CPython virtual machine. A fundamental understanding of how the python objects are organized is important to groking the internals of the python virtual machine. Most of the source that is discussed here is available from the the `Include/` and `Objects/` directories. Unsurprisingly, the implementation of the object system in python is quite complex and 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 will show the ubiquity of the `PyObject` structure. Infact, as we will see later on in this treatise, when the interpreter loop is working on values on the evaluation stack, all of such values are regarded as `PyObject`*s*. For want of a better term, we refer to this as the *super class* of all python objects. Values are actually never declared as `PyObjects` 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 actually a `PyObject` structure.

A word on `C` structs.

When we say *values are never declared as `PyObject`s but a pointer to any object can be cast to a `PyObject`* we are referring to an implementation detail dependent on the C programming langugage and how it interprets data at memory locations. C structs which are used to represent python objects are just groups of bytes which we can interpret in any manner which choose to. For example, a struct, `test`, maybe 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 actually 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.

The `PyObject` structure is shown in listing 4.0 and is composed of a number of fields that must be filled in order 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 next object forming an implicit doubly linked list of all live objects. The `ob_refcnt` field is used for memory management and the `*ob_type` is a pointer to a type object that indicates the type of an object. It is this type that determines what the data represents, what kind of data it contains and the kind of operations that can be performed on that object. 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 *since 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 indeed points back at itself hence the saying that the type of a `type` is `type`.

A word on reference counting.

CPython uses reference counting for memory management. This is a simple method in which whenever a new reference to an object is created such as the case of binding a name to an object as in listing 4.1, the reference count of the object goes up. 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. When the reference count of an object gets to zero, it can be deallocated by the VM. In the VM world, the `Py_INCREF` and `Py_DECREF` are used to increase and decrease 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 collection of functions that are filled in by each type. We look at this data structure next.

4.2 Under the cover of 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. The `_typeobject` structure definition is reproduced in listing 4.2 for convenience.

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;
```

```
40          struct _typeobject *tp_base;
41          PyObject *tp_dict;
42          descrgetfunc tp_descr_get;
43          descrsetfunc tp_descr_set;
44          Py_ssize_t tp_dictoffset;
45          initproc tp_init;
46          allocfunc tp_alloc;
47          newfunc tp_new;
48          freefunc tp_free;
49          inquiry tp_is_gc;
50          PyObject *tp_bases;
51          PyObject *tp_mro;
52          PyObject *tp_cache;
53          PyObject *tp_subclasses;
54          PyObject *tp_weaklist;
55          destructor tp_del;
56
57          unsigned int tp_version_tag;
58          destructor tp_finalize;
59  } PyTypeObject;
```

The `PyObject_VAR_HEAD` field is an extension of the `PyObject` field that was discussed in the previous section; this extension adds an `ob_size` field for objects that have the notion of length. A comprehensive description of each of the fields in this type object structure is provided in the python C API documentation[1]. The important thing to note is that the fields in the strucutre each implement a part of the type's behaviour. Most of these fields are part of what we can call an object interface or protocol because they map to functions in that can be called on python objects but their actual implementation under the covers is type dependent. For example, `tp_hash` field is a reference to a hash function for a given type - this field can 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 is a reference 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 actually needs to run an initialization function especially when the type is immutable such as tuples while 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 `PyNumberMethods *tp_as_number` field. This field is a reference to a set of functions that implement number like operations and it means that the type will support arithmetic that have implementations

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

included in the `tp_as_number` set. For example, the non-numeric `set` type has an entry into this field because it supports arithmetic operations such as `-`, `<=` and so on.
2. Sequence protocol - A type that implements this protocol will have a value in the `PySequenceMethods *tp_as_sequence` field. This means that the type will support some or all of the 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 will enable instance of such type to be treated like python dictionaries using the dictionary subscripting 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.

As we progress through this chapter, we will look at various fields that make up a type object in more detail but for now we look at a number of different type objects as a concrete case study of how these fields are populated in actual type objects.

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. The implementation for the `tuple` type is shown in listing 4.3.

Listing 4.3: Tuple type definition

```
1    PyTypeObject PyTuple_Type = {
2        PyVarObject_HEAD_INIT(&PyType_Type, 0)
3        "tuple",
4        sizeof(PyTupleObject) - sizeof(PyObject *),
5        sizeof(PyObject *),
6        (destructor)tupledealloc,                   /* tp_dealloc */
7        0,                                          /* tp_print */
8        0,                                          /* tp_getattr */
9        0,                                          /* tp_setattr */
10       0,                                          /* tp_reserved */
11       (reprfunc)tuplerepr,                        /* tp_repr */
```

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

```
        0,                                      /* tp_as_number */
        &tuple_as_sequence,                     /* tp_as_sequence */
        &tuple_as_mapping,                      /* tp_as_mapping */
        (hashfunc)tuplehash,                    /* tp_hash */
        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 have been 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 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 key 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 that is 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 rest of the fields with `0` values have been left empty as they are not needed for the functionality of a tuple. 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 the function referenced by `tp_new` hence this field is left empty.

The `type` type

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

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

The `object` type

Another important type is the `object` type which 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 are used to fill in the

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

type fields of a user-defined type. This is due to the fact that user defined types behave in a different way 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 differs significantly from those provided by the `type` type.

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 ability to create an instance of a type. To fully understand the process of creating new type instances, it is important to remember that just as we differentiate between builtin types and user defined types [4], the internal structure of both will most likely differ too. 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 represent 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, except it is not inherited 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. To create a new tuple object, this function is dereferenced and invoked.

[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

```
1    static PyObject * tuple_new(PyTypeObject *type, PyObject *args,
2                                PyObject *kwds){
3        PyObject *arg = NULL;
4        static char *kwlist[] = {"sequence", 0};
5
6        if (type != &PyTuple_Type)
7            return tuple_subtype_new(type, args, kwds);
8        if (!PyArg_ParseTupleAndKeywords(args, kwds, "|O:tuple", kwlist, &arg))
9            return NULL;
10
11       if (arg == NULL)
12           return PyTuple_New(0);
13       else
14           return PySequence_Tuple(arg);
15   }
```

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 optimisations 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 basically allocates memory for an instance of the `PyTuple_Object` structure on the heap. This is where one difference between internal representation of builtin types and user defined types comes to the fore - instances of builtins types like tuple are actually `C` structs. This probably among other things is for efficiency. So what does this C struct backing a tuple object look like? It can be 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

```
1    typedef struct {
2        PyObject_VAR_HEAD
3        PyObject *ob_item[1];
4
5        /* ob_item contains space for 'ob_size' elements.
6        * Items must normally not be NULL, except during construction when
7        * the tuple is not yet visible outside the function that builds it.
8        */
9    } 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 instance 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 so - `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 derefenced whenever the *call* notation is used on an instance of `Type`. A snippet of the `type_call` 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;
```

Listing 4.7 shows that when an instance of the `Type` object is invoked, all that happens is that the `tp_new` field is dereferenced and whatever function that is referenced is invoked to get a new instance; if the `tp_init` exists, this is invoked on the new instance to carry out initialization of the new instance. This process provides an explanation for builtin types because afterall 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 in the case of `Test`, the `type_new` function checks for the presence of base types (super types/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 which is also 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 interpeter for a user defined type. In the case where the user defined type implements its own 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 then 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 actually 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.

