dis — Python Bytecode Disassembler

Purpose: Convert code objects to a human-readable representation of the bytecodes for analysis.

The `dis` module includes functions for working with Python bytecode by *disassembling* it into a more human-readable form. Reviewing the bytecodes being executed by the interpreter is a good way to hand-tune tight loops and perform other kinds of optimizations. It is also useful for finding race conditions in multi-threaded applications, since it can be used to estimate the point in the code where thread control may switch.

> **Warning**
>
> The use of bytecodes is a version-specific implementation detail of the CPython interpreter. Refer to `Include/opcode.h` in the source code for the version of the interpreter you are using to find the canonical list of bytecodes.

Basic Disassembly

The function `dis()` prints the disassembled representation of a Python code source (module, class, method, function, or code object). A module such as `dis_simple.py` can be disassembled by running `dis` from the command line.

```
# dis_simple.py
```

```python
#!/usr/bin/env python3
# encoding: utf-8

my_dict = {'a': 1}
```

The output is organized into columns with the original source line number, the instruction address within the code object, the opcode name, and any arguments passed to the opcode.

```
$ python3 -m dis dis_simple.py

  4           0 LOAD_CONST               0 ('a')
              2 LOAD_CONST               1 (1)
              4 BUILD_MAP                1
              6 STORE_NAME               0 (my_dict)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

In this case, the source translates to four different operations to create and populate the dictionary, then save the results to a local variable. Since the Python interpreter is stack-based, the first steps are to put the constants onto the stack in the correct order with `LOAD_CONST`, and then use `BUILD_MAP` to pop off the new key and value to be added to the dictionary. The resulting `dict` object is bound to the name `my_dict` with `STORE_NAME`.

Disassembling Functions

Unfortunately, disassembling an entire module does not recurse into functions automatically.

```
# dis_function.py
```

```python
#!/usr/bin/env python3
# encoding: utf-8


def f(*args):
    nargs = len(args)
    print(nargs, args)


if __name__ == '__main__':
    import dis
```

```
    import dis
    dis.dis(f)
```

The results of disassembling `dis_function.py` show the operations for loading the function’s code object onto the stack and then turning it into a function (LOAD_CONST, MAKE_FUNCTION), followed by the body of the function.

```
$ python3 -m dis dis_function.py

  5           0 LOAD_CONST               0 (<code object f at
0x102c2df60, file "dis_function.py", line 5>)
              2 LOAD_CONST               1 ('f')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (f)

 10           8 LOAD_NAME                1 (__name__)
             10 LOAD_CONST               2 ('__main__')
             12 COMPARE_OP               2 (==)
             14 POP_JUMP_IF_FALSE       34

 11          16 LOAD_CONST               3 (0)
             18 LOAD_CONST               4 (None)
             20 IMPORT_NAME              2 (dis)
             22 STORE_NAME               2 (dis)

 12          24 LOAD_NAME                2 (dis)
             26 LOAD_METHOD              2 (dis)
             28 LOAD_NAME                0 (f)
             30 CALL_METHOD              1
             32 POP_TOP
        >>   34 LOAD_CONST               4 (None)
             36 RETURN_VALUE

Disassembly of <code object f at 0x102c2df60, file
"dis_function.py", line 5>:
  6           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (args)
              4 CALL_FUNCTION            1
              6 STORE_FAST               1 (nargs)

  7           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                1 (nargs)
             12 LOAD_FAST                0 (args)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

Earlier versions of Python did not include function bodies in module disassemblies automatically. To see the disassembled version of a function, pass the function directly to `dis()`.

```
$ python3 dis_function.py

  6           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (args)
              4 CALL_FUNCTION            1
              6 STORE_FAST               1 (nargs)

  7           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                1 (nargs)
             12 LOAD_FAST                0 (args)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

To print a summary of the function, including information about the arguments and names it uses, call `show_code()`, passing the function as the first argument.

```
#!/usr/bin/env python3
# encoding: utf-8
```

```
# encoding: utf-8


def f(*args):
    nargs = len(args)
    print(nargs, args)


if __name__ == '__main__':
    import dis
    dis.show_code(f)
```

The argument to show_code() is passed to code_info(), which returns a nicely formatted summary of the function, method, code string, or other code object, ready to be printed.

