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 (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

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