Oneliner-izer

Chelsea Voss Anders Kaseorg

Massachusetts Institute of Technology

csvoss@mit.edu andersk@mit.edu

November 17, 2015

Motivation

Python Bee.

Motivation

Python Bee.
Write a function `f` that takes in a string `s` and returns `True` only if that string is composed of the characters `'A'` and `'a'`.

Motivation

Python Bee.
Write a function f that takes in a string s and returns True only if that string is composed of the characters 'A' and 'a'.

```
def f(s):
    for char in s:
        if char != 'a' and char != 'A':
            return False
    return True
```

Motivation

Python Bee.

Write a function `f` that takes in a string `s` and returns `True` only if that string is composed of the characters `'A'` and `'a'`.

```
def f(s):
    for char in s:
        if char != 'a' and char != 'A':
            return False
    return True
```

Can you solve this in one line?

Motivation

Python Bee.
Write a function `f` that takes in a string `s` and returns `True` only if that string is composed of the characters `'A'` and `'a'`.

```
def f(s):
    for char in s:
        if char != 'a' and char != 'A':
            return False
    return True
```

Can you solve this in one line?

```
f = lambda s: False not in [char in 'Aa' for char in s]
```

Motivation

```
f = lambda s: False not in [char in 'Aa' for char in s]
```

Motivation

```
f = lambda s: False not in [char in 'Aa' for char in s]
```

List comprehension.

```
>>> lst = [-2, -1, 0, 1, 2, 3, 4]
>>> [i * 10 for i in lst if i > 0]
[10, 20, 30, 40]
```

Motivation

```
f = lambda s: False not in [char in 'Aa' for char in s]
```

List comprehension.

```
>>> lst = [-2, -1, 0, 1, 2, 3, 4]
>>> [i * 10 for i in lst if i > 0]
[10, 20, 30, 40]
```

Lambda expression.

```
>>> f = lambda x: x + 2
>>> f(2)
4
```

Motivation

```
f = lambda s: False not in [char in 'Aa' for char in s]
```

List comprehension.

```
>>> lst = [-2, -1, 0, 1, 2, 3, 4]
>>> [i * 10 for i in lst if i > 0]
[10, 20, 30, 40]
```

Lambda expression.

```
>>> f = lambda x: x + 2
>>> f(2)
4
```

...Can we rewrite *any* Python code as a one-liner?

The Challenge

...Can we rewrite *any* Python code as a one-liner?

```
x = Class(47)
result = x.method()
print result
[...]
```

The Challenge

...Can we rewrite *any* Python code as a one-liner?

```
x = Class(47)
result = x.method()
print result
[...]
```

Well, yes.

The Challenge

...Can we rewrite *any* Python code as a one-liner?

```
x = Class(47)
result = x.method()
print result
[...]
```

Well, yes.

```
exec "x = Class(47)\nresult = x.method()\n[...]"
x = Class(47); result = x.method(); print result [...]
```

The Challenge

...Can we rewrite *any* Python code as a one-liner?

```
x = Class(47)
result = x.method()
print result
[...]
```

Well, yes.

```
exec "x = Class(47)\nresult = x.method()\n[...]"
x = Class(47); result = x.method(); print result [...]
```

But that's no fun.

The Rules

- One line: no newlines.
- No semicolons, either.
- No `exec`.
- No `eval`.

The Rules

- One line: no newlines.
- No semicolons, either.
- No `exec`.
- No `eval`.

Aim to be elegant! Make do with `lambda` and list comprehensions, and what other tricks may come, in attempt to implement the same functionality.

Overview

Simple Sequential Code

Determine how to convert a block of code (that is, multiple statements in sequence) into a single line.

```
a = 2 + 2
print a + a
```

Simple Sequential Code

Determine how to convert a block of code (that is, multiple statements in sequence) into a single line.

```
a = 2 + 2
print a + a
```

Answer.

Simple Sequential Code

Determine how to convert a block of code (that is, multiple statements in sequence) into a single line.

```
a = 2 + 2
print a + a
```

Answer.

```
print (lambda a: a + a)(2 + 2)
```

Simple Sequential Code

Determine how to convert a block of code (that is, multiple statements in sequence) into a single line.

```
a = 2 + 2
print a + a
```

Answer.

```
print (lambda a: a + a)(2 + 2)
print [a + a for a in [2 + 2]][0]
```

What about functions?

