Simple Parser
or
How to easily write parsers in Python

Christophe Delord
christophe.delord@free.fr
http://christophe.delord.free.fr/sp/

July 12, 2009

Contents

List of Figures

Part I

Introduction and tutorial

Chapter 1

Introduction

1.1 Introduction

SP (Simple Parser) is a Python[1] parser generator. It is aimed at easy usage rather than performance.

1.2 License

SP is available under the GNU Lesser General Public.

1.3 Structure of the document

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

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

Part III gives the reader some examples to illustrate SP.

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

Chapter 2

Installation

2.1 Getting SP

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

2.2 Requirements

SP is a *pure Python* package. It may run on *any platform* supported by Python. The only requirement of SP is *Python 3.0* or newer. Python can be downloaded at http://www.python.org.

Chapter 3

Tutorial

3.1 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.

3.2 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 describe 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 $\rightarrow$. The grammar of this tutorial is given in figure 3.1.

Figure 3.1: Grammar for expressions

Grammar rule	Description
$expr \rightarrow term\ (('+'\|'-')\ term)*$	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).
$term \rightarrow fact\ (('*'\|'/')\ fact)*$	A term is a factor eventually followed with a $'*'$ or $'/'$ sign and an other factor any number of times.
$Fact \rightarrow ('+'\|'-')\ fact\|\ number\ \|\ '('\ Expr\ ')'$	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 usefull for white spaces and comments. The terminal symbols are given in figure 3.2

Figure 3.2: Terminal symbol definition for expressions

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

This is sufficient to define our parser with SP. The grammar of the expressions in SP can be found in figure 3.3.

Figure 3.3: Grammar of the expression recognizer

```
def Calc():

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

    with Separator(r'\s+'):

        expr = Rule()
        fact = Rule()
        fact |= addop & fact
        fact |= Drop(r'\(') & expr & Drop(r'\)')
        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*[1] of the grammar.

expr and *fact* are recursive rules. They are first declared as empty rules ($expr = Rule()$) and alternatives are later added ($expr \mid= ...$).

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

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.

3.3 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.

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

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 (figure 3.4).

Figure 3.4: *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
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 as in figure 3.5 / 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.

Figure 3.5: Token and rule definitions with functions

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

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

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

Finally the complete parser is given in figure 3.6.

Figure 3.6: 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 = Token(r'[0-9]+') / int
    addop = Token('[+-]')
    mulop = Token('[*/]')

    with Separator(r'\s+'):

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

    return expr
```

3.4 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 just by calling the parser (see figure 3.7).

To use this parser you now just need to instanciate an object as in figure 3.8.

Figure 3.7: 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)
```

Figure 3.8: Complete Python script with expression parser

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

    with Separator(r'\s+'):

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

    return expr

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

3.5 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!

Part II

SP reference

Chapter 4

Usage

4.1 Package content

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

The grammar is a Python object (see figure 4.1).

Figure 4.1: Grammar embeding example

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

Then you can use the new generated parser. The parser is simply a Python object (see figure 4.2).

Figure 4.2: Parser usage example

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

Chapter 5

Grammar structure

5.1 SP grammar structure

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

Tokens are built by the *Token* or *Drop* keyword (see 6.2).

Rules are described after tokens (see 5.1) in a *Separator context*.

See figure 5.1 for a generic SP grammar.

Figure 5.1: SP grammar structure

```
def Foo:

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

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

    return S

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

5.2 Comments

Comments in SP start with # and run until the end of the line, as in Python.

```
# This is a comment
```

Chapter 6

Lexer

6.1 Regular expression syntax

The lexer is based on the *re*[1] 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* module.

Here is a summary[2] of the regular expression syntax:

"." (Dot.) In the default mode, this matches any character except a newline. If the *DOTALL* flag has been specified, this matches any character including a newline.

"ˆ" (Caret.) Matches the start of the string, and in *MULTILINE* mode also matches immediately after each newline.

