CPYTHON'S INTERNALS: FROM SOURCE CODE TO BYTECODE

If you have been using Python for a while, then you probably wondered before what is the black magic that is done by the interpreter in order to get your computer to do what you ask it to do in your code.

This article gives a quick overview of what is going on behind the scenes when you run your code with the interpreter. There are multiple implementations of Python, but in this article we cover the internals of CPython, the reference implementation. When you install a Python distribution from python.org, you are running CPython. The "C" in CPython is a reference to the C language. The interpreter in CPython is written in C, but we can find some Python code in many of the standard library modules.

The purpose is to give a broad overview to the reader, but we will try our best to point you in the right direction if you want to dig deeper on certain topics. Some general compiler concepts are discussed in this article. For those of you who would like to have an academic reference to learn more about those concepts, we recommend reading Compilers: Principles, Techniques, and Tools (better known as the dragon book) or taking this free course with Alex Aiken.

Without further ado, let us start!

The big picture

If we simplify it to the extreme, the Python source code first gets compiled to Python bytecode and the bytecode is then run by a virtual machine. This article will discuss the former part, that is going from source code to Python bytecode. A future article will discuss the later part.

As a side note, we can compare this with what Jython, another implementation of Python, does. Jython compiles Python source code into Java bytecode and then runs this bytecode on the JVM.

The process of generating Python bytecode from source code can be divided into 3 main parts:

1. Tokenizing: Take in input a stream of characters and produce a stream of tokens in output. This is pretty standard with compilers and the piece of software that does that is generally called a tokenizer or a lexer.
2. Parsing: Build an Abstract Syntax Tree (AST) from the stream of tokens produced by the tokenizer. Also pretty standard with compilers and the piece of software that does that is called a parser.
3. Compiling to bytecode: Compile the AST into Python bytecode.

Tokenizing

This first step takes in input a stream of characters as mentioned above and groups them into sequences which are associated with a token name to form a token. A common example is to compare this step with the task of determining what are the words in an English text.

The Grammar/Tokens file contains all the types of tokens. Here's an excerpt:

```
LPAR      '('
RPAR      ')'
LSQB      '['
RSQB      ']'
COLON     ':'
COMMA     ','
SEMI      ';'
PLUS      '+'
MINUS     '-'
STAR      '*'
SLASH     '/'
```

To have a concrete example or if you want to play around with the tokens, you can use the tokenize module. Let's say that you have the following code in a module named test.py: `x = 2 + 3`. Open a Windows command prompt and type in `py -m tokenize -e test.py` and hit enter. You should see the following:

```
0,0-0,0:      ENCODING     'utf-8'
1,0-1,1:      NAME         'x'
1,2-1,3:      EQUAL        '='
1,4-1,5:      NUMBER       '2'
1,6-1,7:      PLUS         '+'
1,8-1,9:      NUMBER       '3'
1,9-1,10:     NEWLINE      ''
2,0-2,0:      ENDMARKER    ''
```

What you see in the leftmost column is the position of the token in the test.py file, so for example the second token starts at line 1, column 0 and ends at line 1, column 1. The second column shows the token name and the last column shows the value of the token. As you can see, the first token contains the encoding. Since Python 3 the default encoding is UTF-8, but you can always change it if you wish to. Also note that CPython adds a blank line at the end of the file if you don't add it yourself, so it's best practice to have one to save the interpreter from having to do it.

Here's the Python documentation on the tokenize module if you wish to learn more about it and you can find the code of the tokenizer in Parser/tokenizer.

Parsing

Now that we have identified the words in our text, we want to understand the structure of it, in other words we want to know what the sentences are and what they mean. For CPython this translates to reading the stream of tokens outputted by the tokenizer, one token at the time, from left to right, and generating the Abstract Syntax Tree (AST).

The parser and grammar

Since Python 3.9 the parser located at Parser/parser is a PEG (Parsing Expression Grammar) parser and directly generates the AST (see PEP 617). The parser is actually generated by a parser generator located at Tools/peg_generator/pegen which takes in input a grammar file located at Grammar/Python.gram that specifies the Python grammar.

Previously, the parser was an LL(1) parser and was generating a Concrete Syntax Tree (CST) sometimes also called a parse tree which was in turn transformed into the AST. This parser was also generated by a parser generator which took in input the Grammar/Grammar module that contained the Extended Backus-Naur Form (EBNF) grammar specification of the Python language.

A more detailed discussion of PEG parsers and LL(1) parsers is done in PEP 617.

The grammar in the grammar file essentially defines the structure of valid Python code. It consists of a sequence of rules of the form: “rule_name: expression”. The expressions can take the following forms:

e1 e2	to match e1 and then e2
e1 | e2	to match e1 or e2
(e)	to match e
[e] or **e?**	to optionally match e
e*	to match zero or more occurrences of e
e+	to match one or more occurrences of e
s.e+	to match one or more occurrences of e separated by s
&e	that succeeds if e can be parsed without consuming any input
!e	that fails if e can be parsed without consuming any input
~	that commits to the current alternative even if it fails

The parser matches the tokens recursively to their corresponding rule. When a match occurs, it calls the rule’s corresponding rule function that was produced by the parser generator. Each rule function calls the appropriate AST node creation functions thus creating new AST nodes and connecting them as needed. If an error occurs, the rule function backtracks and tries another rule function. The AST node creation helper functions are contained in Python/Python-ast which is generated by Parser/asdl_c from Parser/Python.asdl, which contains the definition of the AST nodes. ASDL stands for Abstract Syntax Definition Language and refers to the fact that the specification of the AST nodes is specified using the Zephyr ASDL.

