In the name of Allah

How CPython Compiler Works

Mahdi Haghverdi

Isfahan University

Content

Overview

Overview

- Which steps does CPython takes 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: to 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: to 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: to 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: to 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: to 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: to 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 -e
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

Abstract Systax Tree - “Sentence” to “Semantics”

Abstract Systax Tree - “Sentence” to “Semantics”

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

Compiling - “Sematics” to “Bytecode”

Compiling - “Sematics” to “Bytecode”

- Take the AST and generates the bytecode to be executed