```
$ python3 dis_show_code.py

Name:              f
Filename:          dis_show_code.py
Argument count:    0
Kw-only arguments: 0
Number of locals:  2
Stack size:        3
Flags:             OPTIMIZED, NEWLOCALS, VARARGS, NOFREE
Constants:
   0: None
Names:
   0: len
   1: print
Variable names:
   0: args
   1: nargs
```

Classes

Classes can be passed to dis(), in which case all of the methods are disassembled in turn.

```
# dis_class.py
```

```
#!/usr/bin/env python3
# encoding: utf-8

import dis


class MyObject:
    """Example for dis."""

    CLASS_ATTRIBUTE = 'some value'

    def __str__(self):
        return 'MyObject({})'.format(self.name)

    def __init__(self, name):
        self.name = name


dis.dis(MyObject)
```

The methods are listed in alphabetical order, not the order they appear in the file.

```
$ python3 dis_class.py

Disassembly of __init__:
 16           0 LOAD_FAST                1 (name)
              2 LOAD_FAST                0 (self)
              4 STORE_ATTR               0 (name)
              6 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

```
              8 RETURN_VALUE

Disassembly of __str__:
 13           0 LOAD_CONST               1 ('MyObject({})')
              2 LOAD_METHOD              0 (format)
              4 LOAD_FAST                0 (self)
              6 LOAD_ATTR                1 (name)
              8 CALL_METHOD              1
             10 RETURN_VALUE
```

Source Code

It is often more convenient to work with the source code for a program than with the code objects themselves. The functions in dis accept string arguments containing source code, and convert them to code objects before producing the disassembly or other output.

```
# dis_string.py

import dis

code = """
my_dict = {'a': 1}
"""

print('Disassembly:\n')
dis.dis(code)

print('\nCode details:\n')
dis.show_code(code)
```

Passing a string lets you save the step of compiling the code and holding a reference to the results yourself, which is more convenient in cases when statements outside of a function are being examined.

```
$ python3 dis_string.py

Disassembly:

  2           0 LOAD_CONST               0 ('a')
              2 LOAD_CONST               1 (1)
              4 BUILD_MAP                1
              6 STORE_NAME               0 (my_dict)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE

Code details:

Name:              <module>
Filename:          <disassembly>
Argument count:    0
Kw-only arguments: 0
Number of locals:  0
Stack size:        2
Flags:             NOFREE
Constants:
   0: 'a'
   1: 1
   2: None
Names:
   0: my_dict
```

Using Disassembly to Debug

Sometimes when debugging an exception it can be useful to see which bytecode caused a problem. There are a couple of ways to disassemble the code around an error. The first is by using `dis()` in the interactive interpreter to report about the last exception. If no argument is passed to `dis()`, then it looks for an exception and shows the disassembly of the top of the stack that caused it.

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
```

```
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import dis
>>> j = 4
>>> i = i + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
>>> dis.dis()
  1 -->       0 LOAD_NAME                0 (i)
              3 LOAD_CONST               0 (4)
              6 BINARY_ADD
              7 STORE_NAME               0 (i)
             10 LOAD_CONST               1 (None)
             13 RETURN_VALUE
>>>
```

The `-->` after the line number indicates the opcode that caused the error. There is no `i` variable defined, so the value associated with the name cannot be loaded onto the stack.

A program can also print the information about an active traceback by passing it to `distb()` directly. In this example, there is a `DivideByZero` exception, but since the formula has two divisions it may not be clear which part is zero.

```
# dis_traceback.py

#!/usr/bin/env python3
# encoding: utf-8

i = 1
j = 0
k = 3

try:
    result = k * (i / j) + (i / k)
except Exception:
    import dis
    import sys
    exc_type, exc_value, exc_tb = sys.exc_info()
    dis.distb(exc_tb)
```

The error is easy to spot when it is loaded onto the stack in the disassembled version. The bad operation is highlighted with the `-->`, and the previous line pushes the value for `j` onto the stack.