Determine how to allow function definitions in blocks of code:
How can you convert the following into a single line of code?

```
def f(x):
    return x * 10
print f(3)
```

What about functions?

Determine how to allow function definitions in blocks of code: How can you convert the following into a single line of code?

```
def f(x):
    return x * 10
print f(3)
```

Answer.

What about functions?

Determine how to allow function definitions in blocks of code: How can you convert the following into a single line of code?

```
def f(x):
    return x * 10
print f(3)
```

Answer.

```
print (lambda f: f(3))(lambda x: x * 10)
```

What about functions?

Determine how to allow function definitions in blocks of code:
How can you convert the following into a single line of code?

```
def f(x):
    return x * 10
print f(3)
```

Answer.

```
print (lambda f: f(3))(lambda x: x * 10)
```

Note that this works as-is with `*args` and `**kwargs`!

```
lambda x, y, *args, **kwargs: ...
```

What about operations that don't assign to a variable?

Determine how to allow statements which only act via side effects: they do not set a variable to a new value.

```
do_something()
print 42
```

What about operations that don't assign to a variable?

Determine how to allow statements which only act via side effects: they do not set a variable to a new value.

```
do_something()
print 42
```

Answer. Since the output value of `do_something()` isn't used, we can funnel it to the unused variable `_`.

What about operations that don't assign to a variable?

Determine how to allow statements which only act via side effects: they do not set a variable to a new value.

```
do_something()
print 42
```

Answer. Since the output value of `do_something()` isn't used, we can funnel it to the unused variable `_`.

```
print (lambda _: 42)(do_something())
```

What about operations that don't assign to a variable?

Determine how to allow statements which only act via side effects: they do not set a variable to a new value.

```
do_something()
print 42
```

Answer. Since the output value of `do_something()` isn't used, we can funnel it to the unused variable `_`.

```
print (lambda _: 42)(do_something())
```

Or:

```
print (do_something(), 42)[1]
```

Putting it all together

```
x = 2 + 2
def f(x):
    return x * 5
__print(x)
y = f(x)
...
```

Putting it all together

```
x = 2 + 2
def f(x):
    return x * 5
__print(x)
y = f(x)
...
```

```
(lambda y: ...)(f(x))
```

Putting it all together

```
x = 2 + 2
def f(x):
    return x * 5
__print(x)
y = f(x)
...
```

```
(lambda _:
     (lambda y: ...)(f(x))
)(__print(x))
```

Putting it all together

```
x = 2 + 2
def f(x):
    return x * 5
__print(x)
y = f(x)
...
```

```
(lambda f:
     (lambda _:
          (lambda y: ...)(f(x))
     )(__print(x))
)(lambda x: x * 5)
```

Putting it all together

```
x = 2 + 2
def f(x):
    return x * 5
__print(x)
y = f(x)
...
```

```
(lambda x:
    (lambda f:
        (lambda _:
            (lambda y: ...)(f(x))
        )(__print(x))
    )(lambda x: x * 5)
)(2 + 2)
```

if/else Statements

How can we make this code one line?

```
if True:
    x = 5
else:
    x = 10
print x * 100
```

`if`/`else` Statements

How can we make this code one line?

```
if True:
    x = 5
else:
    x = 10
print x * 100
```

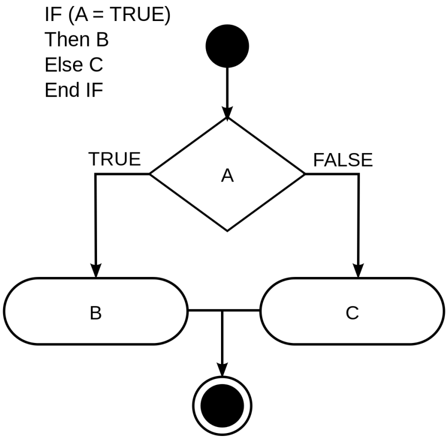
Answer. Conditional expressions (`B if A else C`), plus continuation passing.

if/else Statements

How can we make this code one line?

```
if True:
    x = 5
else:
    x = 10
print x * 100
```

Answer. Conditional expressions (`B if A else C`), plus continuation passing.