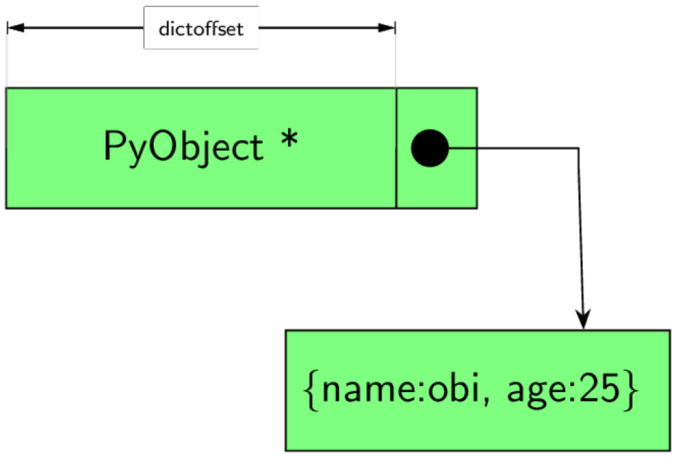


Figure 4.0: How instances of user defined types are structured.

For example, if an instance *PyObject* is at `0x10` and the offset is `16` then the instance dictionary that contains instance attributes can be found at `0x10 + 16`. This is hardly 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 - this is not the full story in the case of instances when `__slots__` are defined . The `dict` data structure can be found in one of two places depending on the type of the object as was mentioned in the previous section.

1. For objects that have 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 that have a type other than type *(i.e instances of 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.

Doing a simple diction access to obtain attributes seems straightforward but this is hardly the end of the story. 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 that is at the heart of attribute referencing in python.

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

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 while objects implementing the `__get__` and `__set__` are data descriptors meaning that such descriptor objects are writeable. We are interested in descriptors and their application in representing object attributes. 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 attribute of a class which it is used to represent. It is important to note that descriptors are effective 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.

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

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 level then any assignment to the attribute will overwrite the descriptor. One has to go through the python vm source code to get an appreciation for how integral descriptors are to Python. Descriptors provide the mechanism behind properties, static methods, class methods and a host of other functionality in Python. To give a concrete illustration of the importance of descriptors, consider the algorithm for resolving an attribute from an instance, `b`, of a user defined type as shown in listing 4.12.

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

```
    1. type(b).__dict__ is searched for the attribute name. If the name is
    found and it is a data descriptor, the result of calling the descriptor's
    __get__ method is returned. If the name is not found, then all base classes
    in the *mro* of type(b) are searched in the same way.
    2. b.__dict__ is searched and if attribute name is found here, it is
    returned.
    3. if the name from 1 is a non-data descriptor the value of call
    __get__ is returned,
    4. If the name is not found, an AttributeError is raised or __getattr__()
    is called if provided by the user defined type.
```

The algorithm in listing 4.12 shows that during attribute referencing we first check for descriptor objects; it also illustrates how the `TypedAttribute` descriptor is able to represent an attribute of an

object - whenever an attribute is referenced such as `b.name` the `Account` type object is searched for the attribute and in this case, a `TypedAttribute` descriptor is found and its `__get__` method is called accordingly. The `TypedAttribute` example illustrates a descriptor but is rather 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 algorith in listing 4.12 differs from the algorithm that is used when referencing an attribute whose type is `type`. The algorithm for such is shown in listing 4.13

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

```
1. type(type).__dict__ is searched for the attribute name. If the name
is found and it is a data descriptor, the result of calling the
descriptor's __get__ method is returned. If the name is not found, then
all base classes in the *mro* of type(type) are searched in the same way.
2. type.__dict__ and all its bases are searched for the attribute name.
If the name is found and it is a descriptor then return the value from
invoking its `__get__` method; if it is an ordinary attribute then return
this.
3. If a value was found in (1) and it is non-data descriptor then return
the value from invoking its __get__ function.
4. if the value found in (1) is not a descriptor then return the value.
```

Examples of Attribute Referencing with Descriptors inside the VM

Descriptors play a very major role in attribute referencing in Python. Consider the `type` data structure discussed earlier on 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 that wishes to be treated as a descriptor. A function object is a very good 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. The definition of a function object type is shown in listing 4.15.

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 when an instance is passed in to the call, then a new method object is created using the function and the instance. This sums up how python is able to return a different type for the same function reference using a descriptor.

When a method is defined in a class, we use the `self` argument as the first parameter to any instance method because in reality, instance methods take the instance (called `self` by convention) as the first argument. A call such as `b.name_balance_str()` is actually the samething as `type(b).name_balance_str(b)`. The reason why we are able to invoke `b.name_balance_str()` is because 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 builtin `type` and an instance of a user defined type.

Listing 4.17: Accesing the __dict__ attribute from an instance of the builtin type and an instance of a user defined type

```
        class A:
            pass


        >>> A.__dict__
        mappingproxy({'__module__': '__main__', '__doc__': None, '__weakref__': <att\
ribute '__weakref__' of 'A' objects>, '__dict__': <attribute '__dict__' of 'A' objec\
ts>})
        >>> 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 a mapping proxy that we cannot even assign to while 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 object 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 hence the reason why we can get different object types. We would like to find out what happens under the covers for this descriptors just as we did in the case of *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) which are shown in listing 4.19. This is the collection of values for which descriptor objects will be insert into the `type's` type `__dict__` attribute - this is the mapping that the `tp_dict` slot of the type object points to.

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 for which descriptors are insert into 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 `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 lets see what the *getter* function of this descriptors 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 4.19 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 the `__dict__` attribute of a type object is queried.

So what about the instance of A, a user defined type, how is the `__dict__` attribute resolved? Now recall that `A` is actually an object of type `type` so we go hunting in the `Object/typeobject.c` module to see how new type instance 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. The `PyObject_GenericGetDict` and an associated helper that actually fetches the instance dict are shown in listing 4.24. 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 the instance `dict`. In a situation where this function returns a null value, `PyObject_GenericGetDict` can go ahead to 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 descriptors are used to implement custom attribute access depending on types. The same strategy described above is used throughout the VM for other instances in which attribute access is performed using descriptors. Descriptors are pervasive within 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 but without discussing much about it so in this section we go into a bit more detail on *mro*. In python, types can belong to a multiple inheritance hierarchy so there is need for an order in which methods are searched 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. Some notations used in explaining this algorithm are shown in listing 4.25.