```
    $ python3 dis_traceback.py

      4           0 LOAD_CONST               0 (1)
                  2 STORE_NAME               0 (i)

      5           4 LOAD_CONST               1 (0)
                  6 STORE_NAME               1 (j)

      6           8 LOAD_CONST               2 (3)
                 10 STORE_NAME               2 (k)

      8          12 SETUP_EXCEPT            24 (to 38)

      9          14 LOAD_NAME                2 (k)
                 16 LOAD_NAME                0 (i)
                 18 LOAD_NAME                1 (j)
        -->      20 BINARY_TRUE_DIVIDE
                 22 BINARY_MULTIPLY
                 24 LOAD_NAME                0 (i)
                 26 LOAD_NAME                2 (k)
                 28 BINARY_TRUE_DIVIDE
                 30 BINARY_ADD
                 32 STORE_NAME               3 (result)

...trimmed...
```

Performance Analysis of Loops

Besides debugging errors, dis can also help identify performance issues. Examining the disassembled code is especially useful with tight loops where the number of Python instructions is low but they translate to an inefficient set of bytecodes. The helpfulness of the disassembly can be seen by examining a few different implementations of a class, `Dictionary`, that reads a list of words and groups them by their first letter.

```
# dis_test_loop.py

import dis
import sys
import textwrap
import timeit

module_name = sys.argv[1]
module = __import__(module_name)
Dictionary = module.Dictionary

dis.dis(Dictionary.load_data)
print()
t = timeit.Timer(
    'd = Dictionary(words)',
    textwrap.dedent("""
    from {module_name} import Dictionary
    words = [
        l.strip()
        for l in open('/usr/share/dict/words', 'rt')
    ]
    """).format(module_name=module_name)
)
iterations = 10
print('TIME: {:0.4f}'.format(t.timeit(iterations) / iterations))
```

The test driver application `dis_test_loop.py` can be used to run each incarnation of the `Dictionary` class, starting with a straightforward, but slow, implementation.

```
# dis_slow_loop.py

#!/usr/bin/env python3
# encoding: utf-8


class Dictionary:

    def __init__(self, words):
        self.by_letter = {}
        self.load_data(words)

    def load_data(self, words):
        for word in words:
            try:
                self.by_letter[word[0]].append(word)
            except KeyError:
                self.by_letter[word[0]] = [word]
```

Running the test program with this version shows the disassembled program and the amount of time it takes to run.

```
$ python3 dis_test_loop.py dis_slow_loop

 12           0 SETUP_LOOP              83 (to 86)
              3 LOAD_FAST                1 (words)
              6 GET_ITER
        >>    7 FOR_ITER                75 (to 85)
             10 STORE_FAST               2 (word)

 13          13 SETUP_EXCEPT            28 (to 44)

 14          16 LOAD_FAST                0 (self)
```

```
             19 LOAD_ATTR                0 (by_letter)
             22 LOAD_FAST                2 (word)
             25 LOAD_CONST               1 (0)
             28 BINARY_SUBSCR
             29 BINARY_SUBSCR
             30 LOAD_ATTR                1 (append)
             33 LOAD_FAST                2 (word)
             36 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             39 POP_TOP
             40 POP_BLOCK
             41 JUMP_ABSOLUTE            7

 15     >>   44 DUP_TOP
             45 LOAD_GLOBAL              2 (KeyError)
             48 COMPARE_OP              10 (exception match)
             51 POP_JUMP_IF_FALSE       81
             54 POP_TOP
             55 POP_TOP
             56 POP_TOP

 16          57 LOAD_FAST                2 (word)
             60 BUILD_LIST               1
             63 LOAD_FAST                0 (self)
             66 LOAD_ATTR                0 (by_letter)
             69 LOAD_FAST                2 (word)
             72 LOAD_CONST               1 (0)
             75 BINARY_SUBSCR
             76 STORE_SUBSCR
             77 POP_EXCEPT
             78 JUMP_ABSOLUTE            7
        >>   81 END_FINALLY
             82 JUMP_ABSOLUTE            7
        >>   85 POP_BLOCK
        >>   86 LOAD_CONST               0 (None)
             89 RETURN_VALUE

TIME: 0.0568
```

The previous output shows `dis_slow_loop.py` taking 0.0568 seconds to load the 235886 words in the copy of `/usr/share/dict/words` on OS X. That is not too bad, but the accompanying disassembly shows that the loop is doing more work than it needs to. As it enters the loop in opcode 13, it sets up an exception context (SETUP_EXCEPT). Then it takes six opcodes to find `self.by_letter[word[0]]` before appending `word` to the list. If there is an exception because `word[0]` is not in the dictionary yet, the exception handler does all of the same work to determine `word[0]` (three opcodes) and sets `self.by_letter[word[0]]` to a new list containing the word.

One technique to eliminate the exception setup is to pre-populate `self.by_letter` with one list for each letter of the alphabet. That means the list for the new word should always be found, and the value can be saved after the lookup.

