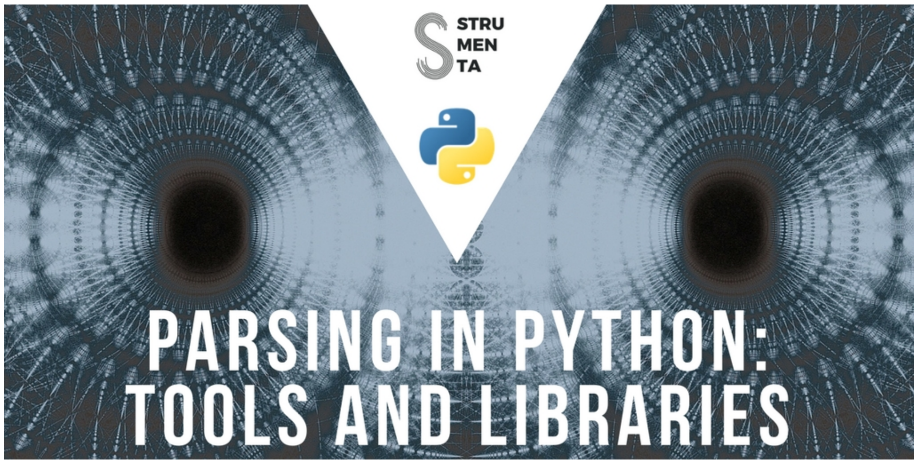
FEDERICO TOMASSETTI
Software Architect

Blog

Parsing In Python: Tools And Libraries

July 19, 2017 / in Parsing / by Gabriele Tomassetti



This is an article similar to a previous one we wrote: Parsing in Java [https://tomassetti.me/parsing-in-java/] , so the introduction is the same. Skip to chapter 3 if you have already read it.

If you need to parse a language, or document, from Python there are fundan
ways to solve the problem:

- use an existing library supporting that specific language: for example a lib
  XML
- building your own custom parser by hand
- a tool or library to generate a parser: for example ANTLR, that you can us
  parsers for any language

Get the free Cheatshee

Sign up to receive the cheatsheet version
guide in your inbox, to read it on all your d
whenever you want.

Name Email

Sign Up & Get the Cheatsheet

COURSE: USING ANTLR LIKE A PROFESSIONAL

A complete video course on parsing and ANTLR, that will teach you how to build parser for everything from programming languages to data formats. Taught from professionals that build parsers for a living.

Cheatsheet of Parsing in Python

A cheatsheet that sums up the strong and weak points of each parsing tool and tells you which tools to use for each parsing need

Sign up to download this FREE Cheatsheet

Email Address

GET THE CHEATSHEET

Use An Existing Library

The first option is the best for well known and supported languages, like XML or HTML. A good library usually include also API to programmatically build and modify documents in that language. This is typically more of what you get from a basic parser. The problem is that such libraries are not so common and they support only the most common languages. In other cases you are out of luck.

Building Your Own Custom Parser By Hand

You may need to pick the second option if you have particular needs. Both in the sense that the language you need to parse cannot be parsed with traditional parser generators, or you have specific requirements that you cannot satisfy using a typical parser generator. For instance, because you need the best possible performance or a deep integration between different components.

A Tool Or Library To Generate A Parser

In all other cases the third option should be the default one, because is the one that is most flexible and has the shorter development time. That is why on this article we concentrate on the tools and libraries that correspond to this option.

Note: text in blockquote describing a program comes from the respective documentation

Table Of Contents

If you need help designing and developing DSLs, languages, parsers, compilers, interpreters, and editors you can check the services page of the Consulting studio we founded: Strumenta.

STRUMENTA

BLOG CATEGORIES

ANTLR (14)
Code processing (21)
Consulting (13)
Domain specific languages (14)
Jetbrains MPS (11)
Language design (13)
Language Engineering (28)
Miscellany (4)
Model driven development (4)
Non software development (3)
Open-source (8)
Parsing (19)

Cheatsheet of Parsing in Python

A cheatsheet that sums up the strong and weak points of each parsing tool and tells you which tools to use for each parsing need

Sign up to download this FREE Cheatsheet

Email Address

GET THE CHEATSHEET

Tools To Create Parsers

We are going to see:

- tools that can generate parsers usable from Python (and possibly from other languages)
- Python libraries to build parsers

Tools that can be used to generate the code for a parser are called **parser generators** or **compiler compiler**. Libraries that create parsers are known as **parser combinators**.

