Simple Iterator-based Parsing

Fredrik Lundh | November 2005 | Originally posted to online.effbot.org

Iterator-based parsing is an efficient and straightforward way of writing recursive-descent parsers in Python. Here's an outline:

- 1. use an iterator to split the source into a stream of tokens or token descriptors.
- 2. pass the iterator's **next** method and the first token to the toplevel parser class.
- 3. use separate functions, where appropriate, for individual grammar rules. pass the **next** method and the current token on to these functions as well.
- 4. to check the current token, inspect the token argument. to fetch the next token, call the **next** method.

Here's a simple example. This is a limited but hopefully safe version of Python's **eval** function, which handles strings, floating point values, integers, and tuples only. Tuples can be nested.

```
import cStringIO, tokenize
def atom(next, token):
    if token[1] == "(":
        out = []
        token = next()
        while token[1] != ")":
            out.append(atom(next, token))
            token = next()
            if token[1] == ",":
                token = next()
        return tuple(out)
    elif token[0] is tokenize.STRING:
        return token[1][1:-1].decode("string-escape")
    elif token[0] is tokenize.NUMBER:
           return int(token[1], 0)
        except ValueError:
            return float(token[1])
    raise SyntaxError ("malformed expression (%s)" % token[1])
def simple eval(source):
    src = cStringIO.StringIO(source).readline
    src = tokenize.generate tokens(src)
    return atom(src.next, src.next())
```

The **simple_eval** function uses the **generate_tokens** function in the standard **tokenize** module to produce a stream of tokens (for each token, this function returns a tuple containing a token code, the token value, start and end offsets, and the source line (as returned by **readline**)).

The **atom** function implements a simplified version of the *atom* rule in the Python grammar. In this version, it handles tuples, numbers, and strings with inline code, and calls itself to extract the tuple members.

Here's the code in action:

```
>>> simple_eval("'hello'")
'hello'
>>> simple_eval("(1, 2, 3, 4711.0)")
(1, 2, 3, 4711.0)
>>> simple_eval("('spam',123,('spam','egg'),0x10,(3.14,))")
('spam', 123, ('spam', 'egg'), 16, (3.140000000000001,))
```

This first version doesn't check if anything's left in the token stream before it returns the result, so valid expressions that happens to be followed by junk characters will parse just fine:

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

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 erronous 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.ifilter** 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.

A django SITE rendered by a django application. hosted by webfaction.