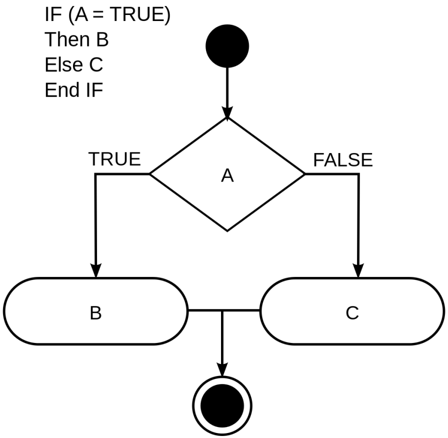
if/else Statements

```
def continuation(x):
    print x * 100
```

if/else Statements

```
def continuation(x):
    print x * 100

if True:
    x = 5
    continuation(x)
else:
    x = 10
    continuation(x)
```



if/else Statements

```
def continuation(x):
    print x * 100

if True:
    x = 5
    continuation(x)
else:
    x = 10
    continuation(x)
```

if/else Statements

```
(lambda continuation:
```

```
def continuation(x):
    print x * 100

if True:
    x = 5
    continuation(x)
else:
    x = 10
    continuation(x)
```

if/else Statements

```
def continuation(x):
    print x * 100

if True:
    x = 5
    continuation(x)
else:
    x = 10
    continuation(x)
```

```
(lambda continuation:
  (lambda x:
    continuation(x)
  )(5)
  if True else
  (lambda x:
    continuation(x)
  )(10)
)
```

if/else Statements

```
def continuation(x):
    print x * 100

if True:
    x = 5
    continuation(x)
else:
    x = 10
    continuation(x)
```

```
(lambda continuation:
  (lambda x:
    continuation(x)
  )(5)
  if True else
  (lambda x:
    continuation(x)
  )(10)
)(lambda x:
  __print(x * 100)
)
```

while Loops

How can we make this code one line?

```
x = 5
while x < 20:
   x = x + 4
print x
```

while Loops

How can we make this code one line?

```
x = 5
while x < 20:
   x = x + 4
print x
```

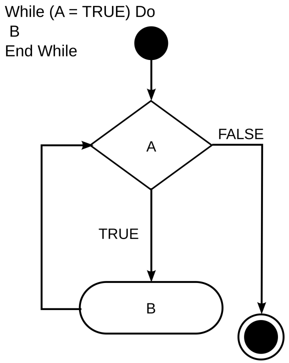
Answer. Conditional expressions and continuation passing... again!

while Loops

How can we make this code one line?

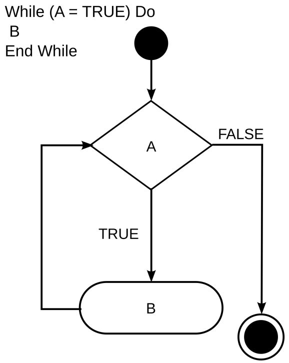
```
x = 5
while x < 20:
   x = x + 4
print x
```

Answer. Conditional expressions and continuation passing... again!



while Loops

```
x = 5
def while_loop(x):
    if x < 20:
        x = x + 4
        while_loop(x)
    else:
        print x
while_loop(x)
```



while Loops

```
x = 5
def while_loop(x):
    if x < 20:
        x = x + 4
        while_loop(x)
    else:
        print x
while_loop(x)
```

Problem: `while_loop` is recursive!

while Loops

```
x = 5
def while_loop(x):
    if x < 20:
        x = x + 4
        while_loop(x)
    else:
        print x
while_loop(x)
```

Problem: `while_loop` is recursive!
Solution: **Y combinator.**

`while` Loops

```
x = 5
def while_loop(x):
    if x < 20:
        x = x + 4
        while_loop(x)
    else:
        print x
while_loop(x)
```

Problem: `while_loop` is recursive!
Solution: **Y combinator.**

```
Y =
(lambda f: (lambda x: x(x))
(lambda y: f(lambda: y(y)())))
```

`while` Loops

```
x = 5
def while_loop(x):
    if x < 20:
        x = x + 4
        while_loop(x)
    else:
        print x
while_loop(x)
```

Problem: `while_loop` is recursive!
Solution: **Y combinator.**

```
Y =
(lambda f: (lambda x: x(x))
(lambda y: f(lambda: y(y)())))
```

```
(lambda x: (lambda while_loop: while_loop(x))
(Y(lambda while_loop: (lambda x: (lambda x:
while_loop(x))(x+4) if x<20 else __print(x)))))(5)
```

