Python PLY (Python による構文解析)

[Python]

PLY (Python Lex-Yacc)

- http://www.dabeaz.com/ply/
- PLY 3.0 Document

参考

- http://www.okisoft.co.jp/esc/ply-algol/index.html
- lex
- yacc

概要

■ PLY-3.0 要件

- Python 2、Python 3 互換。ただし、Python2 系で、2.4 以降推奨

■紹介

- Pure Python による、lex および yacc の実装
- LALR(1) および広範な入力検査、エラー、診断レポートを備えるので、他言語で yacc を利用しているなら 比較的素直に PLY を利用できる

■概要

- PLY は、lex.py および yacc.py の 2 つのモジュールを ply パッケージに含んでいる。
- 2 つのツールは協調して作業を行う

lex.py

- モジュールは入力されたテキストを 正規表現により定義された字句のコレクションに分解する。
- 次の有効なトークンを入力ストリームから返す、token() 関数を、外部にインターフェースとして提供する。

yacc.py

- 自由文法として定義された言語の構文を評価する。
- LR で解析を行い、LALR(1) (デフォルト) もしくは SLR アルゴリズムでパースする。
- lex.py の token() を繰り返し呼び出し、トークンを参照し、文法ルールを呼び出す。
- 出力は、大抵、抽象構文木 (AST) となる。

Lex

■サンプル

lex_test.py

```
# -*- coding: utf-8 -*-
import ply.lex as lex


# トークンリスト 常に必須
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
)

# 正規表現による簡単なトークンのルール
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# 正規表現とアクションコード
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# 行番号をたどれるように
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# スペースおよびタブは無視
t_ignore = ' \t'

# エラーハンドリングルール
def t_error(t):
    print u"不正な文字 '%s'" % t.value[0]

# lexer を構築
lexer = lex.lex()

if __name__ == '__main__':

    # ここからテスト
    data = '''
      3 + 4 * 10
      + -20 *2
    '''

    lexer.input(data)

    while True:
        tok = lexer.token()
        if not tok:
            # これ以上トークンはない
            break
        print tok
```

・lexer はイテレーションプロトコル対応のため、while の部分は、以下のようにも書ける

```
for tok in lexer:
    print tok
```

結果

```
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
```

```
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,9)
LexToken(PLUS,'+',3,14)
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,17)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

■ LexToken

・lexer.token() が返す トークンは、LexToken のインスタンス
・以下の属性を持つ

属性	内容
type	文字列のようなトークンの型
value	語彙 正規表現がマッチした文字列
lineno	行番号
lexpos	入力データの中で、トークンの開始からの相対位置

例

```
for tok in lexer:
    print "type=%s, value=%s, lineno=%s, lexpos=%s" % (tok.type, tok.value, tok.lineno, tok.lexpos)
```

結果

```
type=NUMBER, value=3, lineno=2, lexpos=1
type=PLUS, value=+, lineno=2, lexpos=3
type=NUMBER, value=4, lineno=2, lexpos=5
type=TIMES, value=*, lineno=2, lexpos=7
type=NUMBER, value=10, lineno=2, lexpos=9
type=PLUS, value=+, lineno=3, lexpos=14
type=MINUS, value=-, lineno=3, lexpos=16
type=NUMBER, value=20, lineno=3, lexpos=17
type=TIMES, value=*, lineno=3, lexpos=20
type=NUMBER, value=2, lineno=3, lexpos=21
```

■トークンリスト

・すべての lexer はトークンのリストを提供する必要がある。
・このリストは常に必須であり、各種の妥当性検査の実行に使用される。
・yacc.py にて、終端を特定するのにも使われる。

■トークンの仕様

・トークンの仕様は、正規表現によって定められる。
・これらのルールは、特別なプレフィックス t_ で宣言する。
・正規表現による、簡単なトークンの例

```
t_PLUS = r'\+'
```

・アクションを実行する必要がある場合、関数にできる (文字列を整数に変換する例)

・関数を使う場合、正規表現ルールは、ドキュメント文字列で指定する
・関数は常に LexToken を 1 つ引数としてとる

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

■予約語の取り扱い

・予約語を取り扱うには、マッチさせる特別な識別子を 1 つ用意し、以下のようにすることで、正規表現の記述を大幅に削減できる

```
# 予約語
reserved = {
   'if' : 'IF',
   'then' : 'THEN',
   'else' : 'ELSE',
   'while' : 'WHILE',
   ...
}

# 識別子 ID を追加
tokens = ['LPAREN','RPAREN',...,'ID'] + list(reserved.values())

# 識別子 ID に対して、予約語をチェックする処理を追加
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'ID')    # 予約語のチェック
    return t
```

予約語に対して、以下のようにルールを記述できる

```
t_FOR   = r'for'
t_PRINT = r'print'
```

これらのルールは、"forget" or "printed" など、予約語を含む語彙にも一致してしまう。

■トークンの値

・lex からトークンが戻されるとき、トークンは value 属性を持ちます
・通常、value 属性はマッチしたテキストですが、任意の Python オブジェクトと結びつけることができます
・たとえば、識別子名と情報をソートされたシンボルテーブルから得たい場合、以下のようにします。

```
def t_ID(t):
    ...
    # シンボルテーブル情報を検索し、タプルを返す
    t.value = (t.value, symbol_lookup(t.value))
    ...
    return t
```

■トークンの破棄

・コメントのような破棄するトークンは、値を返さないことで定義する。

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # 値を返さないと、トークンは破棄される
```

