In the name of Allah

How CPython Compiler Works

Mahdi Haghverdi

Isfahan University

Content

Overview

Overview

- Which steps does CPython take to compile your source code?
- Why these steps?
- How they are done?

Diagram

- ```
  ------------------------------------------------------------
  | Decoding -> Tokenizing -> Parsing -> AST | -> Compiling |
  ------------------------------------------------------------
  ```
- Front-end: Decoding, Tokenizing, Parsing and AST
- Back-end: Compiling
  ```

Explanation

- We've got a front-end and a back-end part in this process.
- Front-end: Getting down to the AST
- Back-end: Get the generated AST and compile it down to something
- Good example is PyPy which is a front-end for Python
- Ease of writing the code
- A better view to the process

Explanation

- We've got a front-end and a back-end part in this process.
- Front-end: Getting down to the AST
- Back-end: Get the generated AST and compile it down to something
- Good example is PyPy which is a front-end for Python
- Ease of writing the code
- A better view to the process

Explanation

- We've got a front-end and a back-end part in this process.
- Front-end: Getting down to the AST
- Back-end: Get the generated AST and compile it down to something
- Good example is PyPy which is a front-end for Python
- Ease of writing the code
- A better view to the process

Explanation

- We've got a front-end and a back-end part in this process.
- Front-end: Getting down to the AST
- Back-end: Get the generated AST and compile it down to something
- Good example is PyPy which is a front-end for Python
- Ease of writing the code
- A better view to the process

Explanation

- We've got a front-end and a back-end part in this process.
- Front-end: Getting down to the AST
- Back-end: Get the generated AST and compile it down to something
- Good example is PyPy which is a front-end for Python
- Ease of writing the code
- A better view to the process

Explanation

- We've got a front-end and a back-end part in this process.
- Front-end: Getting down to the AST
- Back-end: Get the generated AST and compile it down to something
- Good example is PyPy which is a front-end for Python
- Ease of writing the code
- A better view to the process

Decoding - “Bytes” to “Text”

Decoding - “Bytes” to “Text”

- Translate bytes from disk to actual text

Encoding Declaration

- As of PEP 263, you can specify the encoding of your Python module (basically a module is a text file which python code is written into) at the very top line of the file something like:

Encoding Declaration (Cont'd)

Declaration:

```
1  #!/usr/bin/python
2  # -*- coding: <encoding name> -*-
```

e.g.

```
1  #!/usr/bin/python
2  # -*- coding: ascii -*-
3
4  import math
5  print(math.sin(math.radians(90)))  # 1.0
```

Encoding Declaration (Cont'd)

Which gets compiled like this:

```
1                re.compile("conding[:=]\s*([-\w.]+)")
```

Default Encoding and Non-ASCII Characters

- From PEP 3120 UTF-8 is considered as the default enconding, and along with this with PEP 3131
- Python supports Non-ASCII identifiers also, this means that you can use french or germen alphabet (with accent) in your variable names, like:

Default Encoding and Non-ASCII Characters

- From PEP 3120 UTF-8 is considered as the default enconding, and along with this with PEP 3131
- Python supports Non-ASCII identifiers also, this means that you can use french or germen alphabet (with accent) in your variable names, like:

Default Encoding and Non-ASCII Characters (Cont'd)

```
1  löwis = 'Löwis'
2  print(löwis)
```

Tokenizing - “Text” to “Words”

Tokenizing - “Text” to “Words”

- Take the text and break it up into words

Tokenizing - “Text” to “Words” (Cont’d)

- At this point we have our text, but we’ve got just a bunch of characters following each other
- NOW we do *tokenizing*
  which the term *token* is just a fancy word for words.

Tokenizing - “Text” to “Words” (Cont’d)

- At this point we have our text, but we’ve got just a bunch of characters following each other
- NOW we do *tokenizing*
  which the term *token* is just a fancy word for words.

Tokenizing - “Text” to “Words” (Cont’d)

How do we know to break the word? For instance in english language, its based on a space, but for programs it does not make sence.
i.e. there’s is no diffrenece between:

```
1    print((lambda x: x*2 - 1)(2))  # 3
```

and

```
1    print((lambda    x: x * 2-    1)(2))  # 3
```

these should be tokenized like:

Tokenizing - “Text” to “Words” (Cont’d)

```
import shlex
print(list(shlex.shlex(
'print((lambda x: x*2 - 1)(2))  # 3'
))

print(list(shlex.shlex(
'print((lambda    x: x * 2-    1)(2))  # 3'
)))
```

as

```
['print', '(', '(', 'lambda', 'x', ':', 'x',
'*', '2', '-', '1', ')', '(', '2', ')', ')']
```

`tokenize` library

