DD2358 – Bytecode: Under the Hood with `dis`

Stefano Markidis

KTH Royal Institute of Technology

Intended Learning Outcomes

- Use the `dis` (disassembler) module to inspect the Python bytecode of selected high-level Python functions of our code
- Analyze the results of the `dis` output.

Python Bytecode

- Python is often described as an interpreted language
  - your source code is translated into native CPU instructions <u>as the program runs</u>
    - > *but this is only partially correct*
- Python, like many interpreted languages, actually **compiles source code to a set of instructions for a virtual machine**
  - This intermediate format is called ***bytecode***.
  - The Python interpreter is an implementation of that virtual machine.

Investigating the Python Bytecode

- We have reviewed various ways to measure the cost of Python code
  - for both CPU and RAM usage
- We haven’t yet looked at the **<u>underlying bytecode</u>** used by the interpreter.
- Understanding what’s going on “under the hood” helps to build a **mental model of what’s happening in slow functions**, and it’ll help when you come to compile your code.
- We are going to look at how obtain Python bytecode.

Using the `dis` Module to Examine CPython Bytecode

- The `dis` (disassembler) module lets us inspect the underlying bytecode that we run inside the **stack-based CPython virtual machine**.
- Having an understanding of what's happening in the virtual machine that runs our higher-level Python code will help you to understand **why some styles of coding are faster than others**.
- It will also help when we come to use a tool like `Cython`, which steps outside of Python and generates C code.
- The `dis` module is built in. We can pass it code or a module, and it will print out a disassembly.

Dissambling a Function in JuliaSet.py

Python 3.8.9 (default, Aug 3 2021, 19:21:54)

>>> import JuliaSet

>>> import dis

>>> dis.dis(JuliaSet.calculate_z_serial_purepython)

```
63           0 LOAD_CONST               1 (0)
             2 BUILD_LIST               1
             4 LOAD_GLOBAL              0 (len)
             6 LOAD_FAST                1 (zs)
             8 CALL_FUNCTION            1
            10 BINARY_MULTIPLY
            12 STORE_FAST               3 (output)

64          14 LOAD_GLOBAL              1 (range)
            16 LOAD_GLOBAL              0 (len)
            18 LOAD_FAST                1 (zs)
            20 CALL_FUNCTION            1
            22 CALL_FUNCTION            1
            24 GET_ITER
       >>   26 FOR_ITER                74 (to 102)
            28 STORE_FAST               4 (i)

65          30 LOAD_CONST               1 (0)
            32 STORE_FAST               5 (n)

66          34 LOAD_FAST                1 (zs)
            36 LOAD_FAST                4 (i)
            38 BINARY_SUBSCR
            40 STORE_FAST               6 (z)

67          42 LOAD_FAST                2 (cs)
            44 LOAD_FAST                4 (i)
            46 BINARY_SUBSCR
            48 STORE_FAST               7 (c)

68     >>   50 LOAD_GLOBAL              2 (abs)
            52 LOAD_FAST                6 (z)
            54 CALL_FUNCTION            1
            56 LOAD_CONST               2 (2)
            58 COMPARE_OP               0 (<)
```

Analyzing the Bytecode

- The first column contains **line numbers**
- The second column contains several >> symbols
  - *these are the destinations for **jump points** elsewhere in the code.*
- The third column is the **operation address**
- The fourth has the **operation name**
- The fifth column contains the **parameters for the operation**
- The sixth column contains **annotations** to help line up the bytecode with the original Python parameters.

```
63           0 LOAD_CONST               1 (0)
             2 BUILD_LIST               1
             4 LOAD_GLOBAL              0 (len)
             6 LOAD_FAST                1 (zs)
             8 CALL_FUNCTION            1
            10 BINARY_MULTIPLY
            12 STORE_FAST               3 (output)

64          14 LOAD_GLOBAL              1 (range)
            16 LOAD_GLOBAL              0 (len)
            18 LOAD_FAST                1 (zs)
            20 CALL_FUNCTION            1
            22 CALL_FUNCTION            1
            24 GET_ITER
      >>    26 FOR_ITER                74 (to 102)
            28 STORE_FAST               4 (i)

65          30 LOAD_CONST               1 (0)
            32 STORE_FAST               5 (n)

66          34 LOAD_FAST                1 (zs)
            36 LOAD_FAST                4 (i)
            38 BINARY_SUBSCR
            40 STORE_FAST               6 (z)

67          42 LOAD_FAST                2 (cs)
            44 LOAD_FAST                4 (i)
            46 BINARY_SUBSCR
            48 STORE_FAST               7 (c)

68    >>    50 LOAD_GLOBAL              2 (abs)
            52 LOAD_FAST                6 (z)
            54 CALL_FUNCTION            1
            56 LOAD_CONST               2 (2)
            58 COMPARE_OP               0 (<)
```

Example of Analysis – First Line

- The bytecode starts on Python line 63 by putting the **constant value 0 onto the stack**, and then it builds a single-element list.
- Next, it searches the namespaces to **find the len function**, puts it on the stack, searches the namespaces again to find zs, and then puts that onto the stack.
- It **calls the len function from the stack**, which consumes the zs reference in the stack
- then it applies a binary multiply to the last two arguments (the length of zs and the single-element list) and stores the result in output.

```
63            0 LOAD_CONST               1 (0)
              2 BUILD_LIST               1
              4 LOAD_GLOBAL              0 (len)
              6 LOAD_FAST                1 (zs)
              8 CALL_FUNCTION            1
             10 BINARY_MULTIPLY
             12 STORE_FAST               3 (output)

64           14 LOAD_GLOBAL              1 (range)
             16 LOAD_GLOBAL              0 (len)
             18 LOAD_FAST                1 (zs)
             20 CALL_FUNCTION            1
             22 CALL_FUNCTION            1
             24 GET_ITER
        >>   26 FOR_ITER                74 (to 102)
             28 STORE_FAST               4 (i)

65           30 LOAD_CONST               1 (0)
             32 STORE_FAST               5 (n)

66           34 LOAD_FAST                1 (zs)
             36 LOAD_FAST                4 (i)
             38 BINARY_SUBSCR
             40 STORE_FAST               6 (z)

67           42 LOAD_FAST                2 (cs)
             44 LOAD_FAST                4 (i)
             46 BINARY_SUBSCR
             48 STORE_FAST               7 (c)

68      >>   50 LOAD_GLOBAL              2 (abs)
             52 LOAD_FAST                6 (z)
             54 CALL_FUNCTION            1
             56 LOAD_CONST               2 (2)
             58 COMPARE_OP               0 (<)
```

To Summarize

- The `dis` (disassembler) module allows us to inspect the Python bytecode
- Investigating bytecode might help in understand how high-level Python is translated to low-level bytecode and compare different approaches (that translates to different bytecode)