Introduction to Pyparsing: An Object-oriented Easy-to-Use Toolkit for Building Recursive Descent Parsers

Author: Paul McGuire <ptmcg@austin.rr.com>

Version: 1.1

Agenda

- Basic parsing terminology
- The Zen of Pyparsing
- Two Kinds of Parsing Applications
- Basic Pyparsing
- Helpful Pyparsing Built-ins
- Intermediate Pyparsing
- Advanced Pyparsing
- Debugging Pyparsing Grammars
- Pyparsing Applications
- Who's Using Pyparsing?
- Where to Get Pyparsing

Basic parsing terminology

- parser
- tokenizer
- grammar (aka BNF)
- recursive-descent

The Zen of Pyparsing

Don't clutter up the parser grammar with whitespace, just handle it! (likewise for comments)

Grammars must tolerate change, as grammar evolves or input text becomes more challenging.

Grammars do not have to be exhaustive to be useful.

Simple grammars can return lists; complex grammars need named results.

Class names are easier to read and understand than specialized typography.

Parsers can (and sometimes should) do more than tokenize.

The Zen of Pyparsing Development

Grammars should be:

- easy (and *quick*) to write
- easy to read
- easy to update

No separate code-generation step.

No imposed naming conventions.

Stay Pure Python.

Two Kinds of Parsing Applications

Design-driven:

```
language -> BNF -> parser impl --+->
 concept     ^                   |
             |   refine/extend   |
             +---  language    --+
```

Data-driven:

```
gather    determine
sample ->   text    -> BNF -> parser ---+->
inputs     patterns                     |
             ^                          |
             |         gather new       |
             +----- (non-conforming) ---+
                         inputs
```

Pyparsing "Hello World"

Compose grammar:

```
greeting = oneOf("Hello Ahoy Yo Hi") +
           Literal(",") +
           Word( string.uppercase, string.lowercase ) +
           Literal("!")
```

Call parseString():

```
print greeting.parseString("Hello, World!")
print greeting.parseString("Ahoy, Matey !")
print greeting.parseString("Yo,Adrian!")
['Hello', ',', 'World', '!']
['Ahoy', ',', 'Matey', '!']
['Yo', ',', 'Adrian', '!']
```

Basic Pyparsing

Words and Literals

- Word:

```
Word("ABC","def") matches "C", "Added", "Beef",
                  but not "BB", "ACE", "ADD"
```

- Literal:

```
Literal("name(x)") matches "name(x)",
                   but not "name (x)"
```

- CaselessLiteral:

```
CaselessLiteral("ABC") matches "abc", "ABC", "AbC"
                   returns "ABC" in all cases
```

Basic Pyparsing (2)

Combining

- And (operator `+`) - must match all (ignoring whitespace):

```
assignment = varName + Literal("=") + arithExpression
```

  or:

```
assignment = varName + "=" + arithExpression
```

- Or (operator `^`) - match longest:

```
qty = integerNum ^ realNum
```

- MatchFirst (operator `|`) - match first:

```
arithOp = Literal("+") | Literal("-") | Literal("X") |
            Literal("/") | Literal("^") | Literal("%")
```

Basic Pyparsing (3)

Repetition and Collection

- OneOrMore, ZeroOrMore
- Group
- Combine
- Optional

Whitespace is not matched as part of the grammar:

```
Literal("name") + Literal("(") + Literal("x") + Literal(")")
    matches "name(x)", "name (x)", "name( x )", "name ( x )"
```

Helpful Pyparsing Built-ins

- oneOf("string of space-separated literal strings"):

  ```
  matches "strings", "string", "of", "space-separated", or "literal"

  arithOp = oneOf("+ - X / ^ %")
  ```

- delimitedList(expression, delim=','):

  ```
  matches expression or expression,expression,...
  ```

- commaSeparatedList:

  ```
  special form of delimitedList with a pre-defined
  expression for comma-separated values (strings, quoted
  strings, integers, etc.)
  ```