```
# dis_faster_loop.py
```

```python
#!/usr/bin/env python3
# encoding: utf-8

import string


class Dictionary:

    def __init__(self, words):
        self.by_letter = {
            letter: []
            for letter in string.ascii_letters
        }
        self.load_data(words)

    def load_data(self, words):
        for word in words:
            self.by_letter[word[0]].append(word)
```

The change cuts the number of opcodes in half, but only shaves the time down to 0.0567 seconds. Obviously the exception handling had some overhead, but not a significant amount.

```
$ python3 dis_test_loop.py dis_faster_loop

 17           0 SETUP_LOOP              38 (to 41)
              3 LOAD_FAST                1 (words)
              6 GET_ITER
        >>    7 FOR_ITER                30 (to 40)
             10 STORE_FAST               2 (word)

 18          13 LOAD_FAST                0 (self)
             16 LOAD_ATTR                0 (by_letter)
             19 LOAD_FAST                2 (word)
             22 LOAD_CONST               1 (0)
             25 BINARY_SUBSCR
             26 BINARY_SUBSCR
             27 LOAD_ATTR                1 (append)
             30 LOAD_FAST                2 (word)
             33 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             36 POP_TOP
             37 JUMP_ABSOLUTE            7
        >>   40 POP_BLOCK
        >>   41 LOAD_CONST               0 (None)
             44 RETURN_VALUE

TIME: 0.0567
```

The performance can be improved further by moving the lookup for `self.by_letter` outside of the loop (the value does not change, after all).

```python
# dis_fastest_loop.py

#!/usr/bin/env python3
# encoding: utf-8

import collections


class Dictionary:

    def __init__(self, words):
        self.by_letter = collections.defaultdict(list)
        self.load_data(words)

    def load_data(self, words):
        by_letter = self.by_letter
        for word in words:
            by_letter[word[0]].append(word)
```

Opcodes 0-6 now find the value of `self.by_letter` and save it as a local variable `by_letter`. Using a local variable only takes a single opcode, instead of two (statement 22 uses `LOAD_FAST` to place the dictionary onto the stack). After this change, the run time is down to 0.0473 seconds.

```
$ python3 dis_test_loop.py dis_fastest_loop

 14           0 LOAD_FAST                0 (self)
              3 LOAD_ATTR                0 (by_letter)
              6 STORE_FAST               2 (by_letter)

 15           9 SETUP_LOOP              35 (to 47)
             12 LOAD_FAST                1 (words)
             15 GET_ITER
        >>   16 FOR_ITER                27 (to 46)
             19 STORE_FAST               3 (word)

 16          22 LOAD_FAST                2 (by_letter)
             25 LOAD_FAST                3 (word)
```

```
             28 LOAD_CONST               1 (0)
             31 BINARY_SUBSCR
             32 BINARY_SUBSCR
             33 LOAD_ATTR                1 (append)
             36 LOAD_FAST                3 (word)
             39 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             42 POP_TOP
             43 JUMP_ABSOLUTE           16
        >>   46 POP_BLOCK
        >>   47 LOAD_CONST               0 (None)
             50 RETURN_VALUE

TIME: 0.0473
```

A further optimization, suggested by Brandon Rhodes, is to eliminate the Python version of the `for` loop entirely. If `itertools.groupby()` is used to arrange the input, the iteration is moved to C. This is safe because the inputs are known to be sorted. If that was not the case, the program would need to sort them first.

```python
# dis_eliminate_loop.py

#!/usr/bin/env python3
# encoding: utf-8

import operator
import itertools


class Dictionary:

    def __init__(self, words):
        self.by_letter = {}
        self.load_data(words)

    def load_data(self, words):
        # Arrange by letter
        grouped = itertools.groupby(
            words,
            key=operator.itemgetter(0),
        )
        # Save arranged sets of words
        self.by_letter = {
            group[0][0]: group
            for group in grouped
        }
```

The `itertools` version takes only 0.0332 seconds to run, about 60% of the run time for the original.