```
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 class are 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/

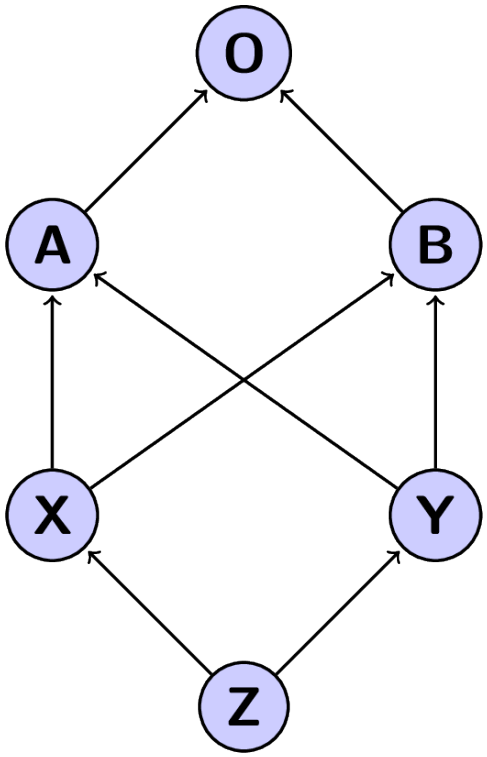


Figure 4.1: A simple multiple inheritance hierarchy.

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 skip 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`.

Using the same 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 now 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

In this part of the write-up, we explore code objects. Code objects are central to the operation of the python virtual machine. Code objects encapsulate the bytecode of the python virtual machine; we may call the bytecode the assembly language for the python virtual machine.

Code objects as the name suggests represent compiled executable python code. We have seen code objects before when we discussed the compilation of python source. Code objects are produced whenever a block of python code is compiled. 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 when run alter the state of the python virtual machine. Given a function, we can access the code object for the function body using the `__code__` attribute of the function 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 come with a number of fields that are used by the interpreter loop when executing and we look at some of these fields in the following sections.

5.1 Exploring code objects

A good place to start with code objects is be to compile a simple function and inspect the code object generated by that function. We use the simple `fizzbuzz` function as a guinea pig as shown in listing 5.2.

[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 printed out seem almost self-explanatory except for the `co_lnotab` and `co_code` fields that seem to contain gibberish. We go ahead and explain each of these fields and their importance o the python virtual machine.

1. *co_argcount*: This is the number of arguments to a code block. This has a value only for function code blocks. The value is set during the compilation process to the length of the argument set of the code blocks's `AST`. 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 that is executed by the evaluation loop. Each of these bytecode instruction sequence is composed of an `opcode` and an `oparg` - argument to the opcode where it exists. For example, `co.co_code[0]` returns the first byte of 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 contained 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 constant list. For example 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 argument to the opcode is a `0` and we can see that the `None` value has an index of `0` in the constants list from where it is actually loaded by the `LOAD_CONST` instruction.

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

The result of introspecting on the code object for the function shown in listing 5.4 is shown in listing 5.5.

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 obvious. `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 if the name `p` is not found, an error is thrown. The bytecode instructions for the function in listing 5.4 is shown in listing 5.6 and it is obvious how this works.

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 that the code object makes use of. 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 a stack based machine i.e values used in evaluation and results of 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 which cell storage objects have to be created 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 created for storing 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 life time may exceed that of the enclosing code object so such values have to be stored in other locations that do not get deallocated when the execution of the code object is completed.

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 an contained in an array, then this value is the index into that array at which the given instruction can be found. The third column is the actual human readable instruction opcode; the full range of opcodes can be 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 the 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 a stack based machine 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.

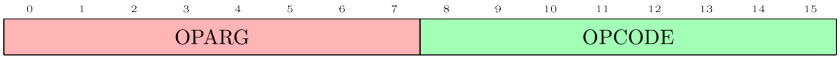


Figure 5.0: Bytecode instruction format showing opcode and oparg

Sometimes, the argument to an opcode maybe unable to fit into the default single byte. For these kind of arguments, the python virtual machine makes use of the `EXTENDED_ARG` opcode. 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 argument to the `LOAD_CONST` opcode, this value cannot fit into a single byte so the `EXTENDED_ARG` opcode is used. The binary represenation 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 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 being compiled. Assuming we were 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)
```

When the module code block is compiled we get the output shown in listing 5.10.

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 that is stored as the name `f` - our function definition using the `MAKE_FUNCTION` Instruction. The content of this code object is shown in listing 5.11

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 actually created during the compilation of the module. What we have are just code objects - the functions are actually created during the execution of the code objects as seen in listing 5.10. Inspecting the attributes of the code object will actually 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 being created only during the execution of the code object.

5.3 Code Objects in the VM

The implementation of code objects within the VM is very similar in terms of object attributes to that of its implementation in the python space. Like most builtin 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. Instances of code objects are represented by the structure shown in listin 5.13.

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 exist for optimisation purposes. This holds a reference to a frame object that was previously used as a context to execute the code object. This is used as the execution frame when such code object is being re-executed in order to prevent the overhead of allocating memory for another frame object.

6. Frames Objects

Code objects contain the executable byte code but lack the contextual information required for the execution of such code. Take the set of bytecode instructions in listing 6.0 for example, `LOAD_COST` takes an index as argument but the code object has no array or data structure that contains data to load the value at the index from.

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
```

Another data structure that provides such contextual information is required for the execution of code objects and this is where frame objects come in. 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 that are required during the execution of some code object. As usual, python does indeed 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 has to be created. Such a frame object contains all the namespaces required for 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 prior to 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 all the way 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 have been 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 evaluation of bytecode, values are read from the top of a stack and results from the evaluation of byte code are stored on the top of a 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 to.
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 is used for tracing the execution of python code.
9. `f_exc_type`, `f_exc_value`, `f_exc_traceback`, `f_gen`: are fields that are used for book keeping in order to be able to cleanly execute generator code. 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 provides a mechanism for 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 provides faster name access than their counterpart `LOAD_NAME` and `STORE_NAME` opcodes because they use array indexing for accessing value of names and this is done in approximately constant time unlike their counterparts that search a mapping for the value associated with 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 which is 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 complex and we look at that in the coming chapters.

6.1 Allocating Frame Objects

Frame objects are ubiquitous during python code evaluation - every code block that is executed needs a frame object that provides some context. New frame objects are created by a 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 remain 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 actually 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 tiny 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. There are still a few things that have been left out of this discussion but which we cover 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 will be answered when we look at the very important interpreter and thread state data structures in the next chapter and then the evaluation loop in subsequent chapters.

7. Interpreter and Thread States

As disucussed in opening chapters, one of the steps during the bootstrapping of the python interpreter is the initialisation of the interpreter state and thread state data structures. In this chapter, we look at these data structures in detail and explain the importance of these data structures.

7.1 The Interpreter state

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

The interpreter state is a very simple data structure that captures the global state that is shared by a set of cooperating threads of execution in a python process. A cross section of this data structure definition is provided in listing 7.0 to provide some insight into this very important data structure.

