Simple Parser

How to easily write parsers in Python

Author: Christophe Delord
Contact: christophe.delord@free.fr
Web site: http://christophe.delord.free.fr/sp

Date: Sunday 24 January 2010

Table of Contents

1 Introduction and tutorial

1.1 Introduction

SP (Simple Parser) is a Python[1] parser generator. It is aimed at easy usage rather than performance. SP produces Top-Down Recursive descent parsers. SP also uses memoization to optimize parsers' speed when dealing with ambiguous grammars.

License

SP is available under the GNU Lesser General Public:

```
Simple Parser: A Python parser generator

Copyright (C) 2009 Christophe Delord

Simple Parser is free software: you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published
by the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

Simple Parser is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with Simple Parser.  If not, see <http://www.gnu.org/licenses/>.
```

Structure of the document

Introduction and tutorial starts smoothly with a gentle tutorial as an introduction. I think this tutorial may be sufficent to start with SP.

SP reference is a reference documentation. It will detail SP as much as possible.

Some examples to illustrate SP gives the reader some examples to illustrate SP.

1.2 Installation

Getting SP

SP is freely available on its web page (http://christophe.delord.free.fr/sp).

Requirements

SP is a *pure Python* package. It may run on *any platform* supported by Python. The only requirement of SP is *Python 2.6* or newer[2]. Python can be downloaded at http://www.python.org.

1.3 Tutorial

Introduction

This short tutorial presents how to make a simple calculator. The calculator will compute basic mathematical expressions (`+`, `-`, `*`, `|`) possibly nested in parenthesis. We assume the reader is familiar with regular expressions.

[1] Python is a wonderful object oriented programming language available at http://www.python.org

[2] Older *Python* versions may work (tested with Python 2.4 and 2.5). See the Older Python versions chapter.

Defining the grammar

Expressions are defined with a grammar. For example an expression is a sum of terms and a term is a product of factors. A factor is either a number or a complete expression in parenthesis.

We describe such grammars with rules. A rule describes the composition of an item of the language. In our grammar we have 3 items (expr, term, factor). We will call these items *symbols* or *non terminal symbols*. The decomposition of a symbol is symbolized with `->`.

Grammar for expressions:

Grammar rule	**Description**
`expr -> term (('+'|'-') term)*`	An expression is a term eventually followed with a plus (`+`) or a minus (`-`) sign and an other term any number of times (`*` is a repetition of an expression 0 or more times).
`term -> fact (('*'|'/') fact)*`	A term is a factor eventually followed with a `*` or `/` sign and an other factor any number of times.
`fact -> ('+'|'-') fact | number | '(' expr ')'`	A factor is either a factor precedeed by a sign, a number or an expression in parenthesis.

We have defined here the grammar rules (i.e. the sentences of the language). We now need to describe the lexical items (i.e. the words of the language). These words - also called *terminal symbols* - are described using regular expressions. In the rules we have written some of these terminal symbols (`+`, `-`, `*`, `/`, `(`, `)`). We have to define `number`. For sake of simplicity numbers are integers composed of digits (the corresponding regular expression can be `[0-9]+`). To simplify the grammar and then the Python script we define two terminal symbols to group the operators (additive and multiplicative operators). We can also define a special symbol that is ignored by SP. This symbol is used as a separator. This is generaly useful for white spaces and comments.

Terminal symbol definition for expressions:

Terminal symbol	**Regular expression**	**Comment**
`number`	`[0-9]+ or \d+`	One or more digits
`addop`	`[+-]`	a `+` or a `-`
`mulop`	`[*/]`	a `*` or a `/`
`spaces`	`\s+`	One or more spaces

This is sufficient to define our parser with SP.

Grammar of the expression recognizer:

```
def Calc():

    number = R(r'[0-9]+')
    addop = R('[+-]')
    mulop = R('[*/]')

    with Separator(r'\s+'):

        expr = Rule()
        fact = Rule()
        fact |= addop & fact
        fact |= '(' & expr & ')'
```

```
        fact |= number
        term = fact & ( mulop & fact )[:]
        expr |= term & ( addop & term )[:]

    return expr
```

Calc is the name of the Python function that returns a parser. This function returns expr which is the *axiom*[3] of the grammer.

expr and fact are recursive rules. They are first declared as empty rules (expr = Rule()) and alternatives are later added (expr |= ...).

Slices are used to implement repetitions. foo[:] parses foo zero or more times, which is equivalent to foo* is a classical grammar notation.