- quotedString, dblQuotedString, sglQuotedString:

  ```
  matches "abc", 'def', "string with escaped \" character"
  ```

Helpful Pyparsing Built-ins (2)

- makeHTMLTags("A") - returns beginning and ending tags:

  ```
  beginning matches "<A>", "<a href="blah">", "<A/>"
  ending matches "</A>", "</a>"
  ```

- makeXMLTags("BODY") - like makeHTMLTags, but with tighter case matching:

  ```
  beginning matches "<BODY>", "<BODY align="top">"
      but not "<body>" or "<Body>"
  ending matches "</BODY>", but not "</body>"
  ```

- Typical comment forms:

  ```
  cStyleComment     /* block comment,
                         spans lines */
  dblSlashComment   // everything up to end of line
  cppStyleComment = cStyleComment | dblSlashComment
  htmlComment       <!-- commented out HTML -->
  pythonComment     # everything up to end of line
  ```

Helpful Pyparsing Built-ins (3)

- Strings of common character sets for Word definitions:

```
alphas - A-Za-z
alphas8bit - accented and umlauted characters
    ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþ
nums - 0-9
alphanums = alphas + nums
hexnums = 0-9A-Fa-f
printables = string.printables, minus space character
srange('A-Za-z')
    same as 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcde...'
```

Intermediate Pyparsing

SkipTo - "wildcard" advance to expression

Suppress - match, but suppress matching results; useful for punctuation, delimiters

NotAny - negative lookahead

FollowedBy - assertive lookahead

Intermediate Pyparsing (2)

Other ParserElement methods

- ignore - ignore and suppress occurrences of given expression; useful for comments
- setParseAction - invoke user routine when parse expression matches; parse action may update the matched tokens before returning
- setResultsName - assign name to matched tokens, to support access to results by name (attribute or dict form)
- scanString - searches input text for matches; for each match, returns tuple of tokens, start location, and end location
- transformString - similar to scanString, but applies token suppression and parse actions to transform the input string
- searchString - another simplfied form of scanString, that searches for the first occurrence of matching text

Advanced Pyparsing

Forward - to define recursive grammars - example in just a few slides...

Regex - define regular expression to match

For example:

```
Regex(r"(\+|-)?\d+")
```

is equivalent to:

```
Combine( Optional(oneOf("+ -")) + Word(nums) )
```

Named re groups in the Regex string are also preserved as ParseResults names

(Don't be too hasty to optimize, though...)

Advanced Pyparsing (2)

Dict - creates named fields using input data for field names (see pyparsing examples)

StringStart, StringEnd, LineStart, LineEnd - positional expressions for line-break dependent expressions

White - (sigh...) if you must parse whitespace, use this class

Exceptions

- ParseException - parse actions can throw this to fail the current expression match
- FatalParseException - parse actions can throw this to abort parsing
- exceptions support lineno, col, line, and msg attributes to debug parsing errors

Debugging Pyparsing Grammars

setName(exprName) - useful for exceptions and debugging, to name parser expression (instead of default repr-like string)

For example, compare:

```
integer = Word(nums)
(integer + integer).parseString("123 J56")
pyparsing.ParseException: Expected W:(0123...) (at char 4), (line:1, col:5)
```

vs:

```
integer = Word(nums).setName("integer")
(integer + integer).parseString("123 J56")
pyparsing.ParseException: Expected integer (at char 4), (line:1, col:5)
```

Debugging Pyparsing Grammars (2)

setDebug() - activates logging of all match attempts, failures, and successes; custom debugging callback methods can be passed as optional arguments:

```
integer.setDebug()
(integer + integer).parseString("123 A6")
```

prints:

```
Match integer at loc 0 (1,1)
Matched integer -> ['123']
Match integer at loc 3 (1,4)
```

```
Exception raised: Expected integer (at char 4), (line:1, col:5)
```

Pyparsing Documentation

Pyparsing distribution includes

- PNG and JPG UML class diagrams
- epydoc-generated help files
- examples directory