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 shown in listing 7.0 maybe familiar to anyone that has covered all the material up to this point 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. The memory and *Global Interpreter Lock* available to the process is however shared by all interpreter threads within that process.

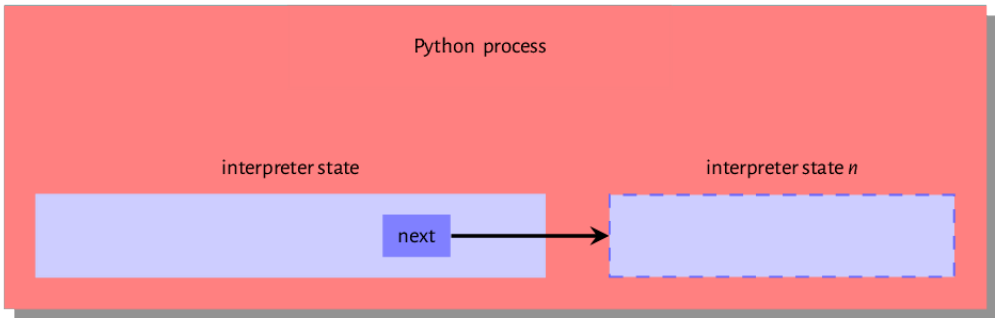


Figure 7.0: Interpreter state within the executing python process

- `*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 executin 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 is the set of builtin 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 values in these fields are used to locate such codecs as well as handle errors that maybe 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 it is able to do this by maintaining a thread state data structure for each thread of execution. We look at this data structure next.