Parser generators (or parser combinators) are not trivial: you need some time to learn how to use them and not all types of parser generators are suitable for all kinds of languages. That is why we have prepared a list of the best known of them, with a short introduction for each of them. We are also concentrating on one target language: Python. This also means that (usually) the parser itself will be written in Python.

To list all possible tools and libraries parser for all languages would be kind of interesting, but not that useful. That is because there will be simple too many options and we would all get lost in them. By concentrating on one programming language we can provide an apples-to-apples comparison and help you choose one option for your project.

Useful Things To Know About Parsers

To make sure that these list is accessible to all programmers we have prepa
explanation for terms and concepts that you may encounter searching for a
not trying to give you formal explanations, but practical ones.

Structure Of A Parser

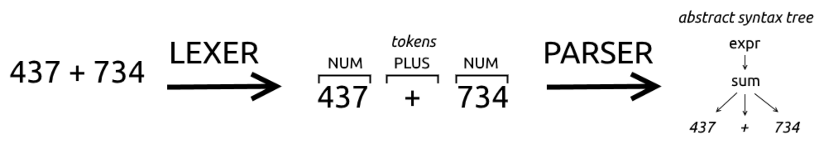
A parser is usually composed of two parts: a *lexer*, also known as *scanner* o
the proper parser. Not all parsers adopt this two-steps schema: some parser
depend on a lexer. They are called *scannerless parsers*.

A lexer and a parser work in sequence: the lexer scans the input and produc
matching tokens, the parser scans the tokens and produces the parsing resu

Let's look at the following example and imagine that we are trying to parse a
operation.

```
437 + 734
```

The lexer scans the text and find '4', '3', '7' and then the space ' '. The job of
recognize that the first characters constitute one token of type *NUM.* Then th
'+' symbol, which corresponds to a second token of type *PLUS*, and lastly it
token of type *NUM*.



The parser will typically combine the tokens produced by the lexer and group them.

The definitions used by lexers or parser are called *rules* or *productions*. A lexer rule will specify that a sequence of digits correspond to a token of type *NUM*, while a parser rule will specify that a sequence of tokens of type *NUM, PLUS, NUM* corresponds to an expression.

Scannerless parsers are different because they process directly the original text, instead of processing a list of tokens produced by a lexer.

It is now typical to find suites that can generate both a lexer and parser. In the past it was instead more common to combine two different tools: one to produce the lexer and one to produce the parser. This was for example the case of the venerable lex & yacc couple: lex produced the lexer, while yacc produced the parser.

Parse Tree And Abstract Syntax Tree

There are two terms that are related and sometimes they are used interchangeably: parse tree and Abstract SyntaxTree (AST).

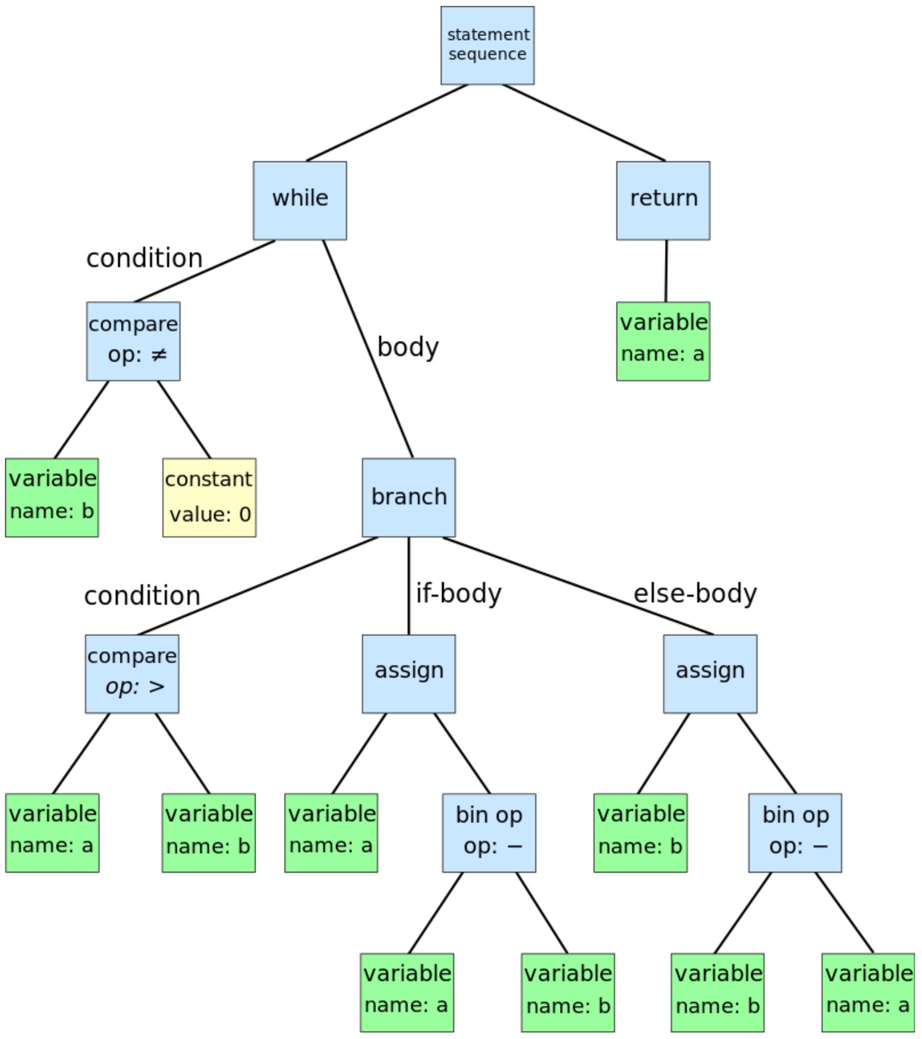
Conceptually they are very similar:

- they are both **trees**: there is a root representing the whole piece of code parsed. Then there are smaller subtrees representing portions of code that become smaller until single tokens appear in the tree
- the difference is the level of abstraction: the parse tree contains all the tokens which appeared in the program and possibly a set of intermediate rules. The AST instead is a polished version of the parse tree where the information that could be de important to understand the piece of code is removed

In the AST some information is lost, for instance comments and grouping sy (parentheses) are not represented. Things like comments are superfluous fo and grouping symbols are implicitly defined by the structure of the tree.

A parse tree is a representation of the code closer to the concrete syntax. It details of the implementation of the parser. For instance, usually a rule corre type of a node. A parse tree is usually transformed in an AST by the user, pc some help from the parser generator.

A graphical representation of an AST looks like this.



Sometimes you may want to start producing a parse tree and then derive fro
This can make sense because the parse tree is easier to produce for the pa
direct representation of the parsing process) but the AST is simpler and easi
by the following steps. By following steps we mean all the operations that yo
perform on the tree: code validation, interpretation, compilation, etc..

Grammar

> A grammar is a formal description of a language that can b
> recognize its structure.

In simple terms is a list of rules that define how each construct can be comp
example, a rule for an if statement could specify that it must starts with the "i
followed by a left parenthesis, an expression, a right parenthesis and a state

A rule could reference other rules or token types. In the example of the if sta
keyword "if", the left and the right parenthesis were token types, while expre
statement were references to other rules.

The most used format to describe grammars is the **Backus-Naur Form (BN**
has many variants, including the **Extended Backus-Naur Form**. The Extend
the advantage of including a simple way to denote repetitions. A typical rule
Naur grammar looks like this:

```
<symbol> ::= __expression__
```

The `<simbol>` is usually nonterminal, which means that it can be replaced by the group of elements on the right, `__expression__`. The element `__expression__` could contains other nonterminal symbols or terminal ones. Terminal symbols are simply the ones that do not appear as a `<symbol>` anywhere in the grammar. A typical example of a terminal symbol is a string of characters, like “class”.

Left-recursive Rules

In the context of parsers an important feature is the support for left-recursive rules. This means that a rule could start with a reference to itself. This reference could be also indirect.

Consider for example arithmetic operations. An addition could be described as two expression(s) separated by the plus (+) symbol, but an expression could also contain other additions.

```
addition       ::= expression '+' expression
multiplication ::= expression '*' expression
// an expression could be an addition or a multiplication or a number
expression     ::= addition | multiplication |// a number
```

This description also match multiple additions like 5 + 4 + 3. That is because it can be interpreted as expression (5) (‘+’) expression(4+3). And then 4 + 3 itself can be divided in its two components.

The problem is that this kind of rules may not be used with some parser generators. The alternative is a long chain of expressions that takes care also of the precedence of operators.

Some parser generators support direct left-recursive rules, but not indirect one.

Types Of Languages And Grammars

We care mostly about two types of languages that can be parsed with a pars
regular languages and *context-free language*s. We could give you the forma
according to the Chomsky hierarchy of languages
[https://en.wikipedia.org/wiki/Chomsky_hierarchy] , but it would not be that u
at some practical aspects instead.