Online resources

- SourceForge pyparsing support forum (RSS feed via Gmane)

- comp.lang.python (but not always best)

- ONLamp article, Building Recursive Descent Parsers with Python

    - http://www.onlamp.com/pub/a/python/2006/01/26/pyparsing.html

Pyparsing Applications

- Parsing lists
- HTML scraping
- JSON parser
- Extracting C function declarations

Parsing Lists

How to parse the string "['a', 100, 3.14]" back to a Python list

- without using eval

First pass:

```
lbrack = Literal("[")
rbrack = Literal("]")
integer = Word(nums).setName("integer")
real = Combine(Optional(oneOf("+ -")) + Word(nums) + "." +
               Optional(Word(nums))).setName("real")

listItem = real | integer | quotedString

listStr = lbrack + delimitedList(listItem) + rbrack
```

Parsing Lists (2)

```
test = "['a', 100, 3.14]"

print listStr.parseString(test)
```

Returns:

```
['[', "'a'", '100', '3.14', ']']
```

Brackets []'s are significant during parsing, but are not interesting as part of the results (just as delimitedList discarded the delimiting commas for us):

```
lbrack = Suppress("[")
rbrack = Suppress("]")
```

Parsing Lists (3)

We also want actual values, not strings, and we really don't want to carry the quotes around from our quoted strings, so use parse actions for conversions:

```
cvtInt = lambda s,l,toks: int(toks[0])
integer.setParseAction( cvtInt )

cvtReal = lambda s,l,toks: float(toks[0])
real.setParseAction( cvtReal )

listItem = real | integer | quotedString.setParseAction(removeQuotes)
```

Updated version now returns:

```
['a', 100, 3.1400000000000001]
```

Parsing Lists (4)

How would we handle nested lists?

We'd like to expand listItem to accept listStr's, but this creates a chicken-and-egg problem - listStr is defined in terms of listItem.

Solution: Forward declare listStr before listItem, using Forward():

```
listStr = Forward()
```

Add listStr as another type of listItem - enclose as Group to preserve nesting:

```
listItem = real | integer | quotedString.setParseAction(removeQuotes) \
            | Group(listStr)
```

Finally, define listStr using '<<' operator instead of '=' assignment:

```
listStr << ( lbrack + delimitedList(listItem) + rbrack )
```

Parsing Lists (5)

Using a nested list as input:

```
test = "['a', 100, 3.14, [ +2.718, 'xyzzy', -1.414] ]"
```

We now get returned:

```
['a', 100, 3.1400000000000001, [2.718, 'xyzzy', -1.4139999999999999]]
```

Parsing Lists (6)

Entire program:

```
cvtInt = lambda s,l,toks: int(toks[0])
cvtReal = lambda s,l,toks: float(toks[0])

lbrack = Suppress("[")
rbrack = Suppress("]")
integer = Word(nums).setName("integer").setParseAction( cvtInt )
real = Combine(Optional(oneOf("+ -")) + Word(nums) + "." +
               Optional(Word(nums))).setName("real").setParseAction( cvtReal )

listStr = Forward()
listItem = real | integer | quotedString.setParseAction(removeQuotes)
            | Group(listStr)
listStr << ( lbrack + delimitedList(listItem) + rbrack )

test = "['a', 100, 3.14, [ +2.718, 'xyzzy', -1.414] ]"
print listStr.parseString(test)
```

HTML Scraping

Want to extract all links from a web page to external URLs.