The grammar can also be defined with the mini grammar language provided by SP:

```
def Calc():
    return compile("""
        number = r'[0-9]+' ;
        addop = r'[+-]' ;
        mulop = r'[*/]' ;

        separator: r'\s+' ;

        !expr = term (addop term)* ;
        term = fact (mulop fact)* ;
        fact = addop fact ;
        fact = '(' expr ')' ;
        fact = number ;
    """)
```

Here the *axiom*[3] is identified by !.

With this small grammar we can only recognize a correct expression. We will see in the next sections how to read the actual expression and to compute its value.

Reading the input and returning values

The input of the grammar is a string. To do something useful we need to read this string in order to transform it into an expected result.

This string can be read by catching the return value of terminal symbols. By default any terminal symbol returns a string containing the current token. So the token '(' always returns the string '('. For some tokens it may be useful to compute a Python object from the token. For example number should return an integer instead of a string, addop and mulop, followed by a number, should return a function corresponding to the operator. That's why we will add a function to the token and rule definitions. So we associate int to number and op1 and op2 to unary and binary operators.

int is a Python function converting objects to integers and op1 and op2 are user defined functions.

op1 and op2 functions:

```
op1 = lambda f,x: {'+':pos, '-':neg}[f](x)
op2 = lambda f,y: lambda x: {'+': add, '-
': sub, '*': mul, '/': div}[f](x,y)

# red applyies functions to a number
```

[3] The axiom is the symbol from which the parsing starts

```
def red(x, fs):
    for f in fs: x = f(x)
    return x
```

To associate a function to a token or a rule it must be applyed using / or * operators:

- / applyies a function to an object returned by a (sub)parser.
- * applyies a function to an tuple of objects returned by a sequence of (sub) parsers.

Token and rule definitions with functions:

```
number = R(r'[0-9]+') / int

fact |= (addop & fact) * op1
term = (fact & ( (mulop & fact) * op2 )[:]) * red

# R(r'[0-9]+') applyed on "42" will return "42".
# R(r'[0-9]+') / int will return int("42")

# addop & fact applyied on "+ 42" will return ('+', 42)
# (addop & fact) * op1 will return op1(*('+', 42)), i.e. op1('+', 42)
# so (addop & fact) * op1 returns +42

# (addop & fact) * op2 will return op2(*('+', 42)), i.e. op2('+', 42)
# so (addop & fact) * op2 returns lambda x: add(x, 42)

# fact & ( (mulop & fact) * op2 )[:] returns a number and a list of func-
tions
# for instance (42, [(lambda x:mul(x, 43)), (lambda x:mul(x, 44))])
# so (fact & ( (mulop & fact) * op2 )[:]) * red applyied on "42*43*44"
# will return red(42, [(lambda x:mul(x, 43)), (lambda x:mul(x, 44))])
# i.e. 42*43*44
```

And with the SP language:

```
number = r'[0-9]+' : `int` ;

addop = r'[+-]' ;
mulop = r'[*/]' ;

fact = addop fact :: `op1` ;
term = fact (mulop fact :: `op2`)* :: `red` ;

# r'[0-9]+' applyed on "42" will return "42".
# r'[0-9]+' : `int` will return int("42")

# "addop fact" applyied on "+ 42" will return ('+', 42)
# "addop fact :: `op1`" will return op1(*('+', 42)), i.e. op1('+', 42)
# so "addop fact :: `op1`" returns +42

# "addop fact :: `op2`" will return op2(*('+', 42)), i.e. op2('+', 42)
# so "addop fact :: `op2`" returns lambda x: add(x, 42)

# "fact (mulop fact :: `op2`)*" returns a number and a list of functions
```

```
# for instance (42, [(lambda x:mul(x, 43)), (lambda x:mul(x, 44))])
# so "fact (mulop fact :: ‘op2‘)* :: ‘red‘" applyied on "42*43*44"
# will return red(42, [(lambda x:mul(x, 43)), (lambda x:mul(x, 44))])
# i.e. 42*43*44
```

In the SP language, `:` (as `/`) applies a Python function (more generally a callable object) to a value returned by a sequence and `::` (as `*`) applies a Python function to several values returned by a sequence.

Here is finally the complete parser.