Storing state

Old way:

```
(lambda x: ...)(42)
```

Storing state

Old way:

```
(lambda x: ...)(42)
```

New way:

```
[... for __l['x'] in [42]]
```

for Loops

```
total = 0
for item in iterable:
    total += item
    print total
...
```

for Loops

```
total = 0
for item in iterable:
    total += item
    print total
...
```

Reminder.
The items of `iterable` must be consumed one-by-one in order. We can't index into it with `iterable[i]`.

```
>>> iterable = {10, 20, 30}
>>> for item in iterable:
...     print item
...
10
20
30
>>> iterable[2]
TypeError: 'set' object
does not support indexing
```

for Loops

```
total = 0
for item in iterable:
    total += item
    print total
...
```

Partial answer.

for Loops

```
total = 0
for item in iterable:
    total += item
    print total
...
```

Partial answer.
Use the `reduce` builtin.
(No easy way to `break` or `return`.)

```
(lambda total:
  ...
)(reduce(
  lambda total, item:
    (lambda total:
      (__print(total), total)[1]
    )(total + item),
  {10, 20, 30},
  0
))
```

for Loops

```
total = 0
for item in iterable:
    total += item
    print total
...
```

Answer.

for Loops

```
total = 0
for item in iterable:
    total += item
    print total
...
```

Answer.
Convert to a while loop that consumes the iterable using next.

```
total = 0
items = iter(iterable)
sentinel = []
while True:
    item = next(items, sentinel)
    if item is sentinel:
        break
    total += item
    print total
...
```

Imports

```
import random as rnd
print rnd.choice([1, 2, 3, 10])
```

Imports

```
import random as rnd
print rnd.choice([1, 2, 3, 10])
```

Answer. This is equivalent to:

```
rnd = __import__('random')
print rnd.choice([1, 2, 3, 10])
```

Fortunately, `__import__` itself doesn't need to be imported.

Functional Print

How can we get the `__print()` function that we've been using?

Functional Print

How can we get the `__print()` function that we've been using?

Answer. In Python 2, usually we can use `from __future__ import print_function`. However, that's not a real import statement, it's a compiler directive.

Functional Print

How can we get the `__print()` function that we've been using?

Answer. In Python 2, usually we can use `from __future__ import print_function`. However, that's not a real import statement, it's a compiler directive.

Instead:

```
__builtin__ = __import__('__builtin__')
__print = __builtin__.__dict__['print']
```

Classes

```
class Person(object):
    def __init__(self):
        ...
```

Classes

```
class Person(object):
    def __init__(self):
        ...
```

$\rightarrow$

```
type('Person', (object,), {'__init__': lambda self: ...})
```

Raising Errors

- `raise Bad()`

Raising Errors

- `raise Bad()`

  $\rightarrow$

  `([] for [] in []).throw(Bad())`

Raising Errors

- `raise Bad()`

  $\rightarrow$

  `([] for [] in []).throw(Bad())`
- `assert good`

  `...`

Raising Errors

- `raise Bad()`

  →

  ```
  ([] for [] in []).throw(Bad())
  ```

- `assert good`

  ```
  ...
  ```

  →

  ```
  ... if good else ([] for [] in []).throw(
    AssertionError)
  ```

try/except

Problem:

```
try:
    foo()
except Bad as ev:
    bar(ev)
```

try/except

Problem:

```
try:
    foo()
except Bad as ev:
    bar(ev)
```

Solution: abuse the context manager protocol!

try/except

Solution: abuse the context manager protocol!

```
class Handler:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        if et is not None and issubclass(et, Bad):
            bar(ev); return True
        return False
with Handler():
    foo()
```

try/except

```
class Body:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        foo()
class Handler:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        if et is not None and issubclass(et, Bad):
            bar(ev); return True
        return False
with Handler(), Body():
    pass
```

try/except

```
class Body:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        foo()
class Handler:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        if et is not None and issubclass(et, Bad):
            bar(ev); return True
        return False
with contextlib.nested(Handler(), Body()):
    pass
```

try/except

```
class Body:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        foo()  # Why __exit__?  Python issue 5251.
class Handler:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        if et is not None and issubclass(et, Bad):
            bar(ev); return True
        return False
with contextlib.nested(Handler(), Body()):
    pass
```

