RPly Documentation

Release 0.7.4

Alex Gaynor

December 18, 2016

Contents

Welcome to RPLY! A pure python parser generator, that also works with RPython. It is a more-or-less direct port of David Beazley's awesome PLY, with a new public API, and RPython support.

To start using RPly head over to Generating Lexers to learn how to generate lexers, and Generating Parsers after that to see how you can turn the tokens generated by your lexer into something more useful.

CHAPTER 1

User's Guide

1.1 Generating Lexers

In order to parse text, you first have to turn that text into individual tokens with a lexer. Such a lexer can be generated with the `rply.LexerGenerator`.

Lexers are generated by adding rules to a *LexerGenerator* instance. Such a rule consists of a name, which will be used as the type of the token generated with that rule, and a regular expression defining the piece of text to be matched.

As an example we will attempt to generate a lexer for simple mathematical expressions:

```
lg = LexerGenerator()

lg.add('NUMBER', r'\d+')

lg.add('PLUS', r'\+')
lg.add('MINUS', r'-')
```

We have no defined rules for numbers, an addition and subtraction operator. We can now build a lexer and use it:

```
>>> l = lg.build()
>>> for token in l.lex('1+1-1'):
...     print(token)
...
Token(NUMBER, '1')
Token(ADD, '+')
Token(NUMBER, '1')
Token(MINUS, '-')
Token(NUMBER, '1')
```

This works quite nicely however there is but a small problem:

```
>>> for token in l.lex('1 + 1'):
...     print(token)
Token('NUMBER', '1')
Traceback (most recent call last):
...
rply.errors.LexingError
```

What happened is that the lexer is able to match the *1* at the beginning of the string and it yields the correct token for that but afterwards the string ‘‘ + 1‘‘ is left and no rule matches.

While we do want lexing to continue at that stage, we do not care about whitespace and would like to ignore it. This can be done using `ignore()`:

```
lg.ignore(r'\s+')
```

This adds a rule which will be ignored by the lexer and not produce any tokens:

```
>>> l = lg.build()
>>> for token in l.lex('1 + 1'):
...     print(token)
...
Token('NUMBER', '1')
Token('ADD', '+')
Token('NUMBER', '1')
```

With this you know everything there is to know about generating lexers.

1.2 Generating Parsers

In this part of the tutorial we will generate a parser for simple mathematical expressions as defined by the following BNF grammar:

```
<expression> ::= "\d+"
               | <expression> "+" <expression>
               | <expression> "-" <expression>
               | <expression> "*" <expression>
               | <expression> "/" <expression>
               | "(" <expression> ")"
```

Furthermore we use the following lexer:

```
from rply import LexerGenerator

lg = LexerGenerator()

lg.add('NUMBER', r'\d+')
lg.add('PLUS', r'\+')
lg.add('MINUS', r'-')
lg.add('MUL', r'\*')
lg.add('DIV', r'/')
lg.add('OPEN_PARENS', r'\(')
lg.add('CLOSE_PARENS', r'\)')

lg.ignore('\s+')

lexer = lg.build()
```

Before we begin working on the parser, we define ourselves an abstract syntax tree:

```
from rply.token import BaseBox

class Number(BaseBox):
    def __init__(self, value):
        self.value = value

    def eval(self):
        return self.value

class BinaryOp(BaseBox):
    def __init__(self, left, right):
```

```
        self.left = left
        self.right = right

class Add(BinaryOp):
    def eval(self):
        return self.left.eval() + self.right.eval()

class Sub(BinaryOp):
    def eval(self):
        return self.left.eval() - self.right.eval()

class Mul(BinaryOp):
    def eval(self):
        return self.left.eval() * self.right.eval()

class Div(BinaryOp):
    def eval(self):
        return self.left.eval() / self.right.eval()
```

In RPython variables must have a specific type, so we use polymorphism with `BaseBox` to ensure that. In your own code you can achieve the same by inheriting from `BaseBox` as we did here. If you are not writing RPython code, you can ignore this completely.