Expression recognizer and evaluator:

```
from sp import *

def Calc():

    from operator import pos, neg, add, sub, mul, truediv as div

    op1 = lambda f,x: {'+':pos, '-':neg}[f](x)
    op2 = lambda f,y: lambda x: {'+': add, '-
': sub, '*': mul, '/': div}[f](x,y)

    def red(x, fs):
        for f in fs: x = f(x)
        return x

    number = R(r'[0-9]+') / int
    addop = R('[+-]')
    mulop = R('[*/]')

    with Separator(r'\s+'):

        expr = Rule()
        fact = Rule()
        fact |= (addop & fact) * op1
        fact |= '(' & expr & ')'
        fact |= number
        term = (fact & ( (mulop & fact) * op2 )[:]) * red
        expr |= (term & ( (addop & term) * op2 )[:]) * red

    return expr
```

Or with SP language:

```
from sp import *

def Calc():

    from operator import pos, neg, add, sub, mul, truediv as div

    op1 = lambda f,x: {'+':pos, '-':neg}[f](x)
    op2 = lambda f,y: lambda x: {'+': add, '-
': sub, '*': mul, '/': div}[f](x,y)

    def red(x, fs):
        for f in fs: x = f(x)
```

```
        return x

    return compile("""
        number = r'[0-9]+' : `int` ;
        addop = r'[+-]' ;
        mulop = r'[*/]' ;

        separator: r'\s+' ;

        !expr = term (addop term :: `op2`)* :: `red` ;
        term = fact (mulop fact :: `op2`)* :: `red` ;
        fact = addop fact :: `op1` ;
        fact = '(' expr ')' ;
        fact = number ;
    """)
```

Embeding the parser in a script

A parser is a simple Python object. This example show how to write a function that returns a parser. The parser can be applyied to strings by simply calling the parser.

Writting SP grammars in Python:

```
from sp import *

def MyParser():

    parser = ...

    return parser

# You can instanciate your parser here
my_parser = MyParser()

# and use it
parsed_object = my_parser(string_to_be_parsed)
```

To use this parser you now just need to instanciate an object.

Complete Python script with expression parser:

```
from sp import *

def Calc():

    ...

calc = Calc()
while True:
    expr = input('Enter an expression: ')
    try: print(expr, '=', calc(expr))
    except Exception as e: print("%s:"%e.__class__.__name__, e)
```

Conclusion

This tutorial shows some of the possibilities of SP. If you have read it carefully you may be able to start with SP. The next chapters present SP more precisely. They contain more examples to illustrate all the features of SP.

Happy SP'ing!

2 SP reference

2.1 Usage

SP is a package which main function is to provide basic objects to build a complete parser.

The grammar is a Python object.

Grammar embeding example:

```
def Foo():
    bar = R('bar')
    return bar
```

Then you can use the new generated parser. The parser is simply a Python object.

Parser usage example:

```
test = "bar"
my_parser = Foo()
x = my_parser(test)              # Parses "bar"
print x
```

2.2 Grammar structure

SP grammars are Python objects. SP grammars may contain two parts:

Tokens are built by the `R` or `K` keywords.

Rules are described after tokens in a `Separator` context.

Example of SP grammar structure:

```
def Foo():

    # Tokens
    number = R(r'\d+') / int

    # Rules
    with Separator(r'\s+'):
        S = number[:]

    return S

foo = Foo()
result = foo("42 43 44") # return [42, 43, 44]
```

2.3 Lexer

Regular expression syntax

The lexer is based on the *re*[4] module. SP profits from the power of Python regular expressions. This document assumes the reader is familiar with regular expressions.

You can use the syntax of regular expressions as expected by the *re*[5] module.

Predefined tokens

Tokens can be explicitely defined by the `R`, `K` and `Separator` keywords.

Expression	Usage
`R`	defines a regular token. The token is defined with a regular expression and returns a string (or a tuple of strings if the regular expression defines groups).
`K`	defines a token that returns nothing (useful for keywords for instance). The keyword is defined by an identifier (in this case word boundaries are expected around the keyword) or another string (in this case the pattern is not considered as a regular expression). The token just recognizes a keyword and returns nothing.
`Separator`	is a context manager used to define separators for the rules defined in the context. The token is defined with a regular expression and returns nothing.

A token can be defined by:

a name which identifies the token. This name is used by the parser.

a regular expression which describes what to match to recognize the token.

an action which can translate the matched text into a Python object. It can be a function of one argument or a non callable object. If it is not callable, it will be returned for each token otherwise it will be applied to the text of the token and the result will be returned. This action is optional. By default the token text is returned.

Token definition examples:

```
integer = R(r'\d+') / int
identifier = R(r'[a-zA-Z]\w*\b')
boolean = R(r'(True|False)\b') / (lambda b: b=='True')

spaces = K(r'\s+')
comments = K(r'#.*')

with Separator(spaces|comments):
    # rules defined here will use spaces and comments as separators
    atom = '(' & expr & ')'
```

There are two kinds of tokens. Tokens defined by the `R` or `K` keywords are parsed by the parser and tokens defined by the `Separator` keyword are considered as separators (white spaces or comments for example) and are wiped out by the lexer.