"$" Matches the end of the string or just before the newline at the end of the string, and in *MULTILINE* mode also matches before a newline. foo matches both 'foo' and 'foobar', while the regular expression $foo\$$ matches only 'foo'. More interestingly, searching for $foo.\$$ in 'foo1\nfoo2\n' matches 'foo2' normally, but 'foo1' in *MULTILINE* mode.

"*" Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. $ab*$ will match 'a', 'ab', or 'a' followed by any number of 'b's.

"+" Causes the resulting RE to match 1 or more repetitions of the preceding RE. $ab+$ will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

"?" Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. $ab?$ will match either 'a' or 'ab'.

***?, +?, ??** The "*", "+", and "?" qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE $< .* >$ is matched against '<H1>title</H1>', it will match the entire string, and not just '<H1>'. Adding "?" after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using $.*?$ in the previous expression will match only '<H1>'.

{m} Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, $a\{6\}$ will match exactly six "a" characters, but not five.

[1] *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

[2] From the Python documentation : http://docs.python.org/lib/re-syntax.html

{m,n} Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, $a\{3,5\}$ will match from 3 to 5 "a" characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example, $a\{4,\}b$ will match aaaab or a thousand "a" characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}? Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', $a\{3,5\}$ will match 5 "a" characters, while $a\{3,5\}$? will only match 3 characters.

"\\" Either escapes special characters (permitting you to match characters like "*", "?", and so forth), or signals a special sequence; special sequences are discussed below.

[] Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a "-". Special characters are not active inside sets. For example, [akm\$] will match any of the characters "a", "k", "m", or "\$"; $[a-z]$ will match any lowercase letter, and $[a-zA-Z0-9]$ matches any letter or digit. Character classes such as \w or \S (defined below) are also acceptable inside a range. If you want to include a "]" or a "-" inside a set, precede it with a backslash, or place it as the first character. The pattern []] will match ']', for example.

You can match the characters not within a range by complementing the set. This is indicated by including a "ˆ" as the first character of the set; "ˆ" elsewhere will simply match the "ˆ" character. For example, [ˆ5] will match any character except "5", and [ˆˆ] will match any character except "ˆ".

"|" $A|B$, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the "|" in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by "|" are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the "|" operator is never greedy. To match a literal "|", use \|, or enclose it inside a character class, as in [|].

(...) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals "(" or ")", use \\(or \\), or enclose them inside a character class: [(] [)].

(?=...) Matches if ... matches next, but doesn't consume any of the string. This is called a lookahead assertion. For example, $Isaac(?=Asimov)$ will match 'Isaac ' only if it's followed by 'Asimov'.

(?!...) Matches if ... doesn't match next. This is a negative lookahead assertion. For example, $Isaac(?!Asimov)$ will match 'Isaac ' only if it's not followed by 'Asimov'.

(?<=...) Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion. $(?<=abc)def$ will find a match in "abcdef", since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or $a|b$ are allowed, but $a*$ and $a\{3,4\}$ are not.

(?<!...) Matches if the current position in the string is not preceded by a match for .... This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

$\backslash A$ Matches only at the start of the string.

$\backslash b$ Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that $\backslash b$ is defined as the boundary between $\backslash w$ and $\backslash W$, so the precise set of characters deemed to be alphanumeric depends on the values of the UNICODE and LOCALE flags. Inside a character range, $\backslash b$ represents the backspace character, for compatibility with Python's string literals.

$\backslash B$ Matches the empty string, but only when it is not at the beginning or end of a word. This is just the opposite of $\backslash b$, so is also subject to the settings of LOCALE and UNICODE.

$\backslash d$ Matches any decimal digit; this is equivalent to the set $[0-9]$.

$\backslash D$ Matches any non-digit character; this is equivalent to the set $[\hat{}0-9]$.

$\backslash s$ Matches any whitespace character; this is equivalent to the set $[\ \backslash t\backslash n\backslash r\backslash f\backslash v]$.

$\backslash S$ Matches any non-whitespace character; this is equivalent to the set $[\hat{}\ \backslash t\backslash n\backslash r\backslash f\backslash v]$.