Having covered all that we actually start working on the parser itself:

```
from rply import ParserGenerator

pg = ParserGenerator(
    # A list of all token names, accepted by the parser.
    ['NUMBER', 'OPEN_PARENS', 'CLOSE_PARENS',
     'PLUS', 'MINUS', 'MUL', 'DIV'
    ],
    # A list of precedence rules with ascending precedence, to
    # disambiguate ambiguous production rules.
    precedence=[
        ('left', ['PLUS', 'MINUS']),
        ('left', ['MUL', 'DIV'])
    ]
)

@pg.production('expression : NUMBER')
def expression_number(p):
    # p is a list of the pieces matched by the right hand side of the
    # rule
    return Number(int(p[0].getstr()))

@pg.production('expression : OPEN_PARENS expression CLOSE_PARENS')
def expression_parens(p):
    return p[1]

@pg.production('expression : expression PLUS expression')
@pg.production('expression : expression MINUS expression')
@pg.production('expression : expression MUL expression')
@pg.production('expression : expression DIV expression')
def expression_binop(p):
    left = p[0]
    right = p[2]
    if p[1].gettokentype() == 'PLUS':
        return Add(left, right)
```

```
    elif p[1].gettokentype() == 'MINUS':
        return Sub(left, right)
    elif p[1].gettokentype() == 'MUL':
        return Mul(left, right)
    elif p[1].gettokentype() == 'DIV':
        return Div(left, right)
    else:
        raise AssertionError('Oops, this should not be possible!')

parser = pg.build()
```

As you can see production rules define a sequence of terminals (tokens) and non-terminals (intermediate values, in this case only *expression*) using the `production()` decorator. The function receives a list of the tokens and non-terminals and returns a non-terminal.

In this case we create an abstract syntax tree. We can now use this parser in combination with the lexer given to parse and evaluate mathematical expressions as defined by our grammar:

```
>>> parser.parse(lexer.lex('1 + 1')).eval()
2
>>> parser.parse(lexer.lex('1 + 2 * 3')).eval()
7
```

1.2.1 Error Handling

As long as we parse code that is well formed according to our grammar, all is fine but one of the more difficult problems in writing a parser is handling errors.

Per default in case of an error you get a `rply.ParsingError`:

```
>>> parser.parse(lexer.lex('1 1'))
```

This error will not provide any information apart from the position at which it occurred accessible through `getsourcepos()`.

While it is not possible to recover from an error, you can define your own error handler:

```
@pg.error
def error_handler(token):
    raise ValueError("Ran into a %s where it was't expected" % token.gettokentype())
```

The *token* passed to the error handler will be the token the parser errored on.

1.2.2 Maintaining State

Sometimes it can be useful to have additional state within the parser, for example as a way to pass information to the parser about the name of the file currently being parsed.

In order to do this we simply define a state object to pass around:

```
class ParserState(object):
    def __init__(self, filename):
        self.filename = filename
```

We can pass *ParserState* objects to the parser simply like this:

```
parser.parse(lexer.lex(source), state=ParserState('foo.py'))
```

This will call every production rule and the error handler with the *ParserState* instance as first argument.

1.2.3 Precedence on rules

Sometimes it is useful to give a rule a manual precedence. For this pass the *precedence* argument to *production*. For example, if we wanted to add an implicit multiplication rule to the above language (so that e.g. *16 32* is parsed the same as *16 * 32*) we use the following:

```
@pg.production('expression : expression expression', precedence='MUL')
def implicit_multiplication(p):
    return Mul(p[0], p[1])
```

CHAPTER 2

API Documentation

The API documentation provides detailed information on the functions, classes and methods, provided by RPly. If you are looking for something specific, take a look at one of these documents:

2.1 rply

2.2 rply.token

CHAPTER 3

Additional Information

Everything not directly related to the usage of RPly.

3.1 License

r

R