The word boundary `\b` can be used to avoid recognizing “True” at the beginning of “Truexyz”.

If the regular expression defines groups, the parser returns a tuple containing these groups:

```
couple = R('<(\d+)-(\d+)>')

couple("<42-43>") == ('42', '43')
```

[4] *re* is a standard Python module. It handles regular expressions. For further information about *re* you can read http://docs.python.org/lib/module-re.html

[5] Read the Python documentation for further information: http://docs.python.org/lib/re-syntax.html

If the regular expression defines only one group, the parser returns the value of this group:

```
first = R('<(\d+)-\d+>')

first("<42-43>") == '42'
```

Unwanted groups can be avoided using `(?:...)`.

A name can be given to a token to make error messages easier to read:

```
couple = R('<(\d+)-(\d+)>', name="couple")
```

Regular expressions can be compiled using specific compilation options. Options are defined in the `re` module:

```
token = R('...', flags=re.IGNORECASE|re.DOTALL)
```

`re` defines the following flags:

I (IGNORECASE) Perform case-insensitive matching.

L (LOCALE) Make `\w`, `\W`, `\b`, `\B`, dependent on the current locale.

M (MULTILINE) `"^"` matches the beginning of lines (after a newline) as well as the string. `"$"` matches the end of lines (before a newline) as well as the end of the string.

S (DOTALL) `"."` matches any character at all, including the newline.

X (VERBOSE) Ignore whitespace and comments for nicer looking RE's.

U (UNICODE) Make `\w`, `\W`, `\b`, `\B`, dependent on the Unicode locale

Inline tokens

Tokens can also be defined on the fly. Their definition are then inlined in the grammar rules. This feature may be useful for keywords or punctuation signs.

In this case tokens can be written without the `R` or `K` keywords. They are considered as keywords (as defined by `K`).

Inline token definition examples:

```
IfThenElse = 'if' & Cond &
             'then' & Statement &
             'else' & Statement
```

2.4 Parser

Declaration

A parser is declared as a Python object.

Grammar rules

Rule declarations have two parts. The left side declares the symbol associated to the rule. The right side describes the decomposition of the rule. Both parts of the declaration are separated with an equal sign (`=`).

Rule declaration example:

```
SYMBOL = (A & B) * (lambda a, b: f(a, b))
```

Sequences

Sequences in grammar rules describe in which order symbols should appear in the input string. For example the sequence `A & B` recognizes an `A` followed by a `B`.

For example to say that a `sum` is a `term` plus another `term` you can write:

```
Sum = Term & '+' & Term
```

Alternatives

Alternatives in grammar rules describe several possible decompositions of a symbol. The infix pipe operator (`|`) is used to separate alternatives. `A | B` recognizes either an `A` or a `B`. If both `A` and `B` can be matched only the first longest match is considered. So the order of alternatives may be very important when two alternatives can match texts of the same size.

For example to say that an `atom` is an *integer* or an *expression in paranthesis* you can write:

```
Atom = integer | '(' & Expr & ')'
```

Repetitions

Repetitions in grammar rules describe how many times an expression should be matched.

Expression	Usage
`A[:1]`	recognizes zero or one `A`.
`A[:]`	recognizes zero or more `A`.
`A[1:]`	recognizes one or more `A`.
`A[m:n]`	recognizes at least m and at most n `A`.
`A[m:n:s]`	recognizes at least m and at most n `A` using `s` as a separator.

Repetitions are greedy. Repetitions are implemented as Python loops. Thus whatever the length of the repetitions, the Python stack will not overflow.

The separator is useful to parse lists. For instance a comma separated parameter list is `parameter[::',']`.

Precedence and grouping

The following table lists the different structures in increasing precedence order. To override the default precedence you can group expressions with parenthesis.

Precedence in SP expressions:

Structure	Example
Alternative	`A | B`
Sequence	`A & B`
Repetitions	`A[x:y]`
Symbol and grouping	`A` and `( ... )`

Actions

Grammar rules can contain actions as Python functions.

Functions are applyied to parsed objects using `/` or `*`.

Expression	Value
`parser / function`	returns *function(result of parser)*.
`parser * function`	returns *function(*result of parser)*.

`*` can be used to analyse the result of a sequence.

Abstract syntax trees

An abstract syntax tree (AST) is an abstract representation of the structure of the input. A node of an AST is a Python object (there is no constraint about its class). AST nodes are completely defined by the user.

AST example (parsing a couple):

```
class Couple:
    def __init__(self, a, b):
        self.a = a
        self.b = b

def Foo():
    couple = ('(' & item & ',' & item & ')') * Couple
    return couple
```

Constants

It is sometimes useful to return a constant. `C` defines a parser that matches an empty input and returns a constant.