$\backslash w$ When the LOCALE and UNICODE flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set $[a-zA-Z0-9\_]$. With LOCALE, it will match the set $[0-9\_]$ plus whatever characters are defined as alphanumeric for the current locale. If UNICODE is set, this will match the characters $[0-9\_]$ plus whatever is classified as alphanumeric in the Unicode character properties database.

$\backslash W$ When the LOCALE and UNICODE flags are not specified, matches any non-alphanumeric character; this is equivalent to the set $[\hat{}a-zA-Z0-9\_]$. With LOCALE, it will match any character not in the set $[0-9\_]$, and not defined as alphanumeric for the current locale. If UNICODE is set, this will match anything other than $[0-9\_]$ and characters marked as alphanumeric in the Unicode character properties database.

$\backslash Z$ Matches only at the end of the string.

$\backslash a$ $\backslash f$ $\backslash n$ $\backslash r$ $\backslash t$ $\backslash v$ $\backslash x$ $\backslash\backslash$ Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser.

$\backslash 0xyz$, $\backslash xyz$ Octal escapes are included in a limited form: If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. As for string literals, octal escapes are always at most three digits in length.

6.2 Token definition

6.2.1 Predefined tokens

Tokens can be explicitely defined by the *Token*, *Drop* and *Separator* keywords.

Token defines a regular token (which returns a string).

Drop defines a token that returns nothing (usefull for keywords for instance).

Separator if a context manager used to define separators for the rules defined in the context.

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.

See figure 6.1 for examples.

Figure 6.1: Token definition examples

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

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

with Separator(spaces|comments):
    # rules defined here will use spaces and comments as separators
```

There are two kinds of tokens. Tokens defined by the *token* keyword 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".

6.2.2 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.

See figure 6.2 for examples.

Figure 6.2: Inline token definition examples

```
IfThenElse = Drop(r'if\b') & Cond &
             Drop(r'then\b') & Statement &
             Drop(r'else\b') & Statement
    ;
```

Chapter 7

Parser

7.1 Declaration

A parser is declared as a Python object.

7.2 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 (=).

See figure 7.1 for example.

Figure 7.1: Rule declaration

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

7.3 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
```

7.4 Alternatives

Alternatives in grammar rules describe several possible decompositions of a symbol. The infix pipe operator (|) is used to separate alternatives. $A \mid B$ recognizes either an A or a B. If both A and B can be matched only the first match is considered. So the order of alternatives is very important. If an alternative has an empty choice, it must be the last. Empty choices in other positions will be reported as syntax errors.

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

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

7.5 Repetitions

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

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.

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

7.6 Precedence and grouping

The figure 7.2 lists the different structures in increasing precedence order. To override the default precedence you can group expressions with parenthesis.

Figure 7.2: Precedence in SP expressions

Structure	Example
Alternative	$A \mid B$
Sequence	$A \& B$
Repetitions	$A[x : y]$
Symbol and grouping	A and (...)

7.7 Actions

Grammar rules can contain actions as Python functions.
Functions are applyied to parsed objects using / or $*$.
$parser/function$ returns $function(result\ of\ parser)$.
$parser * function$ returns $function(*result\ of\ parser)$.
$*$ can be used to analyse the result of a sequence.

7.7.1 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.
The figure 7.3 shows a node symbolizing a couple.

Figure 7.3: AST example

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

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

Part III

Some examples to illustrate SP

Chapter 8

Complete interactive calculator

8.1 Introduction

This chapter presents an extention of the calculator described in the tutorial (see 3). This calculator has more functions and a memory.

8.2 New functions

8.2.1 Memories

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 π and $2 * pi$ will return 6.28.

The variables are saved in a dictionnary.