・"ignore_" プレフィックスを利用してもよい

```
t_ignore_COMMENT = r'\#.*'
```

■行番号と位置情報

・デフォルトでは、lex.py は行番号について関知しない
・t_newline() という特別なルールを定義する必要がある

```
# 行番号の追跡を可能とする
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

■無視する文字

・t_ignore 特別ルールは、lex.py で入力を無視するために予約されている。
・通常、ホワイトスペースや他の本質的でない文字ををスキップするために利用される

■リテラル文字

・字句解析モジュールで、literals 変数を定義することで、リテラル文字を定義できる

```
literals = [ '+','-','*','/' ]
```

・もしくは

```
literals = "+-*/"
```

■エラーハンドリング

・t_error() 関数は不正な文字が見つかった場合に発生する字句解析エラーをハンドリングする。

```
# エラーハンドリングルール
def t_error(t):
    print u"不正な文字 '%s'" % t.value[0]
    t.lexer.skip(1)
```

■ lexer を構築して利用する

・lexer(字句解析機) を構築するには、lex.lex() 関数を利用する
・Python のリフレクション (またはイントロスペクション) を使って、正規表現のルールを lexer の呼び出しと構築コンテキストの外側から読み込む。
・lexer が作られると、2 つのメソッドで制御できるようになる

lexer.input(data)

・lexer をリセットし、新しい入力文字列をセットする

lexer.token()

・次のトークンを返す。成功時には LexToken のインスタンスを、入力テキストが終了したら None を返す

YACC

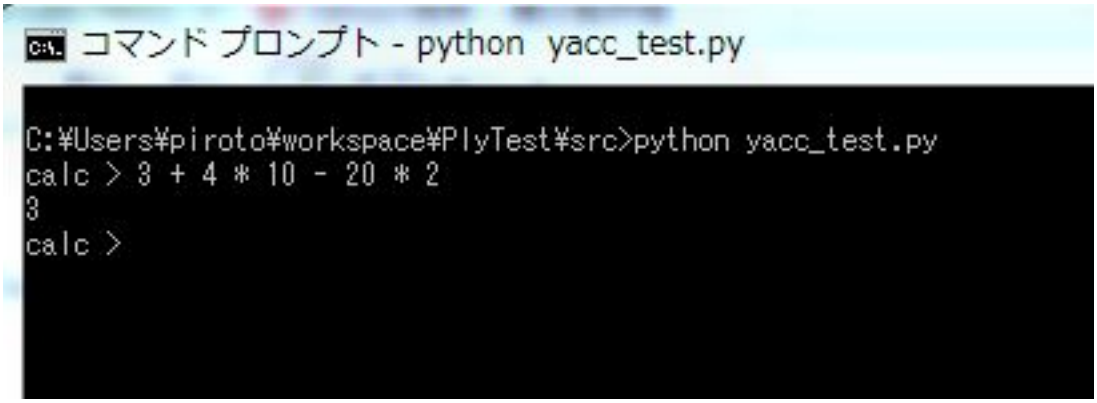
・ply.yacc モジュールは、PLY の構文解析コンポーネント。

■サンプル

yacc_test.py

```
# -*- coding: utf-8 -*-

import ply.yacc as yacc

# Lex のサンプルを参照する
from lex_test import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# 構文エラー
def p_error(p):
    print "Syntax error in input"

# 構文解析器の構築
parser = yacc.yacc()

if __name__ == '__main__':
    while True:
        try:
            s = raw_input('calc > ')
        except EOFError:
            break
        if not s:
            continue
        result = parser.parse(s)
        print result
```

実行結果



説明

- このサンプルでは、すべての構文ルールが Python の関数によって定義されており、その関数のドキュメント文字列は適切な文脈自由の構文の仕様を含んでいる。
- 関数本体のステートメントは構文の意味があらわすアクションをルールとして実装している。
- それぞれの関数はひとつの引数 p を受け取る。
- 引数 p は、シーケンス (連続した値) であり、ルールに一致したそれぞれのシンボルの値を含んでいる。
- 以下のように、p[i] の値は、文法のシンボルにマップされる。

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ↑            ↑       ↑  ↑
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

6.2 Combining Grammar Rule Functions

■文法ルールを合成する

- 文法ルールが似ている場合、ひとつの関数に合成できる。

例

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]


def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

2 つの関数を書く代わりに、以下のようにひとつの関数にすることができる

```
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

■文字リテラル

・希望するなら、1 文字のリテラルで定義されたトークンを含ませてもよい

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
       term       : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

・文字リテラルは、引用符で '+' のように囲む必要がある。
・リテラルが利用された場合、対応する lex ファイルに特別なリテラル宣言をする必要がある

```
# リテラル lex() モジュールにて宣言する必要がある
literals = ['+','-','*','/' ]
```

文字リテラルを利用できるのは、1 文字のシンボルのみ

■空の値を生成する

・yacc.py は、以下のようにルールを定義された、空の値を生成する関数を取り扱うことができる

```
def p_empty(p):
    'empty :'
    pass
```

・空の値を生成する関数を使うには、'empty' シンボルを利用すればよい

```
def p_optitem(p):
    'optitem : item'
    '        | empty'
    ...
```

■開始シンボルを変更

・通常最初に見つかった構文ルールから、適用されるが、これを変更できる。

```
start = 'foo'

def p_bar(p):
    'bar : A B'

# 上記の start 指定によってここからがルールの定義とされる
def p_foo(p):
    'foo : bar X'
    ...
```

start 指定は、大きな文法のサブセットを利用してデバッグする場合などに有用

・以下のようにも指定できる

```
yacc.yacc(start='foo')
```