Constant example:

```
number = (  '1' & C("one")
         |  '2' & C("two")
         |  '3' & C("three")
         )
```

Position in the input string

To know the current position in the input string, the `At()` parser returns an object containing the current index (attribute `index`) and the corresponding line and column numbers (attributes `line` and `column`):

```
position = At() / `lambda p: (p.line, p.column)`
rule = ... & pos & ...
```

2.5 Performances and memory consumption

Backtracking has a cost. The parser may often try to parse again the same string at the same position. To improve the speed of the parser, some time consumming functions are *memoized*. This drastically fasten the parser but requires more memory. If a lot of string are parsed in a single script this mechanism can slow down the computer because of heavy swap disk usage or even lead to a memory error.

To avoid such problems it is recommanded to clean the memoization cache by calling the `sp.clean` function:

```
import sp

...

for s in a_lot_of_strings:
    parse(s)
    sp.clean()
```

3 Older Python versions

This document describes the usage of SP with Python 2.6. Grammars need some adaptations to work with Python 2.5. or older.

3.1 Separators

Separators use context managers which don't exist in Python 2.4. Context managers have been introduced in Python 2.5 (`from __future__ import with_statement`) and in Python 2.6 (as a standard feature). When the context managers are not available, it may be possible to call the `__enter__` and `__exit__` method explicitly (tested for Python 2.4).

Python 2.6 and later:

```
number = R(r'\d+') / int
with Separator('\s+'):
    coord = number & ',' & number
```

Python 2.5 with `with_statement`:

```
from __future__ import with_statement

number = R(r'\d+') / int
with Separator('\s+'):
    coord = number & ',' & number
```

Python 2.5 or 2.4 (or older but not tested) without `with_statement`:

```
sep = Separator('\s+')

number = R(r'\d+') / int
sep.__enter__()
coord = number & ',' & number
sep.__exit__()
```

4 SP mini language

Instead of using Python expressions that can sometimes be difficult to read, it's possible to write grammars in a cleaner syntax and compile these grammar with the `sp.compile` function. This function takes the grammar as a string parameter. The `sp.compile_file` function reads the grammar in a separate file.

Here the equivalence between Python expressions and the SP mini language:

<table>
<tr><th>SP Python expressions</th><th>SP mini language</th><th>Description</th></tr>
<tr><td>R("regular expression")
R("regexpr", name="name")</td><td>r"regular expression"
name.r"regexpr"</td><td>Token defined by a regular expression</td></tr>
<tr><td>K("plain text")
K("plain text", name="name")</td><td>"plain text"
name."plain text"</td><td>Keyword defined by a non interpreted string</td></tr>
<tr><td>t = R('...', flags=re.I|re.S)</td><td>lexer: I S; t = r'...'</td><td>Regular expression options</td></tr>
<tr><td>with Separator(...):</td><td>separator: ... ;</td><td>Separator definition</td></tr>
<tr><td>C(object)</td><td>`object`</td><td>Parses nothing and returns object</td></tr>
<tr><td>... / function</td><td>... : `function`</td><td>Parses ... and apply the result to function (function(...))</td></tr>
<tr><td>... * function</td><td>... :: `function`</td><td>Parses ... and apply the result (multiple values) to function (function(*...))</td></tr>
<tr><td>... & At() & ...</td><td>... @ ...</td><td>Position in the input string</td></tr>
<tr><td>(...)[:]</td><td>(...)*</td><td>Zero or more matches</td></tr>
<tr><td>(...)[1:]</td><td>(...)+</td><td>One or more matches</td></tr>
<tr><td>(...)[:1]</td><td>(...)?</td><td>Zero or one matche</td></tr>
<tr><td>(...)[::S]</td><td>[.../S]*</td><td>Zero or more matches separated by S</td></tr>
<tr><td>(...)[1::S]</td><td>[.../S]+</td><td>One or more matches separated by S</td></tr>
<tr><td>A & B & C</td><td>A B C</td><td>Sequence</td></tr>
<tr><td>A | B | C</td><td>A | B | C</td><td>Alternative</td></tr>
<tr><td>(...)</td><td>(...)</td><td>Grouping</td></tr>
<tr><td>rule_name = ...</td><td>rule_name = ... ;</td><td>Rule definition</td></tr>
<tr><td>axiom_name = ...</td><td>!axiom_name = ... ;</td><td>Axiom definition</td></tr>
</table>

5 Some examples to illustrate SP

5.1 Complete interactive calculator

This chapter presents an extention of the calculator described in the tutorial. This calculator has more functions and a memory.