8.3 Source code

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

```
#!/usr/bin/env python3

# 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
flt32: 32 bit float calculus        |
flt64: 64 bit float calculus        |
rat  : rational calculus            | this help : ?
"""

import struct
import sys

from sp import *
from fractions import Fraction

try:
    import readline
except ImportError:
    pass

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 reversed"%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))

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 = Fraction("%.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]

class Calc:

    def __init__(self):
```

```
self.number = Num

def bin2int(n):
    n = n.replace('_', '')
    n = n.replace('b', '')
    return int(n, 2)

def oct2int(n):
    n = n.replace('_', '')
    n = n.replace('o', '')
    return int(n, 8)

def hex2int(n):
    n = n.replace('_', '')
    n = n.replace('h', '')
    if n.startswith('0x'): n = n[2:]
    return int(n, 16)

def real2float(n):
    n = n.replace('_', '')
    return float(n)

def dec2int(n):
    n = n.replace('_', '')
    return int(n)

bin = (Token(r'b[_0-1]+\b') | Token(r'[_0-1]+b\b')) / bin2int
oct = (Token(r'o[_0-7]+\b') | Token(r'[_0-7]+o\b')) / oct2int
hex = ( Token(r'h[_0-9a-fA-F]+\b') |
        Token(r'[_0-9a-fA-F]+h\b') |
        Token(r'0x[_0-9a-fA-F]+\b')) / hex2int
real = Token(r'(\d+\.\d*|\d*\.\d+)([eE][-+]?\d+)?|\d+[eE][-+]?\d+') / real2float
dec = Token(r'\d+') / dec2int
var = Token(r'[a-zA-Z_]\w*\b')

from operator import pos, neg, invert, add, sub, or_, xor, \
                     mul, mod, truediv, floordiv, and_, \
                     rshift, lshift, pow as pow_
op1 = lambda f,x: {'+':pos, '-':neg, '~':invert}[f](x)
op = lambda f,y: lambda x: {'+':add, '-':sub,
                            '*':mul, '%':mod, '/':truediv,
                            '**':pow_,
                            '&':and_, '|':or_, '^':xor,
                            '>>':rshift, '<<':lshift,
                           }[f](x,y)
def red(x, fs):
    for f in fs: x = f(x)
    return x

with Separator(r'\s+'):

    expr = Rule()

    calc = ( Token(r'\?') / __doc__.strip()
                | Token('num') / self.mode(Num)
                | Token('int8') / self.mode(Int8)
                | Token('int16') / self.mode(Int16)
                | Token('int32') / self.mode(Int32)
                | Token('int64') / self.mode(Int64)
                | Token('int') / self.mode(Int)
                | Token('flt32') / self.mode(Float)
                | Token('flt64') / self.mode(Double)
                | Token('rat') / self.mode(Rat)
                | (((var & Drop('=')) | Const('_')) & expr) * self.assign
                )

            fact = Rule()
            atom = ( Drop(r'\(') & expr & Drop(r'\)')
                   | ('rev|factor' & Drop(r'\(') & expr & Drop(r'\)'))
                     * (lambda f,x: getattr(x, f)())
                   | (bin | oct | hex | real | dec | var / self.val)
                     / self.convert
                   )
            pow = (atom & ((r'\*\*' & fact) * op)[:1]) * red
            fact |= (r'\+|-|~' & fact) * op1 | pow
            term = (fact & ((r'\*|%|/|&|>>|<<' & fact) * op)[:]) * red
            expr |= (term & ((r'\+|-|\||\^' & term) * op)[:]) * red

        self.calc = calc
        self.var = {}

    def mode(self, m):
        def setmode(_):
            self.number = m
            return "%s mode"%self.number.descr
        return setmode

    def convert(self, x):
        return self.number(x)

    def assign(self, var, val):
        self.var[var] = val
        return str(val)

    def val(self, var):
        return self.var[var]

    def __call__(self, s):
        return self.calc(s)

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
    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 s[::-1]

if __name__ == '__main__':
    print(__doc__.strip())
    print()
    calc = Calc()
    for cmd in sys.argv[1:]:
        try:
            calc(cmd)
        except Exception as e:
            print("%s: %s"%(e.__class__.__name__, e))
    while True:
        expr = input("(%s) "%calc.number.name)
        if not expr: continue
        try:
            val = calc(expr)
        except Exception as e:
            print("%s: %s"%(e.__class__.__name__, e))
        else:
            print("=", val)
        print()
```