try/except

```
class Body:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        foo()  # Why __exit__?  Python issue 5251.
class Handler:
    def __enter__(self): pass
    def __exit__(self, et, ev, tb):
        if et is not None and issubclass(et, Bad):
            bar(ev); return True
        return False
ctx = contextlib.nested(Handler(), Body())
ctx.__enter__()
ctx.__exit__(None, None, None)
```

try/except

```
Body = type('Body', (), {
  '__enter__': lambda self: None,
  '__exit__': lambda self, et, ev, tb: foo()
})()
Handler = type('Handler', (), {
  '__enter__': lambda self: None,
  '__exit__': lambda self, et, ev, tb:
    et is not None and issubclass(et, Bad) and
    (bar(ev), True)[1]
})()
ctx = contextlib.nested(Handler(), Body())
ctx.__enter__()
ctx.__exit__(None, None, None)
```

try/except

```
(lambda ctx:
  (ctx.__enter__(), ctx.__exit__(None, None, None))
)(contextlib.nested(
  type('Handler', (), {
    '__enter__': lambda self: None,
    '__exit__': lambda self, et, ev, tb:
      et is not None and issubclass(et, Bad) and
      (bar(ev), True)[1]
  })(), type('Body', (), {
    '__enter__': lambda self: None,
    '__exit__': lambda self, et, ev, tb: foo()
  })()
))
```

What's Left

- `from module import *`
- `yield` and generators
- `with`

Building the Compiler

- `ast` - for parsing Python files into syntax trees

Building the Compiler

- `ast` - for parsing Python files into syntax trees
- `argparse` - for parsing command-line arguments

Building the Compiler

- `ast` - for parsing Python files into syntax trees
- `argparse` - for parsing command-line arguments
- `symtable` - for determining the scope of variables

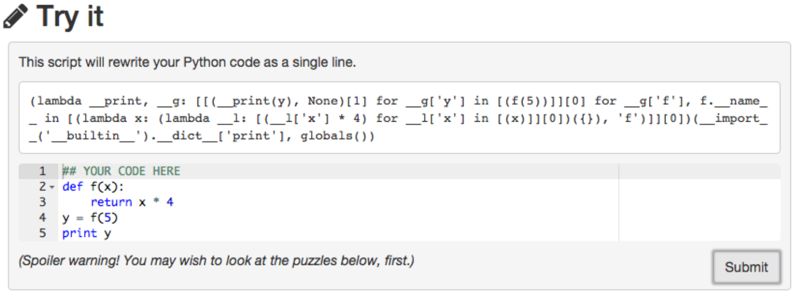
Building the Compiler

- `ast` - for parsing Python files into syntax trees
- `argparse` - for parsing command-line arguments
- `symtable` - for determining the scope of variables
- `unittest` - test suite

