mnfy Documentation

Release 33.0.0

Brett Cannon

January 05, 2016

Contents

1 Web Pages **3**

2 What the heck is mnfy for? **5**

3 Usage **7**
3.1 A note about version numbers and Python version compatibility 7
3.2 Command-line Usage 7

4 Transformations **9**
4.1 Source emission 9
4.2 Safe transformations 9
4.3 Sane transformations 11
4.4 Unsafe transformations 11

Contents

- *mnfy — minify/obfuscate Python 3 source code*
  - *Web Pages*
  - *What the heck is mnfy for?*
- *Usage*
  - *A note about version numbers and Python version compatibility*
  - *Command-line Usage*
- *Transformations*
  - *Source emission*
  - *Safe transformations*
    - *Combine imports*
    - *Combine with statements*
    - *Eliminate unused constants*
    - *Integer constants to power*
  - *Sane transformations*
  - *Unsafe transformations*
    - *Function to lambda*

CHAPTER 1

Web Pages

- Documentation
- Project site (issue tracker)
- PyPI/Cheeseshop

CHAPTER 2

What the heck is mnfy for?

The mnfy project was created for two reasons:

- To show that shipping bytecode files without source, as a form of obfuscation, is not the best option available
- Provide a minification of Python source code when total byte size of source code is paramount

When people ship Python code as only bytecode files (i.e. only `.pyo` files and no `.py` files), there are couple drawbacks. First and foremost, it prevents users from using your code with all available Python interpreters such as Jython and IronPython. Another drawback is that it is a poor form of obfuscation as projects like Meta allow you to take bytecode and reverse-engineer the original source code as enough details are kept that the only details missing are single-line comments.

When the total number of bytes used to ship Python code is paramount, then you want to minify the source code. Bytecode files actually contain so much detail that the space savings can be miniscule (e.g. the `decimal` module from Python's standard libary, which is the largest single file in the stdlib, has a bytecode file that is only 5% smaller than its original source code).

CHAPTER 3

Usage

3.1 A note about version numbers and Python version compatibility

The version number for mnfy is PEP 386 compliant, taking the form of `PPP.FFF.BBB`. The `FFF.BBB` represents the feature and bugfix version numbers of mnfy itself. The `PPP` portion of the version number represents the Python version that mnfy is compatible with: `'{}{}'.format(*sys.version_info[:2])`.

The Python version that mnfy is compatible with is directly embedded in the version number as Python's AST is not guaranteed to be backwards-compatible. This means that you should use each version of mnfy with a specific version of Python. Since mnfy works with source code and not bytecode you can safely use mnfy on code that must be backwards-compatible with older versions of Python, just make sure the interpreter you use with mnfy is correct and can parse the source code (e.g. just because the latest version of mnfy only works with Python 3.3 does not mean you cannot use that release against source code that must work with Python 3.2, just make sure to use a Python 3.3 interpreter with mnfy and that the source code can be read by a Python 3.3 interpreter).

3.2 Command-line Usage

TL;DR: pass the file you want to minify as an argument to mnfy and it will print to stdout the source code minified such that the AST is **exactly** the same as the original source code. To get transformations that will change the AST to varying degrees you will need to specificy various flags.

See the help message for the project for full instructions on usage:

```
python3 -m mnfy -h
python3 mnfy.py -h
```

CHAPTER 4

Transformations

4.1 Source emission

If you want no change to the AST compared to the original source code then you want mnfy's default behaviour of only emitting source code with not AST changes. Any tricks with source code formatting have been verified by passing Python's standard library through mnfy with only source emission used and comparing the result AST for no changes.

As an example of what source emission does, this code (32 characters):

```
if True:
  x = 5 + 2
  y = 9 - 1
```

becomes (19 characters):

```
if True:x=5+2;y=9-1
```

4.2 Safe transformations

For a transformation to be considered safe it must semantically equivalent to running the code as `python3 -OO` but can lead to a change in the AST. As the changes are semantically safe there is only a single option to turn on these transformations.

4.2.1 Combine imports

Take imports that are sequentially next to each other and put them on the same line **without** changing the import order.

From:

```
import X  # 8 characters
import Y  # 8 characters; 16 total
```

to:

```
import X,Y  # 10 characters
```

From:

```
from X import y  # 15 characters
from X import z  # 15 characters; 30 total
```

to:

```
from X import y,z  # 17 characters
```

4.2.2 Combine `with` statements

As of Python 3.2, contextlib.nested() is syntactically supported.

From:

```
with A:
 with B:pass
```

to:

```
with A,B:pass
```

4.2.3 Eliminate unused constants

If a constant isn't used then there is no need to keep it around. This primarily eliminates docstrings. If any block becomes completely empty then a `pass` statement is inserted.

From:

```
def bacon():
 """Docstring"""
```

to:

```
def bacon():pass
```

From:

```
if X:pass
else:4+2
```

to:

```
if X:pass
```

4.2.4 Integer constants to power

For sufficiently large integer constants, it saves space to use the power operator (`**`). Only numbers of base 2 and 10 are used as that is what the math module supports.

From:

```
4294967296
```

to:

```
2**32
```

4.3 Sane transformations

For typical code, sane transformations should be fine (e.g. you are not introspecting local variables). Since these transformations are typically safe you can turn them all on with a single option, but they can also be switched on individually as desired.

Note: Currently there are no sane transformations defined. See the issue tracker for some proposed transformations.

4.4 Unsafe transformations

For the more adventurous who know what features of Python their code relies on, unsafe transformations can be used. Just be very aware of what your code depends on before using any specific transformation. For this reason each unsafe transformation must be switched on individually.

4.4.1 Function to lambda

This is unsafe as lambda functions are not exactly like a function (e.g. lambda functions do not have a `__name__` attribute).

From:

```
def identity(x):return x  # 24 characters
```

to:

```
identity=lambda x:x  # 19 characters
```