A regular language can be defined by a series of regular expressions, while
one need something more. A simple rule of thumb is that if a grammar of a la
recursive elements it is not a regular language. For instance, as we said else
is not a regular language [https://tomassetti.me/antlr-mega-tutorial/] . In fact,
programming languages are context-free languages.

Usually to a kind of language correspond the same kind of grammar. That is
are regular grammars and context-free grammars that corresponds respecti
and context-free languages. But to complicate matters, there is a relatively n
2004) kind of grammar, called Parsing Expression Grammar (PEG). These g
as powerful as Context-free grammars, but according to their authors they d
programming languages more naturally.

The Differences Between PEG and CFG

The main difference between PEG and CFG is that the ordering of choices is
PEG, but not in CFG. If there are many possible valid ways to parse an inpu
ambiguous and thus wrong. Instead with PEG the first applicable choice will
and this automatically solve some ambiguities.

Another difference is that PEG use scannerless parsers: they do not need a separate lexer, or lexical analysis phase.

Traditionally both PEG and some CFG have been unable to deal with left-recursive rules, but some tools have found workarounds for this. Either by modifying the basic parsing algorithm, or by having the tool automatically rewrite a left-recursive rule in a non recursive way. Either of these ways has downsides: either by making the generated parser less intelligible or by worsen its performance. However, in practical terms, the advantages of easier and quicker development outweigh the drawbacks.

If you want to know more about the theory of parsing, you should read **A Guide to Parsing: Algorithms and Terminology** [https://tomassetti.me/guide-parsing-algorithms-terminology/] .

Parser Generators

The basic workflow of a parser generator tool is quite simple: you write a grammar that defines the language, or document, and you run the tool to generate a parser usable from your Python code.

The parser might produce the AST, that you may have to traverse yourself or you can traverse with additional ready-to-use classes, such Listeners [https://en.wikipedia.org/wiki/Observer_pattern] or Visitors [https://en.wikipedia.org/wiki/Visitor_pattern] . Some tools instead offer the chance to embed code inside the grammar to be executed every time the specific rule is matched.

Usually you need a runtime library and/or program to use the generated parser.

Context Free

Let's see the tools that generate Context Free parsers.

ANTLR

ANTLR [http://www.antlr.org/] is a great parser generator written in Java that generate parsers for Python and many other languages. There is also a beta TypeScript [https://github.com/tunnelvisionlabs/antlr4ts] from the same guy t *optimized C#* version. ANTLR is based on an new LL algorithm developed b and described in this paper: Adaptive LL(*) Parsing: The Power of Dynamic [http://www.antlr.org/papers/allstar-techreport.pdf] .

It is quite popular for its many useful features: for instance version 4 support recursive rules. However a real added value of a vast community it is the lar grammars available [https://github.com/antlr/grammars-v4] .

It provides two ways to walk the AST, instead of embedding actions in the gr and listeners. The first one is suited when you have to manipulate or interact elements of the tree, while the second is useful when you just have to do so rule is matched.

The typical grammar is divided in two parts: lexer rules and parser rules. The implicit, since all the rules starting with an uppercase letter are lexer rules, w starting with a lowercase letter are parser rules. Alternatively lexer and parse can be defined in separate files.

A very simple ANTLR grammar

```
grammar simple;

basic   : NAME ':' NAME ;
```

```

NAME    : [a-zA-Z]* ;

COMMENT : '/*' .*? '*/' -> skip ;
```

If you are interested to learn how to use ANTLR, you can look into this giant ANTLR tutorial [https://tomassetti.me/antlr-mega-tutorial] we have written. If you are ready to become a professional ANTLR developer, you can buy our video course to **Build professional parsers and languages using ANTLR [https://tomassetti.me/antlr-course/]** . The course is taught using Python, so you will feel right at home.

Lark

> A modern parsing library for Python, implementing Earley & LALR(1) and an easy interface

Lark [https://github.com/erezsh/lark] is a parser generator that works as a library. You write the grammar in a string or a file and then use it as an argument to dynamically generate the parser. Lark can use two algorithms: Earley is used when you need to parse all grammars and LALR when you need speed. Earley can parse also ambiguous grammars. Lark offers the chance to automatically solve the ambiguity by choosing the simplest option or reporting all options.

