

To fix this, add a check to the **simple_eval** function. In this case, the **tokenize** module uses a special token to indicate that it has reached the end of the source, so all you have to do is to grab the next token and make sure it's the expected end marker:

```
def simple_eval(source):
    src = cStringIO.StringIO(source).readline
    src = tokenize.generate_tokens(src)
    res = atom(src.next, src.next())
    if src.next()[0] is not tokenize.ENDMARKER:
        raise SyntaxError("bogus data after expression")
    return res
```

With this in place, you'll get a nice little exception:

```
>>> simple_eval("'hello'"))
Traceback (most recent call last):
  File "", line 1, in ?
    File "test.py", line 27, in simple_eval
        raise SyntaxError("bogus data after expression")
SyntaxError: bogus data after expression

>>> simple_eval("'hello'")
'hello'
```

Here's another buglet (there is at least one more, but I'll return to that one later): the code uses **cStringIO** to split the source string into individual lines, but the parser chokes on line breaks:

```
>>> simple_eval("(1, 2, 3, \n4711.0)")
Traceback (most recent call last):
  File "test.py", line 33, in ?
    print repr(simple_eval("(1, 2, 3, \n4711.0)"))
  File "test.py", line 27, in simple_eval
    res = atom(src.next, src.next())
  File "test.py", line 8, in atom
    out.append(atom(next, token))
  File "test.py", line 21, in atom
    raise SyntaxError("malformed expression (%s)" % token[1])
SyntaxError: malformed expression (
)
```

It's not obvious, but the erroneous token is a newline character. Changing **(%s)** to **(%r)** in the error format string makes it easier to spot the problem:

```
>>> simple_eval("(1, 2, 3, \n4711.0)")
Traceback (most recent call last):
  File "test.py", line 33, in ?
    print repr(simple_eval("(1, 2, 3, \n4711.0)"))
  File "test.py", line 27, in simple_eval
    res = atom(src.next, src.next())
  File "test.py", line 8, in atom
    out.append(atom(next, token))
  File "test.py", line 21, in atom
    raise SyntaxError("malformed expression (%r)" % token[1])
SyntaxError: malformed expression ('\n')
```

To fix this bug, the code needs to ignore newline characters. The easiest way to do this is to add a filter to the token stream. A generator expression comes in handy:


```
def simple_eval(source):
    src = cStringIO.StringIO(source).readline
    src = tokenize.generate_tokens(src)
    src = (token for token in src if token[0] is not tokenize.NL)
    res = atom(src.next, src.next())
    if src.next()[0] is not tokenize.ENDMARKER:
        raise SyntaxError("bogus data after expression")
    return res
```

The generator expression will simply skip all **NL** tokens, so the rest of the code don't really have to bother.

```
>>> simple_eval("(1, 2, 3, \n4711.0)")
(1, 2, 3, 4711.0)
```

Generator expressions were added in Python 2.4. In Python 2.3, you can use **itertools.filter** instead.

Note that the use of stacked generators results in lazy parsing of the source file; the parser will call the expression's **next** method to get the next non-NL token, and the expression will call the tokenizer's **next** method until it gets one.

 rendered by a [django](#) application. hosted by [webfaction](#).