7.2 The Thread state

Jumping straight into the exploring the *Thread state* data structure which is shown 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;
```

The `next` and `previous` fields of a thread state data structure reference threads states created prior to 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 interpreter state that the thread state belongs to. The `frame` references the current frame of execution; as the code object that is executed changes, the value referenced by this field also changes.

The `recursion_depth` as the name suggest 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 to enable some clean-up operations. The `recursion_critical` flag is used to signal 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 important 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 assocated with a native OS thread within the running python process. Figure 7.1 is a good graphical illustration of this relationship. We can clearly 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 operation system thread of execution.

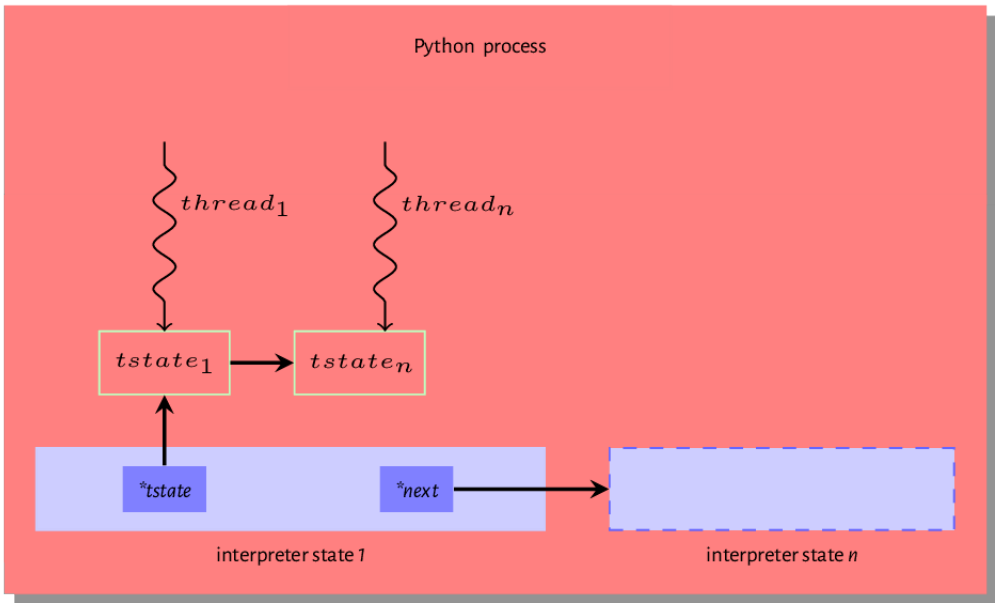


Figure 7.1: Relationship between interpreter state and thread states

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 in order 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 actually 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? There are a myriad of reasons why the GIL is relevant. 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 implemenation 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 slowed down single threaded programs atimes. A global lock also provides much needed synchronization when performing certain 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 serious 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.

At the interpreter startup, 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. When another thread is spawned using the python threading module, the GIL comes into play. The snippet in listing 7.3 is from the Modules/_threadmodule.c and provides an idea of how this process proceeds as a new thread is created.

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 `_PyThreadState_Prealloc` function call creates a new thread state for the thread which has not yet been created. Before the thread is actually created, the main thread of execution must acquire the `GIL`; the 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 actually create the new operating system thread. When a new thread is being spawned, a function that the thread should call when it comes alive is passed to the thread. In this case, that function is the `_tbootstrap` function in the `Modules/_threadmodule.c` module. A snapshot of this bootstrap function is shown 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 actual 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 was able to take the GIL. This is meant to prohibit 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 from listing 7.4 invokes the `PyEval_AcquireThread` function that handles *requesting* for the `GIL`. A lay explanation for what happens when this request is made is thus assuming `A` is the main thread of execution holding the `GIL` while `B` is the new thread being 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 executing on the evaluation loop checks if the `gil_drop_request` variable is set on each iteration of the loop.
4. Thread A drops the `GIL` when it detects that the `gil_drop_request` variable is set and then 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 process core setting in in python most times does not speed up such a program. Infact 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 we have created so far of the python virtual machine. When the python executable is invoked with a file containing some valid source code content, first

the interpreter and thread states are initialised and this is followed by the compilation of the source file into a code object. The code object is then passed to the interpreter loop module where in order to execute the code object, a frame object is created and attached to the main thread of execution. 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 has references a frame that may reference a frame and so on forming a stack of frames. Figure 7.2 provides a graphical representation of this order.

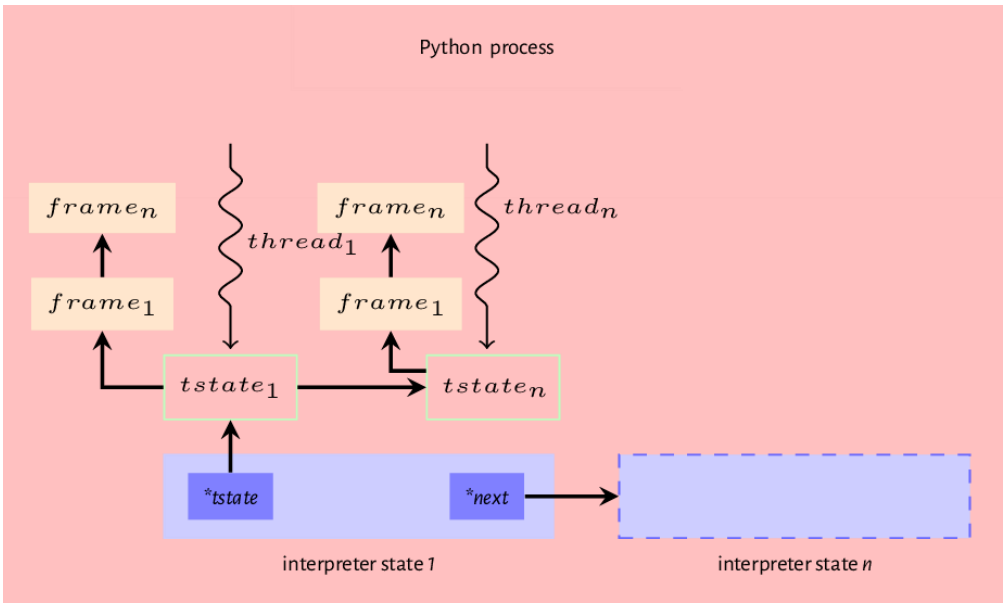


Figure 7.2: Interpreter state, thread state and frame relationship

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 multiple times that the python virtual machine generically treat values for evaluation as `PyObjects`. This leaves 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 argument to an addition operation but how does the virtual machine know if the values actually implement a protocol of which the add operation is part of ?

To understand how alot of the operations on `PyObjects` work, we only have to look at the `Objects/Abstract.c` module. This module defines a number of functions that work on objects that implement a given object protocol. This means that for example, if one was 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 the case of the `BINARY_ADD` opcode, when it is applied to the addition of two numbers, the `PyNumber_Add` function of the `Objects/Abstract.c` module is invoked. The definition of this function is provided in listing 8.1.

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 from listing 8.1 - the call to the `binary_op1` function. The `binary_op1` function is another generic function that takes among its parameters, two values that are numbers or subclass of numbers and applies a binary function to those two values; 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 included 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 take 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` that are 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 a check for if both values are of the same type as there is no need to dereference the second value's function in the op_slot if both values are of the same type. Even if both values are are not of the same type but the functions that were 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 and if that is true, the slotw 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 virtual machine is able to perform operations on values that are supplied to it. We have simplified things a bit here by ignoring opcodes that can be overloaded - for example the + symbol maps to the BINARY_ADD opcode and can be applied to a string, a number or a sequence but in our example above we have only looked at it being applied to numbers and subclasses of numbers. It is not too difficult to imagine how overloaded operations are handled. In the case of the BINARY_ADD if one looks at the PyNumber_Add function, one can see that if the value returned from the binary_op1 call is Py_NotImplemented then the virtual machine will attempt to treat the values as sequences and try to dereference the sequence concatenation method that is then applied to the both values if they implement the sequence protocol. Taking a step back to the interpeter loop in ceval.c, when we observe the case for the evaluation of the BINARY_ADD opcode, we see the following snippet.

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 in order 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 most opcode operations are handled - check the type of the values being evaluated 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 - it is here that the virtual machine iterates over python bytecode instructions of a code object and executes such instructions. This is achieved using an actual `for` loop that iterates over an opcode switching on each type in order to run the desired execution 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 so as 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 execution of bytecode can take place, a number of housekeeping operations such as creating and initializing frames, setting up variables and initializing the virtual machine variables such as instruction pointers have to be carried out. We get our feet wet with these operations first explaining the setup processes that have to take place before the evaluation begins.

9.1 Putting names in place

As mentioned above, the heart of the virtual machine is the `PyEval_EvalFrameEx` function that actually executes the python bytecode but before this bytecode can be executed, a lot of setup - error checking, frame creation and initialization etc - has to take place in order to prepare the evaluation context. This is where the `_PyEval_EvalCodeWithName` function also within the `Python/ceval.c` 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 objects are created for code blocks; these code blocks could either be functions, modules etc. so for a module with the above content, we can safely make the assumption 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 is lies with the `_PyEval_EvalCodeWithName` function that has a signature shown in listing 9.1. It is this function that handles the name setup that is required 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 fully known only at runtime; this basically means that we cannot populate this fastlocals data structure without careful error checking. It is therefore 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 supplied 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
module).
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.

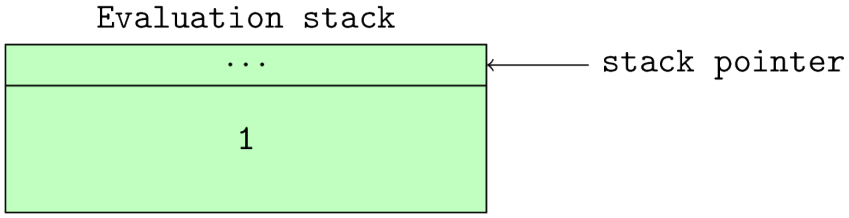


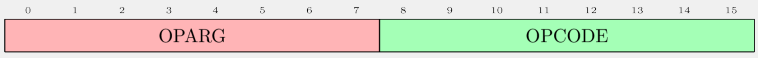
Figure 9.0: Stack pointer after a single value has been pushed onto the stack

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 currently 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 occured. 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: references the names of all values in the code block of the executing frame.
10. consts: 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 format of bytecode instructions 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 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 very important 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 macros 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 peforms 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 checks 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 other wise 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 a few examples are shown in the following snippet.

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 opcode are actually evaluated. The actual loop implementation is pretty anti-climatic as there is really nothing special here, just a *never ending* for loop and a massive switch statement that is used to match 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")
```

A disassembly of the function from listing 9.7 is shown in listing 9.6 and we show how this set of bytecode loops through the evaluation switch.