```
$ echo 'print((lambda x: x*2-1)(2)) # 3' | python -m tokenize -
1,0-1,5:            NAME           'print'
1,5-1,6:            LPAR           '('
1,6-1,7:            LPAR           '('
1,7-1,13:           NAME           'lambda'
1,17-1,18:          NAME           'x'
1,18-1,19:          COLON          ':'
1,20-1,21:          NAME           'x'
1,22-1,23:          STAR           '*'
1,24-1,25:          NUMBER         '2'
1,25-1,26:          MINUS          '-'
1,30-1,31:          NUMBER         '1'
1,31-1,32:          RPAR           ')'
```

```
1,32-1,33:            LPAR           '('
1,33-1,34:            NUMBER         '2'
1,34-1,35:            RPAR           ')'
1,35-1,36:            RPAR           ')'
1,38-1,41:            COMMENT        '# 3'
1,41-1,42:            NEWLINE        '\n'
2,0-2,0:              ENDMARKER      ''
```

Parsing - “Words” to “Sentence”

Parsing - “Words” to “Sentence”

- Take the words and make sentences out of them

Parsing - “Words” to “Sentence” (Cont’d)

- Now we have broken everything into words, we can care about how to structure our sentences and make them meaningful following specific grammer rules.
- In parsing we use a grammer to define a structure, you can check Python grammer in https://github.com/python/cpython/tree/main/Grammar

Parsing - “Words” to “Sentence” (Cont’d)

This is a piece of Python grammer (version 3.9)

```
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)
3
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt |
yield_stmt
break_stmt: 'break'
```

Parsing - “Words” to “Sentence” (Cont’d)

- Before python 3.9’s PEG parser, Python parser was a LL(1) parser
- It was probably was oldest python code which was written by “Guido van Rossum” and hadn’t changed way back since Decemeber of 1998 :-)

Parsing - “Words” to “Sentence” (Cont’d)

- Before python 3.9’s PEG parser, Python parser was a LL(1) parser
- It was probably was oldest python code which was written by “Guido van Rossum” and hadn’t changed way back since Decemeber of 1998 :-)

Abstract Systax Tree - “Sentence” to “Semantics”

Abstract Systax Tree - “Sentence” to “Semantics”

- Take the sentences and figures out what the heck you are saying

Abstract Systax Tree - “Sentence” to “Semantics” (Cont’d)

- Here we take the sentence structure and we make sure that *it makes sence*
  - whether to Python semantics
  - or whether for example to math arithmatic expression rules (e.g. parenthesis).
- In 2006 adding ast to python was done through something called Zephyr ASDL rule or grammer:

Abstract Systax Tree - “Sentence” to “Semantics” (Cont’d)

- Here we take the sentence structure and we make sure that *it makes sence*
  - whether to Python semantics
  - or whether for example to math arithmatic expression rules (e.g. parenthesis).
- In 2006 adding ast to python was done through something called Zephyr ASDL rule or grammer:

Abstract Systax Tree - “Sentence” to “Semantics” (Cont’d)

- Here we take the sentence structure and we make sure that *it makes sence*
  - whether to Python semantics
  - or whether for example to math arithmatic expression rules (e.g. parenthesis).
- In 2006 adding ast to python was done through something called Zephyr ASDL rule or grammer:

Abstract Systax Tree - “Sentence” to “Semantics” (Cont’d)

- Here we take the sentence structure and we make sure that *it makes sence*
  - whether to Python semantics
  - or whether for example to math arithmatic expression rules (e.g. parenthesis).
- In 2006 adding ast to python was done through something called Zephyr ASDL rule or grammer:

Abstract Systax Tree - “Sentence” to “Semantics” (Cont’d)

```
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
```

Abstract Systax Tree - “Sentence” to “Semantics” (Cont’d)

`x = 2 + 3` is:

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

Compiling - “Sematics” to “Bytecode”

Compiling - “Sematics” to “Bytecode”

- Take the AST and generates the bytecode to be executed

Bytecode

- Python implementation detail [1]
- Stack-based
- (Python 3.9 and before) 101 instructions

[1] t means it may change in every minor release and you must not rely on what you have now and to expect to have that in the future

Bytecode (Cont'd)

```
1  from dis import dis  # disassembler
2
3  def func(): x = y + 2
4
5  dis(func.__code__)
6
7  # output:
8      1    0 LOAD_GLOBAL    0 (y)
9           2 LOAD_CONST     1 (2)
10          4 BINARY_ADD
11          6 STORE_FAST     0 (x)
12          8 LOAD_CONST     0 (None)
13         10 RETURN_VALUE
```

Based on

Based on

- “Design of CPython’s Compiler”
  http://docs.python.org/devguide/compiler.html