```
    $ python3 dis_test_loop.py dis_eliminate_loop

     16           0 LOAD_GLOBAL              0 (itertools)
                  3 LOAD_ATTR                1 (groupby)

     17           6 LOAD_FAST                1 (words)
                  9 LOAD_CONST               1 ('key')

     18          12 LOAD_GLOBAL              2 (operator)
                 15 LOAD_ATTR                3 (itemgetter)
                 18 LOAD_CONST               2 (0)
                 21 CALL_FUNCTION            1 (1 positional, 0
    keyword pair)
                 24 CALL_FUNCTION          257 (1 positional, 1
    keyword pair)
                 27 STORE_FAST               2 (grouped)

     21          30 LOAD_CONST               3 (<code object
    <dictcomp> at 0x101517930, file ".../dis_eliminate_loop.py",
line 21>)
                 33 LOAD_CONST               4
    ('Dictionary.load_data.<locals>.<dictcomp>')
```

```
('Dictionary.load_data.<locals>.<dictcomp>')
             36 MAKE_FUNCTION            0

 23          39 LOAD_FAST                2 (grouped)
             42 GET_ITER
             43 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             46 LOAD_FAST                0 (self)
             49 STORE_ATTR               4 (by_letter)
             52 LOAD_CONST               0 (None)
             55 RETURN_VALUE

TIME: 0.0332
```

Compiler Optimizations

Disassembling compiled source also exposes some of the optimizations made by the compiler. For example, literal expressions are folded during compilation, when possible.

```
# dis_constant_folding.py

1  #!/usr/bin/env python3
2  # encoding: utf-8
3
4  # Folded
5  i = 1 + 2
6  f = 3.4 * 5.6
7  s = 'Hello,' + ' World!'
8
9  # Not folded
10 I = i * 3 * 4
11 F = f / 2 / 3
12 S = s + '\n' + 'Fantastic!'
```

None of the values in the expressions on lines 5-7 can change the way the operation is performed, so the result of the expressions can be computed at compilation time and collapsed into single LOAD_CONST instructions. That is not true about lines 10-12. Because a variable is involved in those expressions, and the variable might refer to an object that overloads the operator involved, the evaluation has to be delayed to runtime.

```
$ python3 -m dis dis_constant_folding.py

  5           0 LOAD_CONST               0 (3)
              2 STORE_NAME               0 (i)

  6           4 LOAD_CONST               1 (19.04)
              6 STORE_NAME               1 (f)

  7           8 LOAD_CONST               2 ('Hello, World!')
             10 STORE_NAME               2 (s)

 10          12 LOAD_NAME                0 (i)
             14 LOAD_CONST               0 (3)
             16 BINARY_MULTIPLY
             18 LOAD_CONST               3 (4)
             20 BINARY_MULTIPLY
             22 STORE_NAME               3 (I)

 11          24 LOAD_NAME                1 (f)
             26 LOAD_CONST               4 (2)
             28 BINARY_TRUE_DIVIDE
             30 LOAD_CONST               0 (3)
             32 BINARY_TRUE_DIVIDE
             34 STORE_NAME               4 (F)

 12          36 LOAD_NAME                2 (s)
             38 LOAD_CONST               5 ('\n')
             40 BINARY_ADD
             42 LOAD_CONST               6 ('Fantastic!')
             44 BINARY_ADD
```

```
46 STORE_NAME               5 (S)
48 LOAD_CONST               7 (None)
50 RETURN_VALUE
```

See also

- Standard library documentation for dis – Includes the list of bytecode instructions.
- `Include/opcode.h` – The source code for the CPython interpreter defines the byte codes in `opcode.h`.
- *Python Essential Reference*, 4th Edition, David M. Beazley – http://www.informit.com/store/product.aspx?isbn=0672329786
- thomas.apestaart.org “Python Disassembly” – A short discussion of the difference between storing values in a dictionary between Python 2.5 and 2.6.
- Why is looping over range() in Python faster than using a while loop? – A discussion on StackOverflow.com comparing 2 looping examples via their disassembled bytecodes.
- Decorator for binding constants at compile time – Python Cookbook recipe by Raymond Hettinger and Skip Montanaro with a function decorator that re-writes the bytecodes for a function to insert global constants to avoid runtime name lookups.

abc — Abstract Base Classes
inspect — Inspect Live Objects

This page was last updated 2018-12-09.

Navigation
abc — Abstract Base Classes
inspect — Inspect Live Objects

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

Other Writing

- Blog
- The Python Standard Library By Example