Oneliner-izing the Oneliner-izer

```
(lambda __print, __g, __contextlib, __y, __exec: [(('Convert any Python file into a single line of code.\n\nUsage via the command line:\n\n$ python main.py --help\n    print usages\n$ python main.py infile
.py\n    one-line infile.py, put the result in infile.ol.py\n$ python main.py infile.py outfile.py\n    one-line infile.py, put the result in outfile.py\n', [[[[(lambda __mod: [[[[[[[[[[(lambda __after:
[[(parser.add_argument('infile', nargs='?'), (parser.add_argument('outfile', nargs='?'), (parser.add_argument('--debug', action='store_true'), [[(lambda __after: [[__after() for __g['outfilename'] in [(No
ne)]][0] for __g['original'] in [(sys.stdin.read())]][0] if (args.infile is None) else (lambda __after: (lambda __after: [__after() for __g['outfilename'] in [('_ol.py'.join(args.infile.rsplit('.py', 1)))
]][0] if args.infile.endswith('.py') else [__after() for __g['outfilename'] in [((args.infile + '_ol.py'))]][0])(lambda: __after()) if (args.outfile is None) else [__after() for __g['outfilename'] in [(ar
gs.outfile)]][0])(lambda: __after()))(lambda: (lambda __after: [[(infile.close(), __after())[1] for __g['original'] in [(infile.read().strip())]][0] for __g['infile'] in [(open(args.infile))]][0] if (orig
inal is None) else __after())(lambda: [(lambda __after: (__print(onelined), __after())[1] if (outfilename is None) else [(outfi.write((onelined + '\n')), (outfi.close(), __after())[1])[1] for __g['outfi']
 in [(open(outfilename, 'w'))]][0])(lambda: (lambda __after: (lambda __after: [__after() for sys.stdout in [(sys.stderr)]][0] if (outfilename is None) else __after())(lambda: (__print('--- ORIGINAL ------
---------------------------'), (__print(original), (__print('------------------------------------------------'), [(lambda __out: (lambda __ctx: [__ctx.__enter__(), __ctx.__exit__(None, None, None), __out[0]
(lambda: (__print('--- ONELINED ---------------------------------'), (__print(onelined), (__print('------------------------------------------------'), [(lambda __out: (lambda __ctx: [__ctx.__enter__(), __ct
x.__exit__(None, None, None), __out[0](lambda: __after())][2])(__contextlib.nested(type('except', (), {'__enter__': lambda self: None, '__exit__': lambda __self, __exctype, __value, __traceback: __exctype
 is not None and (issubclass(__exctype, Exception) and [[True for __out[0] in [(lambda __after: (__print(e), __after())[1])]][0] for __g['e'] in [(__value)]][0])})(), type('try', (), {'__enter__': lambda
self: None, '__exit__': lambda __self, __exctype, __value, __traceback: [False for __out[0] in [((__exec((onelined, scope), __g, __g), (lambda __after: __after()))[1])]][0]})())))([None]) for __g['scope']
 in [({})]][0])[1])[1])[1])][2])(__contextlib.nested(type('except', (), {'__enter__': lambda self: None, '__exit__': lambda __self, __exctype, __value, __traceback: __exctype is not None and (issubclass(_
_exctype, Exception) and [[True for __out[0] in [(lambda __after: (__print(e), __after())[1])]][0] for __g['e'] in [(__value)]][0])})(), type('try', (), {'__enter__': lambda self: None, '__exit__': lambda
 __self, __exctype, __value, __traceback: [False for __out[0] in [((__exec((original, scope), __g, __g), (lambda __after: __after()))[1])]][0]})())))([None]) for __g['scope'] in [({})]][0])[1])[1])[1]) if
 args.debug else __after())(lambda: __after())) for __g['onelined'] in [(to_one_line(original))]][0])) for __g['original'] in [(None)]][0] for __g['args'] in [(parser.parse_args())]][0])[1])[1])[1] for __
g['parser'] in [(argparse.ArgumentParser(usage='\n       '.join(usage), description='if infile is given and outfile is not, outfile will be infile_ol.py'))]][0] for __g['usage'] in [(['python main.py --he
lp', 'python main.py [--debug] [infile.py [outfile.py]]'])]][0] if (__name__ == '__main__') else __after())(lambda: None) for __g['to_one_line'], to_one_line.__name__ in [(lambda original: (lambda __l: [[
[[(lambda __after: __l['original'] if ((len(__l['original'].splitlines()) == 1) and (len(__l['t'].body) == 1) and (type(__l['t'].body[0]) in (ast.Delete, ast.Assign, ast.AugAssign, ast.Print, ast.Raise, a
st.Assert, ast.Import, ast.ImportFrom, ast.Exec, ast.Global, ast.Expr, ast.Pass))) else __after())(lambda: get_init_code(__l['t'], __l['table'])) for __l['original'] in [(__l['original'].strip())]][0] for