The grammar has been rewritten using the SP language.

New functions

The calculator has memories. A memory cell is identified by a name. For example, if the user types `pi = 3.14`, the memory cell named `pi` will contain the value of `pi` and `2*pi` will return `6.28`.

The variables are saved in a dictionnary.

Source code

Here is the complete source code (*calc.py*):

```
#!/usr/bin/env python

# Simple Parser
# Copyright (C) 2009 Christophe Delord
# http://christophe.delord.free.fr/sp

# This file is part of Simple Parser.
#
# Simple Parser is free software: you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as published
# by the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# Simple Parser is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with Simple Parser.  If not, see <http://www.gnu.org/licenses/>.

""" Calc

num  : generic numerical calculus   | assignment: name = expression
int  : integral calculus            | binary    : b... or ...b
int8 : integral calculus on 8 bits  | octal     : o... or ...o
int16: integral calculus on 16 bits | hexa      : h... or ...h or 0x...
int32: integral calculus on 32 bits | operators : + -
| ^ * % / & >> << ~ **
int64: integral calculus on 64 bits | functions : rev factor sqrt
flt32: 32 bit float calculus        |
flt64: 64 bit float calculus        |
rat  : rational calculus            | this help : ?
"""

import struct
import sys

import sp
from fractions import Fraction

try:
    import readline
except ImportError:
    pass

class Calc:
    def __init__(self):
        self.number = Num
        self.memory = {}
```

```
    def __getitem__(self, var): return self.memory[var.name]
    def __setitem__(self, var, val): self.memory[var.name] = val

class Help:
    def __init__(self):
        self.doc = __doc__.strip()
    def eval(self, calc=None):
        return self.doc

class Num:
    name, descr = "num", "Number"
    def __init__(self, val):
        if isinstance(val, Num): self.val = val.val
        else: self.val = val
    def __int__(self):      return int(self.val)
    def __float__(self):    return float(self.val)
    def __str__(self):      return str(self.val)
    def __add__(x, y):      return x.__class__(x.val + y.val)
    def __sub__(x, y):      return x.__class__(x.val - y.val)
    def __or__(x, y):       return x.__class__(x.val | y.val)
    def __xor__(x, y):      return x.__class__(x.val ^ y.val)
    def __mul__(x, y):      return x.__class__(x.val * y.val)
    def __mod__(x, y):      return x.__class__(x.val % y.val)
    def __truediv__(x, y):  return x.__class__(x.val / y.val)
    def __and__(x, y):      return x.__class__(x.val & y.val)
    def __rshift__(x, y):   return x.__class__(x.val >> y.val)
    def __lshift__(x, y):   return x.__class__(x.val << y.val)
    def __pos__(x):         return x.__class__(+x.val)
    def __neg__(x):         return x.__class__(-x.val)
    def __invert__(x):      return x.__class__(~x.val)
    def __pow__(x, y):      return x.__class__(x.val ** y.val)
    def rev(self):
        raise TypeError("%s can not be bit re-
versed"%self.__class__.__name__)
    def factor(self):
        n = int(self.val)
        if n != self.val: raise TypeError("%s is not integer"%self.val)
        if n < 0: ds = [-1]; n = -n
        else: ds = []
        rn = n**0.5
        while n > 1 and n%2==0: ds.append(2); n //= 2
        d = 3
        while d <= rn:
            while n%d==0: ds.append(d); n //= d
            d += 2
        if n > 1: ds.append(n)
        return " ".join(map(str, ds))
    def sqrt(self):
        return self.__class__(self.val ** (1/2))

class Float(Num):
    name, descr = "flt32", "32 bit Float"
    def __init__(self, val): self.val = float(val)
```

```
    def __str__(self): return """%s
    ieee: 0x%08X"""%(   self.val,
        ieee_int32(self.val),
    )

class Double(Num):
    name, descr = "flt64", "64 bit Float"
    def __init__(self, val): self.val = float(val)
    def __str__(self): return """%s
    ieee: 0x%16X"""%(   self.val,
                        ieee_int64(self.val),
    )

class Rat(Num):
    name, descr = "rat", "Rational"
    def __init__(self, val):
        if isinstance(val, float):