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
```

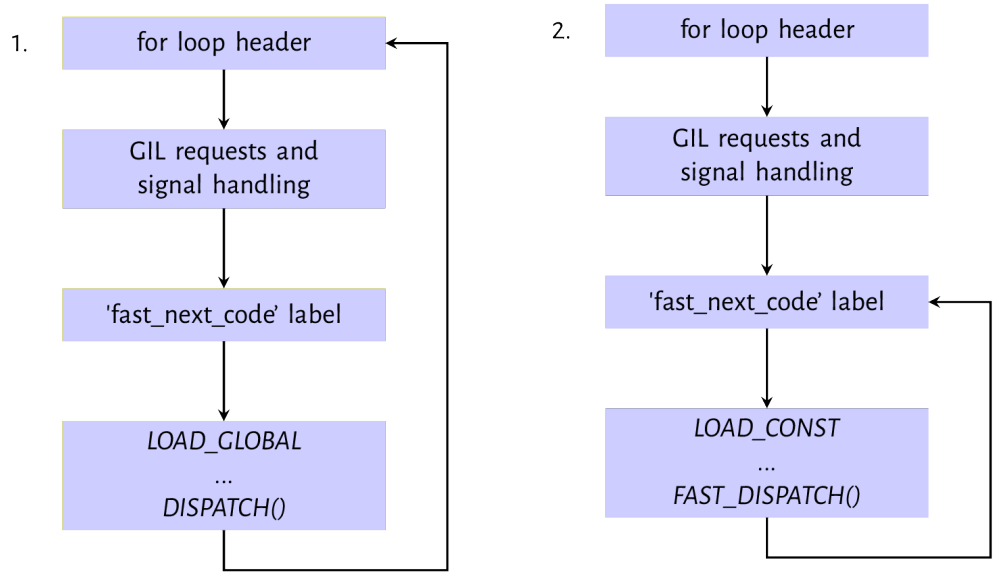


Figure 9.1: Evaluation path for `LOAD_GLOBAL` and `LOAD_CONST` instructions

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 housekeeping tasks that are carried out on every iteration of the evaluation loop. The`GIL` and signal handling checks were discussed in 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 exist 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 is 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

associatd with the name from the f_globals or f_builtins object with the assumption that they implement some protocol for fetching value associated with a given name. If this value is found, it is loaded on to the evaluation stack using the *PUSH(v)* otherwise an error is set and there is a jump to the error code label for error handling. As the flow chart shows, after this value is pushed onto the evaluation stack, the *DISPATCH()* macro is called which is basically an alias for the continue statement.

The second diagram labelled 2 in the figure 9.1 shows the execution of the LOAD_CONST. Listing 9.9 is an the 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 normal 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 implemenations that make C function calls make of the DISPATCH macro while opcodes like the LOAD_GLOBAL that do not make C function calls in their implementation make use of the FAST_DISPATCH macro. This means that the GIL can only be given up after executing opcodes that make C function calls.

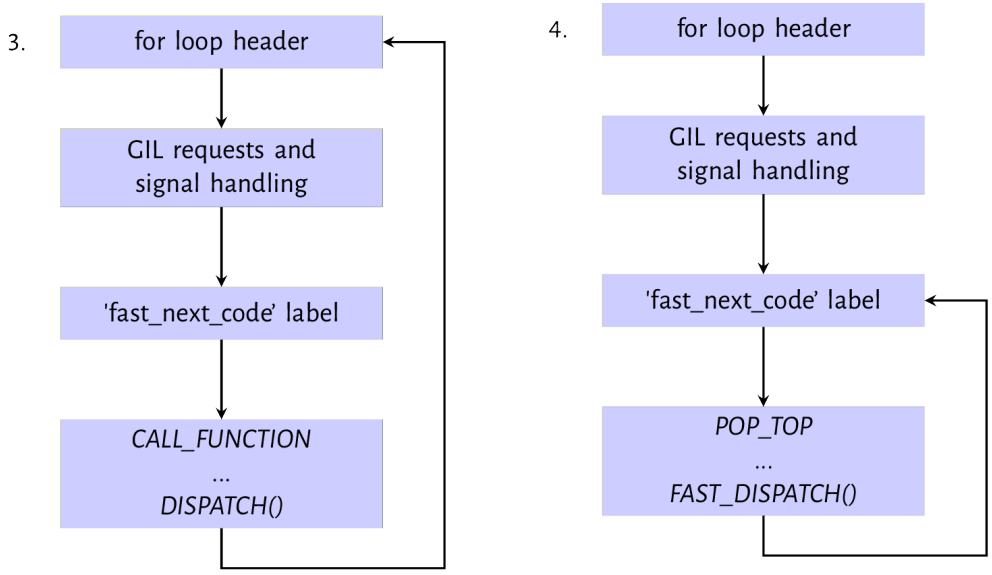


Figure 9.2: Evaluation path for CALL_FUNCTION and POP_TOP instruction

The next opcode that is executed is the CALL_FUNCTION opcode as shown in first image from figure 9.2. This opcode is emitted by the compiler when a function call is made with only positional arguments used in the call. The implemenation of this opcode is shown in listing 9.10. 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.

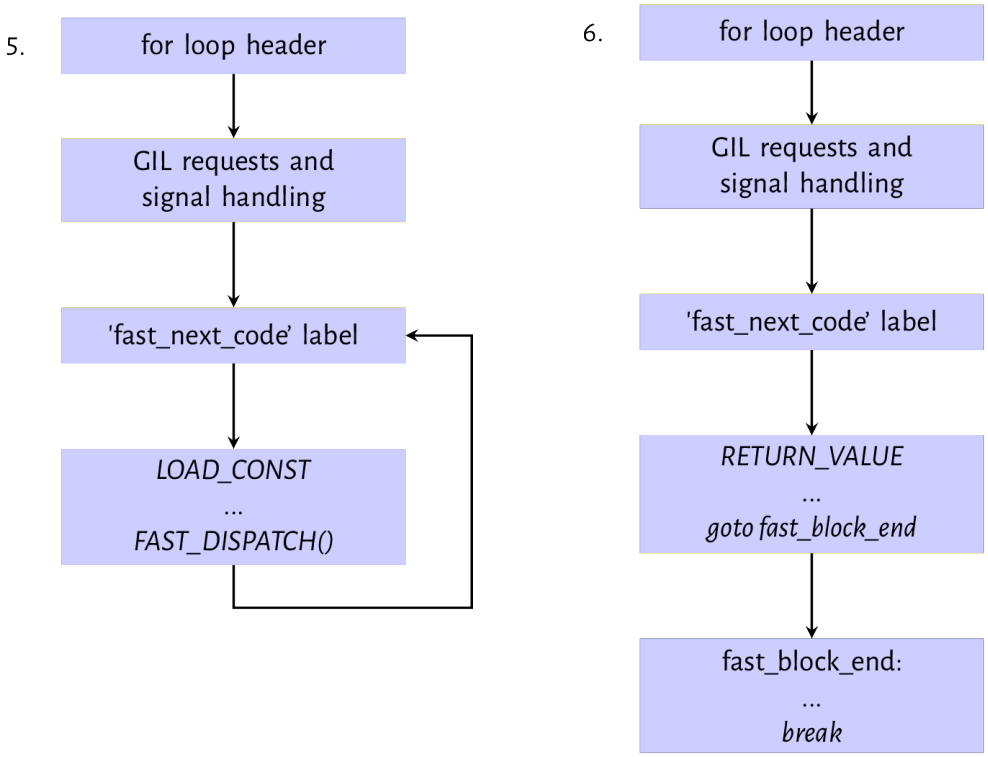


Figure 9.3: Evaluation path for LOAD_CONST and RETURN_VALUE instruction

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 for the RETURN_VALUE

to work with. These tow always go together when a python 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, set the WHY status code to WHY_RETURN and then performs a jump to the fast_block_end code label. The execution continues from there breaking out of the for loop and then subsequently returning the value of the retval variable to the calling function.

Notice that a lot of the code snippets thatwe 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 looked at in this section is rather trivial, it encapsulates the main behaviour of the evaluation loop while executing bytecode instructions. Any other opcode may have an implementation that is a bit more complicated 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 suggest the opcode creates a function object from values on the 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!")
```