Basic form is:

```
<a href="www.blah.com/xyzzy.html">Zork magic words</a>
```

But anchor references are not always so clean looking:

- can contain other HTML attributes
- may start with "A" instead of "a", or use "HREF" instead of "href"
- href arg not always enclosed in quotes
- open tags can implicitly close with trailing '/'
- can be mixed in with other HTML tags
- can be embedded within HTML comments

HTML Scraping (2)

Use pyparsing makeHTMLTags helper method:

```
aStart,aEnd = makeHTMLTags("A")
```

Compose suitable expression:

```
link = aStart + SkipTo(aEnd).setResultsName("link") + aEnd
```

Skip any links found in HTML comments:

```
link.ignore(htmlComment)
```

Use scanString to search through HTML page - avoids need to define complete HTML grammar:

```
for toks,start,end in link.scanString(htmlSource):
    print toks.link, "->", toks.startA.href
```

HTML Scraping (3)

Complete program:

```
from pyparsing import makeHTMLTags,SkipTo,htmlComment
import urllib

serverListPage = urllib.urlopen( "http://www.yahoo.com" )
htmlText = serverListPage.read()
serverListPage.close()

aStart,aEnd = makeHTMLTags("A")

link = aStart + SkipTo(aEnd).setResultsName("link") + aEnd
link.ignore(htmlComment)

for toks,start,end in link.scanString(htmlText):
    print toks.link, "->", toks.startA.href
```

HTML Scraping (4)

Results:

```
<img src="http://us.i1.yimg.com/us.yimg.com/i/ww/bt1/ml.gif" width=36 height=36 border=0
<img src="http://us.i1.yimg.com/us.yimg.com/i/ww/bt1/my.gif" width=36 height=36 border=0
<img src="http://us.i1.yimg.com/us.yimg.com/i/ww/bt1/msgn.gif" width=36 height=36 border=
Advanced -> r/so
My Web -> r/mk
Answers <img src="http://us.i1.yimg.com/us.yimg.com/i/ww/beta.gif" border="0"> -> r/1l
<b>Yahoo! Health</b>: Make your resolutions a reality. -> s/266569
Lose weight -> s/266570
get fit -> s/266571
and be healthy -> s/266572
<b>Sign In</b> -> r/l6
<b>Sign Up</b> -> r/m7
360&#176; -> r/3t
```

```
Autos -> r/cr
Finance -> r/sq
Games -> r/pl
```

JSON Parser

JSON (JavaScript Object Notation) is a simple serialization format for JavaScript web applications:

```
object
    { members }
    {}
members
    string : value
    members , string : value
array
    [ elements ]
    []
elements
    value
    elements , value
value
    string | number | object | array| true| false | null
```

JSON Parser

Keyword constants and basic building blocks:

```
TRUE = Keyword("true")
FALSE = Keyword("false")
NULL = Keyword("null")

jsonString = dblQuotedString.setParseAction( removeQuotes )
jsonNumber = Combine( Optional('-') + ( '0' | Word('123456789',nums) ) +
                    Optional( '.' + Word(nums) ) +
                    Optional( Word('eE',exact=1) + Word(nums+'+-',nums) ) )
```

JSON Parser

Compound elements:

```
jsonObject = Forward()
jsonValue = Forward()
jsonElements = delimitedList( jsonValue )
jsonArray = Group( Suppress('[') + jsonElements + Suppress(']') )
jsonValue << ( jsonString | jsonNumber | jsonObject  |
               jsonArray | TRUE | FALSE | NULL )
memberDef = Group( jsonString + Suppress(':') + jsonValue )
jsonMembers = delimitedList( memberDef )
jsonObject << Dict( Suppress('{') + jsonMembers + Suppress('}') )

jsonComment = cppStyleComment
jsonObject.ignore( jsonComment )
```

JSON Parser

Conversion parse actions:

```
TRUE.setParseAction( replaceWith(True) )
FALSE.setParseAction( replaceWith(False) )
NULL.setParseAction( replaceWith(None) )

jsonString.setParseAction( removeQuotes )

def convertNumbers(s,l,toks):
    n = toks[0]
    try:
        return int(n)
    except ValueError, ve:
        return float(n)

jsonNumber.setParseAction( convertNumbers )
```

JSON Parser - Test Data

```
{   "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList":
                [{
                "ID": "SGML",
                "SortAs": "SGML",
                "GlossTerm": "Standard Generalized Markup Language",
                "TrueValue": true,
                "FalseValue": false,
                "IntValue": -144,
                "FloatValue": 6.02E23,
                "NullValue": null,
                "Acronym": "SGML",
                "Abbrev": "ISO 8879:1986",
                "GlossDef": "A meta-markup language, used to create markup languages such
                "GlossSeeAlso": ["GML", "XML", "markup"]
                }] } } }
```