 __l['table'] in [(symtable.symtable(__l['original'], '<string>', 'exec'))]][0] for __l['t'] in [(ast.parse(__l['original']))]][0] for __l['original'] in [(original)]][0])({}), 'to_one_line')]][0] for __g
['Namespace'] in [((lambda b, d: d.get('__metaclass__', getattr(b[0], '__class__', type(b[0]))))('Namespace', b, d))((ast.NodeVisitor,), (lambda __l: ('\n    AST visitor.\n    ', [[[[[[[[[[[[[[[[[[[
[[[[[[[[[[[[[[[[[[[[[[[[[[[[__l for __l['generic_visit'], __l['generic_visit'].__name__ in [(lambda self, tree: (lambda __l: [([] for [] in []).throw(NotImplementedError(('Case not caught: %s'
 % str(type(__l['tree']))))) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'generic_visit')]][0] for __l['visit_Yield'], __l['visit_Yield'].__name__ in [(lambda self, tree: (lambda __l: [([] fo
r [] in []).throw(NotImplementedError('Open problem: yield')) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Yield')]][0] for __l['visit_With'], __l['visit_With'].__name__ in [(lambda sel
f, tree: (lambda __l: [([] for [] in []).throw(NotImplementedError('Open problem: with')) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_With')]][0] for __l['visit_While'], __l['visit_Whi
le'].__name__ in [(lambda self, tree: (lambda __l: [[[[lambda_function({'__after': T('lambda: {after}')}).format(T('{__y}(lambda __this: lambda: {} if {} else {})()').format(provide(__l['body'], __break='
__after', __continue='__this'), __l['test'], __l['orelse'])) for __l['orelse'] in [(__l['self'].many_to_one(__l['tree'].orelse, after='__after()'))]][0] for __l['body'] in [(__l['self'].many_to_one(__l['t
ree'].body, after='__this()'))]][0] for __l['test'] in [(__l['self'].visit(__l['tree'].test))]][0] for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_While')]][0] for __l['visit_UnaryOp'], __
l['visit_UnaryOp'].__name__ in [(lambda self, tree: (lambda __l: [T('({}{})').format(unaryop_code[type(__l['tree'].op)], __l['self'].visit(__l['tree'].operand)) for __l['self'], __l['tree'] in [(self, tre
e)]][0])({}), 'visit_UnaryOp')]][0] for __l['visit_Tuple'], __l['visit_Tuple'].__name__ in [(lambda self, tree: (lambda __l: [T('({})').format((T(', ').join(map(__l['self'].visit, __l['tree'].elts)) + (',
' * (len(__l['tree'].elts) == 1)))) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Tuple')]][0] for __l['visit_TryFinally'], __l['visit_TryFinally'].__name__ in [(lambda self, tree: (lamb
da __l: [([] for [] in []).throw(NotImplementedError('Open problem: try-finally')) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_TryFinally')]][0] for __l['visit_TryExcept'], __l['visit_
TryExcept'].__name__ in [(lambda self, tree: (lambda __l: [[[(lambda __items, __after, __sentinel: __y(lambda __this: lambda: (lambda __i: [(lambda __after: [__after() for __l['code'] in [(T('{body}'))]][
0] if (__l['handler'].type is None) else [__after() for __l['code'] in [(T('issubclass(__exctype, {type}) and {body}').format(type=__l['self'].visit(__l['handler'].type)))]][0])(lambda: (lambda __after: [
__after() for __l['code'] in [(__l['code'].format(body=assignment_component(T('{body}'), __l['self'].visit(__l['handler'].name), '__value')))]][0] if (__l['handler'].name is not None) else __after())(lamb
da: (__l['handlers'].append(__l['code'].format(body=assignment_component('True', '__out[0]', ('lambda __after: ' + __l['self'].many_to_one(__l['handler'].body, after='__after()'))))), __this())[1])) for _
_l['handler'] in [(__i)]][0] if __i is not __sentinel else __after())(next(__items, __sentinel)))())(iter(__l['tree'].handlers), lambda: lambda_function({'__out': '[None]'}).format(lambda_function({'__ctx
': T("{__contextlib}.nested(type('except', (), {{'__enter__': lambda self: None, '__exit__': lambda __self, __exctype, __value, __traceback: __exctype is not None and ({handlers})}}))(), type('try', (), {{
'__enter__': lambda self: None, '__exit__': lambda __self, __exctype, __value, __traceback: {body}}})())").format(body=assignment_component('False', '__out[0]', __l['body']), handlers=T(' or ').join(__l['
handlers']))}).format(T('[__ctx.__enter__(), __ctx.__exit__(None, None, None), __out[0](lambda: {after})][2]')), []) for __l['handlers'] in [([])]][0] for __l['body'] in [(__l['self'].many_to_one(__l['tr