A disassembly of the code object formt he compilation of the module 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();
}
```

The PyObject_GetAttr function is one we have looked at in the chapter on objects. This function de-references whatever value is in the tp_getattro attribute of the object and uses that to get load the value of the attribute on to the top of the value stack. One can review the chapter on objects to get a better understanding of how this 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. The implementation for this opcode is shown in listin 9.15. Notice how one of the major 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 concept behind the evaluation loop are not terribly complicated - opcodes each have implementations that are defined in C and

these implemenations are the actual do work functions. One very important area that we have not touched upon is exception handling and the block stack, two intimately connected 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 just mention the block stack passingly but then focus on the evaluation stack. However, the block stack is pretty important - there are probably other ways to implement exception handling but as we will see while 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 taking exception handling into consideration. The block stack is also used for loops 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 so lets get straight down to the details. 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 of the frame, 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")
```

When the function from listing 10.0 is disassembled, the result is shown in listing 10.1.

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
```

For a simple function, listing 10.1 is a lot of opcodes but this is due to the combination of a `for` loop and `try .. finally` constructs. 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 push 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. The implementation of this function is shown in listing 10.2.

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. It is best to illustrate the example from above with a graphical representation of the execution process and figure 10.0 illustrates this with a portion of the bytecode.

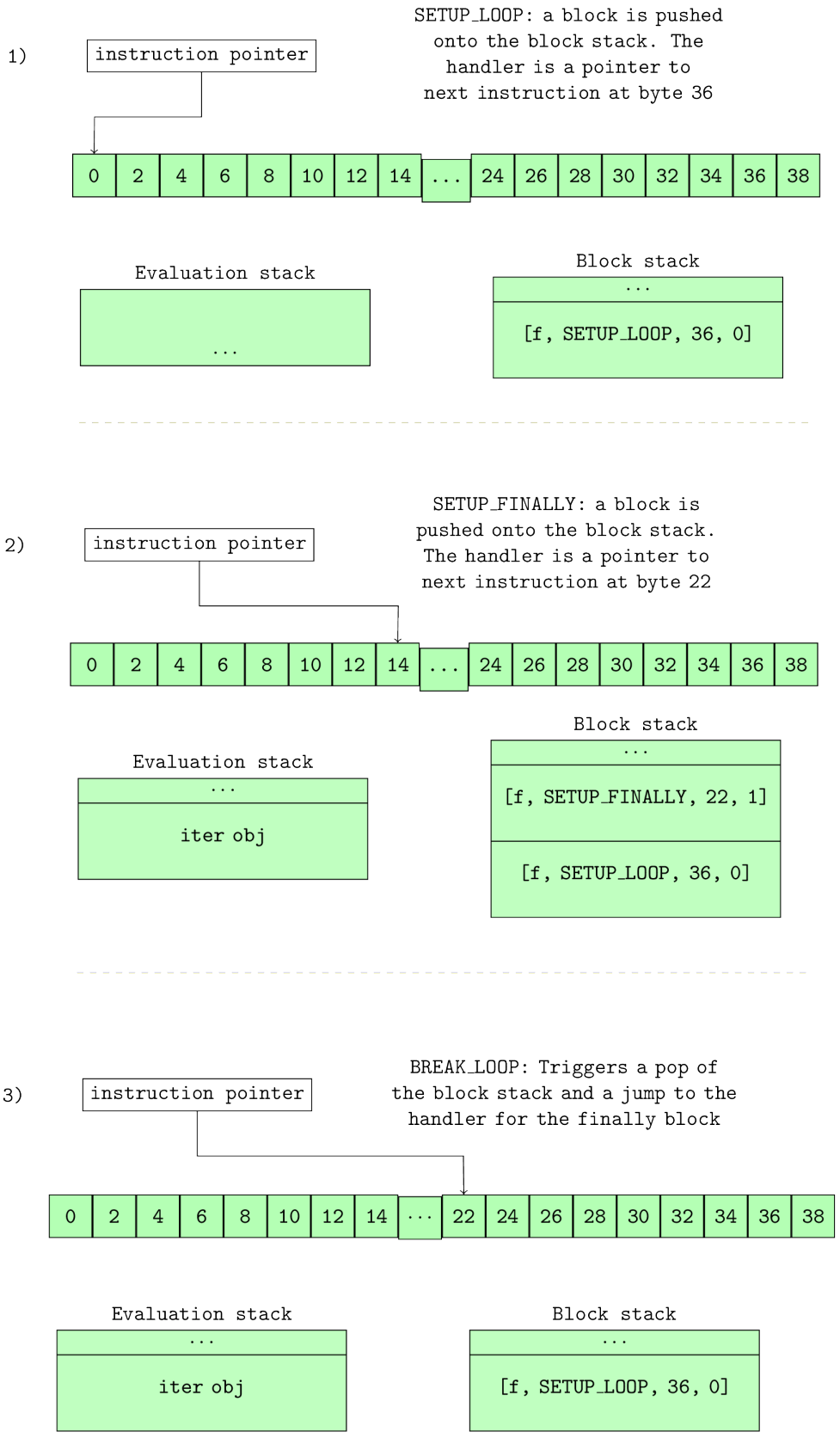


Figure 10.0: How the block stack changes with SETUP_* instructions

Figure 10.0 shows how the block stack varies as each instruction is executed.
In the first diagram of the figure 10.0, the `SETUP_LOOP` opcode is executed and a single `SETUP_LOOP` block is placed on the block stack. The handler for this block is the instruction at offset 36 so when this stack is popped under normal execution, the interpreter will jump to that offset and continue execution from there. When the `SETUP_FINALLY` opcode is encountered, another block is pushed onto the block stack. 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.

The real place where the use of a block stack shines is when the break statement is encountered while inside the exception handler within the loop. When the `BREAK_LOOP` opcode is executed, the `why` variable is set to `WHY_BREAK` and a jump is executed to the `fast_block_end` code label as shown in the second diagram of figure 10.0 where block stack unwinding is handled. 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 of the stack and the interpreter jumps to its handler at bytecode offset 22. Normal execution continues from that offset till the `END_FINALLY` statement is encountered. 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 and continuing with normal execution.

The use of the block stack greatly simplifies the implementation of the virtual machine implementation. If the loops were not implemented with a block stack, an opcode such as the `BREAK_LOOP` will 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 how exceptions and exception handling are implemented. 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")
```

The opcodes generated for the simple function in listing 10.3 are shown in listion 10.4.

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. Buried in the detail of that is the fact that in addition to returning a `NULL` value, the functions there also set exception values on the thread state data structure of the currently executing thread. 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_Restore` function within the same module where the values are set on the currently executing thread.