            self.val = Frac-
tion("%.53f"%(val)).limit_denominator(1000000000)
        elif isinstance(val, Num): self.val = val.val
        else: self.val = Fraction(val)
    def __int__(self): return self.val.numerator//self.val.denominator
    def __float__(self): return self.val.numerator/self.val.denominator
    def __str__(self): return str(self.val)

class Int(Num):
    name, descr = "int", "Integer"
    def __init__(self, val): self.val = int(val)
    def __truediv__(x, y): return x.__class__(x.val // y.val)
    def __str__(self): return base(self.val, radix=10, group=3, width=None)

class Int8(Num):
    name, descr = "int8", "8 bit Integer"
    width = 8
    def __init__(self, val): self.val = int(val) & (2**self.width-1)
    def __truediv__(x, y): return x.__class__(x.val // y.val)
    def __str__(self): return """%s
    hex: %s
    oct: %s
    bin: %s"""%(    base(self.val, radix=10, group=3, width=None),
                    base(self.val, radix=16, group=4, width=self.width),
                    base(self.val, radix=8,  group=3, width=self.width),
                    base(self.val, radix=2,  group=4, width=self.width),
    )
    def rev(self):
        """ reverse bit order """
        return self.__class__(
            sum(    ((self.val>>i)&0x1)<<(self.width-1-i)
                    for i in range(self.width)
            )
        )

class Int16(Int8):
```

```
    name, descr = "int16", "16 bit Integer"
    width = 16
    def __str__(self): return """%s
    hex: %s
    bin: %s"""%(    base(self.val, radix=10, group=3, width=None),
                    base(self.val, radix=16, group=4, width=self.width),
                    base(self.val, radix=2,  group=4, width=self.width),
    )

class Int32(Int8):
    name, descr = "int32", "32 bit Integer"
    width = 32
    def __str__(self): return """%s
    hex: %s
    bin: %s
    flt: %s"""%(    base(self.val, radix=10, group=3, width=None),
                    base(self.val, radix=16, group=4, width=self.width),
                    base(self.val, radix=2,  group=4, width=self.width),
                    ieee_float(self.val),
    )

class Int64(Int32):
    name, descr = "int64", "64 bit Integer"
    width = 64
    def __str__(self): return """%s
    hex: %s
    bin: %s
    flt: %s"""%(    base(self.val, radix=10, group=3, width=None),
                    base(self.val, radix=16, group=4, width=self.width),
                    base(self.val, radix=2,  group=4, width=self.width),
                    ieee_double(self.val),
    )

def ieee_int32(x):
    return struct.unpack("I", struct.pack("f", x))[0]

def ieee_int64(x):
    return struct.unpack("Q", struct.pack("d", x))[0]

def ieee_float(n):
    return struct.unpack("f", struct.pack("I", n))[0]

def ieee_double(n):
    return struct.unpack("d", struct.pack("Q", n))[0]

def base(N, radix=10, group=3, width=None):
    if width:
        N %= 2**width
        bits_per_digit = {16:4, 8:3, 2:1}[radix]
        min_len = width//bits_per_digit
    if N < 0:
        N = -N
        sign = "-"
```

```
    else:
        sign = ""
    s = ""
    while N:
        N, d = divmod(N, radix)
        s = s + "0123456789ABCDEF"[d]
    s = s or "0"
    if width:
        s = s + "0"*(min_len-len(s))
    s = " ".join(s[i:i+group] for i in range(0, len(s), group))
    return sign + s[::-1]

class Bin:
    def __init__(self, n): self.val = int(n.replace('_', ''), 2)
    def eval(self, calc): return calc.number(self.val)

class Oct:
    def __init__(self, n): self.val = int(n.replace('_', ''), 8)
    def eval(self, calc): return calc.number(self.val)

class Hex:
    def __init__(self, n): self.val = int(n.replace('_', ''), 16)
    def eval(self, calc): return calc.number(self.val)

class Dec:
    def __init__(self, n): self.val = int(n.replace('_', ''), 10)
    def eval(self, calc): return calc.number(self.val)

class Real:
    def __init__(self, n): self.val = float(n.replace('_', ''))
    def eval(self, calc): return calc.number(self.val)

class Var:
    def __init__(self, n): self.name = n
    def eval(self, calc):
        try: val = calc[self]
        except KeyError: raise NameError(self.name)