ee'].body, after=T('(lambda __after: {orelse})').format(orelse=__l['self'].many_to_one(__l['tree'].orelse, after='__after()'), pre_return=T('(lambda ret: lambda after: ret)({pre_return}'), post_return=T(
'{post_return})')))]][0] for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_TryExcept')]][0] for __l['visit_Subscript'], __l['visit_Subscript'].__name__ in [(lambda self, tree: (lambda __l: [
T('{}[{}]').format(__l['self'].visit(__l['tree'].value), __l['self'].visit(__l['tree'].slice)) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Subscript')]][0] for __l['visit_Str'], __l['v
isit_Str'].__name__ in [(lambda self, tree: (lambda __l: [T('{!r}').format(__l['tree'].s) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Str')]][0] for __l['visit_Slice'], __l['visit_Slic
e'].__name__ in [(lambda self, tree: (lambda __l: [T('{}:{}{}').format(('' if (__l['tree'].lower is None) else __l['self'].visit(__l['tree'].lower)), ('' if (__l['tree'].upper is None) else __l['self'].vi
sit(__l['tree'].upper)), ('' if (__l['tree'].step is None) else (':' + __l['self'].visit(__l['tree'].step)))) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Slice')]][0] for __l['visit_Se
tComp'], __l['visit_SetComp'].__name__ in [(lambda self, tree: (lambda __l: [__l['self'].comprehension_code(__l['tree'].generators, (lambda ns, g: (lambda ___f_tree, __l: [T('{{{} {}}}').format(__l['ns'].
visit(___f_tree().elt), __l['g']) for __l['ns'], __l['g'] in [(ns, g)]][0])(lambda: __l['tree'], {}))) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_SetComp')]][0] for __l['visit_Set'],
 __l['visit_Set'].__name__ in [(lambda self, tree: (lambda __l: [(T('{{{}}}').format(T(', ').join((lambda __l, __iter, ___f_self: (___f_self().visit(__l['elt']) for __l['elt'] in __iter))({}, __l['tree'].e
lts, lambda: __l['self']))) if __l['tree'].elts else ([] for [] in []).throw(AssertionError('{} is a dict'))) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Set')]][0] for __l['visit_Retu
rn'], __l['visit_Return'].__name__ in [(lambda self, tree: (lambda __l: [T('{pre_return}{}{post_return}').format(__l['self'].visit(__l['tree'].value)) for __l['self'], __l['tree'] in [(self, tree)]][0])({
}), 'visit_Return')]][0] for __l['visit_Repr'], __l['visit_Repr'].__name__ in [(lambda self, tree: (lambda __l: [T('`{}`').format(__l['self'].visit(__l['tree'].value)) for __l['self'], __l['tree'] in [(se
lf, tree)]][0])({}), 'visit_Repr')]][0] for __l['visit_Raise'], __l['visit_Raise'].__name__ in [(lambda self, tree: (lambda __l: [(lambda __after: T('([] for [] in []).throw(*{__sys}.exc_info())') if (__l
['tree'].type is None) else T('([] for [] in []).throw({}{}{})').format(__l['self'].visit(__l['tree'].type), ('' if (__l['tree'].inst is None) else (', ' + __l['self'].visit(__l['tree'].inst))), ('' if (_
_l['tree'].tback is None) else (', ' + __l['self'].visit(__l['tree'].tback)))))(lambda: None) for __l['self'], __l['tree'] in [(self, tree)]][0])({}), 'visit_Raise')]][0] for __l['visit_Print'], __l['visi
t_Print'].__name__ in [(lambda self, tree: (lambda __l: [[(lambda __after: [__after() for __l['to_print'] in [(T('file={}, *[{}]').format(__l['self'].visit(__l['tree'].dest), __l['to_print']))]][0] if (__
l['tree'].dest is not None) else __after())(lambda: [(lambda __after: [__after() for __l['to_print'] in [((lambda o, v: (lambda r: o + v if r is NotImplemented else r)(object.__getattribute__(object.__get
attribute__(type(o), '__dict__').get('__iadd__', lambda self, other: NotImplemented), '__get__')(o, type(o))(v)))(__l['to_print'], ", end=' '"))]][0] if (not __l['tree'].nl) else __after())(lambda: T('({_
```

Links



This script will rewrite your Python code as a single line.

```
(lambda __print, __g: [[(__print(y), None)[1] for __g['y'] in [(f(5))]][0] for __g['f'], f.__name__ in [(lambda x: (lambda __l: [(__l['x'] * 4) for __l['x'] in [(x)]][0])({}), 'f')]][0])(__import__('__builtin__').__dict__['print'], globals())
```

```
## YOUR CODE HERE
def f(x):
    return x * 4
y = f(5)
print y
```

(Spoiler warning! You may wish to look at the puzzles below, first.)

- Demo: `http://onelinepy.herokuapp.com/`
- Code: `https://github.com/csvoss/onelinerizer`