JSON Parser - Results

```
[['glossary',
  ['title', 'example glossary'],
  ['GlossDiv',
   ['title', 'S'],
   ['GlossList',
    [['ID', 'SGML'],
     ['SortAs', 'SGML'],
     ['GlossTerm', 'Standard Generalized Markup Language'],
     ['TrueValue', True],
     ['FalseValue', False],
     ['IntValue', -144],
     ['FloatValue', 6.02e+023],
     ['NullValue', None],
     ['Acronym', 'SGML'],
     ['Abbrev', 'ISO 8879:1986'],
     ['GlossDef',
      'A meta-markup language, used to create markup languages such as DocBook.'],
     ['GlossSeeAlso', ['GML', 'XML', 'markup']]]]]]]
```

JSON Parser - Results

Accessing results by hierarchical path name:

```
print results.glossary.title
print results.glossary.GlossDiv.GlossList.keys()
print results.glossary.GlossDiv.GlossList.ID
print results.glossary.GlossDiv.GlossList.FalseValue
print results.glossary.GlossDiv.GlossList.Acronym

example glossary
['GlossSeeAlso', 'GlossDef', 'FalseValue', 'Acronym', 'GlossTerm',
 'TrueValue', 'FloatValue', 'IntValue', 'Abbrev', 'SortAs', 'ID',
 'NullValue']
SGML
False
SGML
```

Extracting C Function Declarations

Implementer of a C API needed to parse the library header to extract the function and parameter declarations.

API followed a very predictable form:

```
int func1(float *arr, int len, double arg1);
int func2(float **arr, float *arr2, int len, double arg1, double arg2);
```

Was able to define a straightforward grammar using pyparsing:

```
ident = Word(alphas, alphanums + "_")
vartype = oneOf("float double int char") + Optional(Word("*"))
arglist = delimitedList( Group(vartype + ident) )

functionCall = Literal("int") + ident + "(" + arglist+ ")" + ";"
```

Added results names to simplify access to individual function elements

Extracting C Function Declarations (2)

Complete program:

```
from pyparsing import *
testdata = """
```

```
  int func1(float *arr, int len, double arg1);
  int func2(float **arr, float *arr2, int len, double arg1, double arg2);
  """
ident = Word(alphas, alphanums + "_")
vartype = Combine( oneOf("float double int") + Optional(Word("*")), adjacent = False)
arglist = delimitedList( Group(vartype.setResultsName("type") +
                                ident.setResultsName("name")) )
functionCall = Literal("int") + ident.setResultsName("name") + \
                    "(" + arglist.setResultsName("args") + ")" + ";"

for fn,s,e in functionCall.scanString(testdata):
    print fn.name
    for a in fn.args:
        print " -",a.type, a.name
```

Extracting C Function Declarations (3)

Results:

```
func1
 - float* arr
 - int len
 - double arg1
func2
 - float** arr
 - float* arr2
 - int len
 - double arg1
 - double arg2
```

Who's Using Pyparsing?

- matplotlib - TeX markup parsing
- pydot - DOT parser
- twill - web app tester/driver
- ldaptor - LDAP constraint parsing/compiling
- BOAConstructor - source upgrade utility
- Process Stalker - internal GML parser
- VIMen - GDB plugin for vim
- rdfLib - Sparql parser
- MathDOM - MathML parser
- Distribulator - distributed admin command parser

Where to Get Pyparsing

Linux distributions:

- Debian
- Ubuntu
- Fedora
- Gentoo

- Conectiva

CheeseShop:

- http://cheeseshop.python.org/pypi/pyparsing

SourceForge:

- http://pyparsing.sourceforge.net

Summary

Pyparsing is not the solution to every text processing problem, but it is a *potential* solution for many such problems.