        return val.eval(calc)

class Mode:
    def __init__(self, mode): self.mode = mode
    def eval(self, calc):
        calc.number = self.mode
        return "%s mode"%calc.number.descr

class Assign:
    def __init__(self, name, expr): self.name, self.expr = name, expr
    def eval(self, calc):
        calc[self.name] = self.expr
        return self.expr.eval(calc)

class Op2:
    def __init__(self, x, y): self.x, self.y = x, y
```

```
class Op1:
    def __init__(self, x): self.x = x

class Add(Op2):
    def eval(self, calc): return self.x.eval(calc) + self.y.eval(calc)

class Sub(Op2):
    def eval(self, calc): return self.x.eval(calc) - self.y.eval(calc)

class Or(Op2):
    def eval(self, calc): return self.x.eval(calc) | self.y.eval(calc)

class Xor(Op2):
    def eval(self, calc): return self.x.eval(calc) ^ self.y.eval(calc)

class Mul(Op2):
    def eval(self, calc): return self.x.eval(calc) * self.y.eval(calc)

class Mod(Op2):
    def eval(self, calc): return self.x.eval(calc) % self.y.eval(calc)

class Div(Op2):
    def eval(self, calc): return self.x.eval(calc) / self.y.eval(calc)

class And(Op2):
    def eval(self, calc): return self.x.eval(calc) & self.y.eval(calc)

class RShift(Op2):
    def eval(self, calc): return self.x.eval(calc) >> self.y.eval(calc)

class LShift(Op2):
    def eval(self, calc): return self.x.eval(calc) << self.y.eval(calc)

class Pow(Op2):
    def eval(self, calc): return self.x.eval(calc) ** self.y.eval(calc)

class Pos(Op1):
    def eval(self, calc): return +self.x.eval(calc)

class Neg(Op1):
    def eval(self, calc): return -self.x.eval(calc)

class Inv(Op1):
    def eval(self, calc): return ~self.x.eval(calc)

class Rev(Op1):
    def eval(self, calc): return self.x.eval(calc).rev()

class Factor(Op1):
    def eval(self, calc): return self.x.eval(calc).factor()

class Sqrt(Op1):
```

```
    def eval(self, calc): return self.x.eval(calc).sqrt()

def red(x, ys):
    for f, y in ys:
        x = f(x, y)
    return x

def red1(f, x):
    return f(x)

parser = sp.compile(
    r"""
        bin = r'b([_0-1]+)\b' : `Bin` ;
        bin = r'([_0-1]+)b\b' : `Bin` ;
        oct = r'o([_0-7]+)\b' : `Oct` ;
        oct = r'([_0-7]+)o\b' : `Oct` ;
        hex = r'h([_0-9a-fA-F]+)\b' : `Hex` ;
        hex = r'([_0-9a-fA-F]+)h\b' : `Hex` ;
        hex = r'0x([_0-9a-fA-F]+)\b' : `Hex` ;
        real = r'(?:\d+\.\d*|\d*\.\d+)(?:[eE][-+]?\d+)?|\d+[eE][-
+]?\d+' : `Real` ;
        dec = r'\d+' : `Dec` ;
        var = r'[a-zA-Z_]\w*' : `Var`;

        addop = '+' `Add` ;
        addop = '-' `Sub` ;
        addop = '|' `Or` ;
        addop = '^' `Xor` ;

        mulop = '*' `Mul` ;
        mulop = '%' `Mod` ;
        mulop = '/' `Div` ;
        mulop = '&' `And` ;
        mulop = '>>' `RShift` ;
        mulop = '<<' `LShift` ;

        unop = '+' `Pos` ;
        unop = '-' `Neg` ;
        unop = '~' `Inv` ;

        powop = '**' `Pow` ;

        separator: r'\s+' ;

        !S = '?'        `Help()`;

        !S = 'num'      `Mode(Num)`;
        !S = 'int8'     `Mode(Int8)`;
        !S = 'int16'    `Mode(Int16)`;
        !S = 'int32'    `Mode(Int32)`;
        !S = 'int64'    `Mode(Int64)`;
        !S = 'int'      `Mode(Int)`;
        !S = 'flt32'    `Mode(Float)`;
```

```
        !S = 'flt64'    `Mode(Double)`;
        !S = 'rat'      `Mode(Rat)`;

        !S = var '=' expr :: `Assign` ;
        !S = expr ;

        expr = term (addop term)* :: `red` ;
        term = fact (mulop fact)* :: `red` ;
        fact = unop fact :: `red1` | pow ;
        pow = atom (powop fact)? :: `red` ;

        atom = '(' expr ')' ;
        atom = 'rev' '(' expr ')' : `Rev` ;
        atom = 'factor' '(' expr ')' : `Factor` ;
        atom = 'sqrt' '(' expr ')' : `Sqrt` ;
        atom = bin | oct | hex | real | dec | var ;
    """)

if __name__ == '__main__':
    print(Help().eval())
    print()
    calc = Calc()
    while True:
        expr = input("(%s) "%calc.number.name)
        if not expr: continue
        try:
            val = parser(expr).eval(calc)
        except Exception as e:
            print("%s: %s"%(e.__class__.__name__, e))
        else:
            print("=", val)
        print()
```