With the exception values set on the currently executing thread 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. Once the jump to error

handler label occurs, the stack unwinding can begin. The exception values - traceback, exception value and exception type are pushed on top of the value stack, the `SETUP_EXCEPT` handler is popped from the block stack and then there is a jump to the handler - byte offset 14 in this case from where execution continues. Now observe the bytecodes from bytecode offset 16 to bytecode offset 20 in listing 10.4 - the `Exception` class is loaded onto the stack and then this is compared with the exception that was raised and is 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 put anything on the block stack so when an exception happens and there is a jump to the error handling label, there is no block to be unwound from the stack causing the loop to be exit and the error to be dumped.

This doesn't capture the whole inner workings of the exception handling mechanism but it provides a coverage of the fundamentals of the interaction between the block stack and error handling in the python virtual machine. There remains other details such as the case of an exception being thrown while handling an exception, the case of nested exception handlers and nested exceptions 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 actually left out one of the most important parts - the nuts and bolts of how a user defined class gets compiled down to bytecode and executed.

From our discussion on Python objects, we have a rough idea of how new classes *may* be created but that intuition may not totally capture the whole process from the `class ...` definition of a user to actual bytecode that creates new class objects so this chapter aims to bridge that gap and provide an exposition on how this process occurs.

As usual we start with a very simple user defined class module as shown in listing lisitng 11.0.

Listing 11.0: A simple class definition

```
    class Person:

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

When a module containing the above class definition is disassembled with the `dis` module, the stream of bytecode shown in listing 11.1 is output.

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 bytes 12 as these are the actual opcodes that create the new class object and store it so that it can be referenced 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].

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

The description of the process in the documentation though done at a very high level is pretty clear. It can be surmised from the python documentation[2], the behind the scenes process of class creation involves roughly the following processes in no particular order.

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 body of the class 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 prior to 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. In the case that this is not supplied, the class statement examines the first entry in the *tuple* of the the base classes if any. If no base classes are used, the global variable `__metaclass__` is searched for and if no value is found for this, Python uses the default meta-class. More about meta-classes is discussed in subsequent chapters.

The whole class creation process starts with the loading of the `__build_class` function onto the value stack. This is the function responsible for all the heavy lifting in creating new types/classes. Follwoign this, we have step 1 of the creation process mentioned above which is done during the compilation phase; during this step, code object representing the body of the class is loaded on the stack by the instruction at offset 2 - `LOAD_CONST`. This code object is wrapped into a function object by the `MAKE_FUNCTION` opcode and it will soon become clear why this happens. By the time, the execution loop gets to offset 10, the evaluation stack looks similar to that in Figure 11.0.

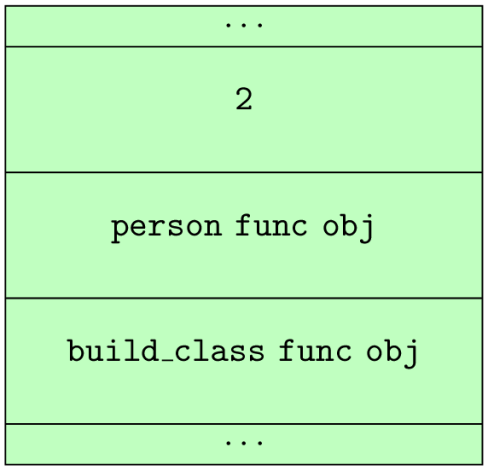


Figure 11.0 : State of evaluation stack just before `CALL_FUNCTION`

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

At offset 10, `CALL_FUNCTION` handles invokes the `__build_class` function with the values above it on the evaluation stack as argument. This function is defined in the `Python/bltinmodule.c` module. A significant part of the function is devoted to sanity checks - check that right arguments are supplied, that they have the correct type etc. After these sanity checks, the function then has to decide on the right metaclass. We state the rules for determining the correctly metaclass verbatim as described in 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 shown 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);
```

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

```
            if (_PyDict_DelItemId(mkw, &PyId_metaclass) < 0) {
                Py_DECREF(meta);
                Py_DECREF(mkw);
                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 can be 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.

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

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

    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 in the event that there is one, the namespace that is used is the result of executing the `__prepare__` attribute as the following 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;
        }
        ...
```

Afer handling the __prepare__ hook, it is now time for the actual class object to be created. First of all, the code object for the body of the class is executed within the namespace created in the previous paragraph. To understand why this is so, we disassemble the code object for the body of the class defined at the start of this chapter 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
```

When this code object is executed, the namespace will contain all the attributes of the class i.e. class attributes, methods etc. This namespace is then used in the next stage of the process 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 actually 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 really all there is to it in Python.

12. Generators: Behind the scenes.

Generators are one of the really beautiful concepts in python. A generator function is a function that contain a `yield` statement and when a generator function is invoked it returns a generator. A very simple use of generators in python is as an iterator that produces values for an iteration on demand. Listing 12.0 is a very simple example of a generator function that returns a generator 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` is a generator fucntion so calling the `firstn` function with a value does not return a simple value like a conventional function would do but rather it returns 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 the `send` method of the generator object to send values into the generator object.

In this chapter, we are not interested in the semantics of the generators objects or how the generators are or should be used. Our interest lies with the nuts and bolts of 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

The `generator` object definition is shown in listing 12.1 and going through this definition provides some intuition into how generator execution can be suspended or resumed. We can see that a generator object contains a *frame* object (recall execution frames from the chapter on frames) and a code object, two objects that are essential to 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 a generator function is called, the generator function does not run to completion and return a value rather a generator object is returned. This is possible because of the `CO_GENERATOR` flag that is set during the compilation of a generator function and this flags 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. As part of the setup process, a check of the `CO_GENERATOR` flag of the function code object is performed and in the case where it is set, rather than call the evaluation loop function, a generator object is created and returned to the caller. 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;
    }
```

What we can see from listing 12.2 is that bytecode for a generator function code object is never executed when the generator function is called - the execution of bytecode only happens when the returned generator object is executing and we look at this next.

12.2 Running a generator

We can run a generator object by passing it as argument to the `next` builtin function. This will cause the generator to execute till it hits a `yield` expression then it suspends execution. The questions of importance to us here is how the generators are able to capture execution state and update those at will.

Looking back at the generator object definition from listing 12.1, we see that generators have a field that references a frame object and this is filled in 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 obect 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`, that simply invoke another function, `gen_send_ex`, that does the actual work of resuming the execution of the generator object. Recall that before the generator object was created, all the intial setup was already carried out by the `_PyEval_EvalCodeWithName` function - frame object was intialised and variables initialised correctly, 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 indepth look at a generator function, we look at the generator function from listing 12.0. The disassembly of the generator function from 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 byteoffset 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 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 calling evaluating the frame object. From listing 12.3, the instruction that follows the `YIELD_VALUE` instruction is the `STORE_FAST` that stores whatever value is on to the top of the stack to the name on the left side of the assignment. In the case where there is no `send` function call then the value that is placed on the top of the stack is `None`.