Let's look at the rule related to the import statement:
`import_stmt[stmt_ty]: import_name | import_from`

This rule means that the parser will first try to match the next tokens with the import_name rule. If it succeeds, then it consumes the tokens that matched the rule and generates the required AST nodes, if it fails it tries to match the next tokens with the import_from rule.

Generating the AST with the ast module

If you want to play with the AST or just see what nodes are created for some source code, you can use the ast module. For example, let's try this code:

```
import ast
print(ast.dump(ast.parse('x = 2 + 3'), indent=4))
```

The output is the following:

```
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=BinOp(
                left=Constant(value=2),
                op=Add(),
                right=Constant(value=3)))],
    type_ignores=[])
```

As you can see, we are doing an assignment of the result from 2 + 3 to x, so we have an Assign node which contains a target, the node Name which represents x, and a value, the node BinOp which tells us we are performing an addition on 2 and 3.

For more information about CPython's parser, take a look at Python's Guide to CPython's Parser. You can also find the grammar specification in the documentation here.

Compiling to bytecode

Now that we have an AST, all that is left to do is to generate the bytecode! CPython does that in two steps: first it generates the symbol table (or symtable) and second it transforms the AST into a Control Flow Graph (CFG) and flattens it to get the bytecode from it.

The symbole table and CFG

The symbol table is a structure that contains a list of namespaces, global variables and local variables. It is basically used to figure out the scoping of each variable. The symtable structure is located at Include/internal/pycore_symtable.h.

A CFG is a directed graph that represents the flow of a program. Each node of the CFG is made of a basic block which represents a sequence of instructions to execute with a single entry point at the beginning of the sequence and a single exit point at the end. Take for example an if else statement. The instructions corresponding to the condition is contained in a first block which ends with a conditional jump based on the result of the condition. That means that this first block points to both the block corresponding to the body of the if and the block corresponding to the body of the else. So essentially the CFG is a graph of bytecode blocks where edges are jumps to other blocks.

To generate the symbol table, each node of the AST is recursively visited. An appropriate function is called for every node visited which feeds the symbol table. If you are interested in playing around with the symbol table, the symtable module allows you to create one from some source code and to examine it.

Similarly, to generate the CFG each node of the AST is recursively visited. An appropriate function is called for every node to create the basic blocks, emit the bytecode and create the jumps to basic blocks. Once this is done the CFG is flattened to a single bytecode sequence using a post-order depth-first search during the assembly. Finally, some bytecode optimizations are applied such as constant folding. Once all of this is over the bytecode is contained in a PyCodeObject instance. A lot of what we just discussed is done in Python/compile.c if you want to take a look.

It's worth noting that Python bytecode is considered a CPython implementation detail which means it can change from one minor version to another (e.g. from 3.10 to 3.11). Also the bytecode is stack-based, so when the bytecode is executed it pushes or pops objects on a stack. You can see the list of opcodes (the single instructions that make up the bytecode) at Lib/opcode.py and read on what each of them does in the documentation of the dis module.

Disassembling bytecode with the dis module

The dis module allows you to disassemble the bytecode from a code object. Used with the compile() built-in function you can inspect what opcodes have been generated for some source code. Here's an example:

```
import dis
co = compile('x = y + 1', 'test.py', mode='exec')
dis.dis(co)
```

Which on Python 3.10 prints:

```
1           0 LOAD_NAME                0 (y)
            2 LOAD_CONST               0 (1)
            4 BINARY_ADD
            6 STORE_NAME               1 (x)
            8 LOAD_CONST               1 (None)
           10 RETURN_VALUE
```

The dis() function can also take directly the string containing the source code in argument, but we wanted to show you that you can use the compile() function to get the code object. As you see, each line printed represents a bytecode instruction. The one that you see completely on the left in the first line represents the line number of the source code for the following instructions, so the first line in our case. The numbers in the first full column on the left correspond to the offset of that instruction - basically the byte index in the sequence of bytes that make the bytecode. The following column as you guessed contains the names of the instructions (the opnames). The column on its right contains the arguments of the instructions (the opargs) which are used by the CPython virtual machine and the last column contains the human-friendly interpretations of these arguments. If we take the oparg of the first line for example, in the case of the LOAD_NAME opcode it represents an index into the co_names property of the code object, so co.co_names[0] should return 'y'. The meaning of the oparg changes from one instruction to another, it is explained in the dis module documentation and you can see what is really done with those in Python/ceval.c.

Conclusion

This concludes this article, you now have a better understanding of what is going on under the hood of the CPython interpreter! There are still a lot of subjects that could be discussed regarding CPython internals such as the virtual machine, the garbage collector and reference counting, the different Python objects and generators. Plenty of subjects for our next article so stay tuned!

References

If you would like to dig deeper into what you've just read, we recommend taking a look at the following references which have been very useful to write this article:

- The Python Developer's Guide holds a lot of interesting information and especially the pages on CPython's parser and the design of CPython's compiler. Many additional resources are available on this page also.
- This article by Anthony Shaw is very complete.
- For those of you who prefer videos, this video from PyCon Canada 2013 with Brett Cannon is a good introduction even if a few things have changed since that time.