Lark grammars are written in an EBNF format. They cannot include actions. This means that they are clean and readable, but also that you have to traverse the resulting tree yourself. Although there is a function that can help with that if you use the LALR algorithm. On the positive side you can also use specific notations in the grammar to automatically generate an AST. You can do that by dropping certain nodes, merging or transforming them.

The following example grammar shows a useful feature of Lark: it includes rules for common things, like whitespace or numbers.

Lark Example

```
parser = Lark('''?sum: product
                | sum "+" product   -> add
                | sum "-" product   -> sub

            ?product: item
                | product "*" item  -> mul
                | product "/" item  -> div

            ?item: NUMBER           -> number
                 | "-" item         -> neg
                 | "(" sum ")"

            %import common.NUMBER
            %import common.WS
            %ignore WS
        ''', start='sum')
```

Lark comes with a tool to convert Nearley grammars in its own format. It als
useful function to transform the tree generated by the parser in an image.

It has a sufficient documentation, with examples and tutorials available. Ther
small reference.

Lrparsing

> lrparsing [http://lrparsing.sourceforge.net/doc/html/] is an L
> parser hiding behind a pythonic interface

Lrparsing is a parser generator whose grammars are defined as Python exp
These expressions are attribute of a class that corresponds to rule of a tradi

They are usually dynamically generated, but the library provide a function to precompile a parse table beforehand.

Given their format depending on Python, lrparsing grammars can be easy to read for Python developers, but they are harder to read than a traditional grammar.

Example lrparsing Python

```python
// from the documentation
class ExprParser(lrparsing.Grammar):
    #
    # Put Tokens we don't want to re-type in a TokenRegistry.
    #
    class T(lrparsing.TokenRegistry):
        integer = Token(re="[0-9]+")
        integer["key"] = "I'm a mapping!"
        ident = Token(re="[A-Za-z_][A-Za-z_0-9]*")
    #
    # Grammar rules.
    #
    expr = Ref("expr")                # Forward reference
    call = T.ident + '(' + List(expr, ',') + ')'
    atom = T.ident | T.integer | Token('(') + expr + ')' | call
    expr = Prio(                      # If ambiguous choose atom 1st,
 ...
        atom,
        Tokens("+ - ~") >> THIS,      # >> means right associative
        THIS << Tokens("* / // %") << THIS,
        THIS << Tokens("+ -") << THIS,# THIS means "expr" here
        THIS << (Tokens("== !=") | Keyword("is")) << THIS)
    expr["a"] = "I am a mapping too!"
    START = expr                      # Where the grammar must start
    COMMENTS = (                      # Allow C and Python comments
        Token(re="#(?:[^\r\n]*(?:\r\n?|\n\r?))") |
        Token(re="/[*](?:[^*]|[*][^/])*[*]/"))
```

Lrparsing also provide some basic functions to print parsing tree and grammar rules for debugging purposes.

The documentation is really good: it explains everything you need to know about the library and it also provide some guidance on creating good grammars (eg. solving ambiguities). There are also quite complex example grammars, like one for SQLite.

PLY

> PLY [https://github.com/dabeaz/ply] doesn't try to do anyth or less than provide the basic lex/yacc functionality. In othe it's not a large parsing framework or a component of some system.

PLY is a stable and maintained tool with a long history starting from 2001. It basic, given that there are no tools for automatic creation of AST, or anything developer of the previous century would define as *fancy stuff*. The tool was created as instructional tool. This explains its simplicity, but it also the reaso offer great support for diagnostics or catching mistakes in the grammar.

A PLY grammar is written in Python code in a BNF-like format. Lexer and pa can be used separately. The following example shows only the lexer, but the in the same way.

PLY example with only the lexer

```python
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

The documentation [http://www.dabeaz.com/ply/ply.htm] is extensive, clear, with abundant examples and explanations of parsing concepts. All that you need, if you can get pass the '90 looks.

There is a port for RPython called RPLY [https://github.com/alex/rply].

PlyPlus

> Plyplus is a general-purpose parser built on top of PLY [http://www.dabeaz.com/ply/] (LALR(1)), and written in Pytl
> Plyplus features a modern design, and focuses on simplici
> losing power.

PlyPlus is a tool that is built on top of PLY, but it is very different from it. The the way the names are written are different. Compared to its father the docu lacking, but the features are many.

You can write a grammar in a `.g` file or in a string, but it is always generate The format is based on EBNF, but a grammar can also include special notati the creation of an AST. This notation allows to exclude or drop certain rules f generated tree.

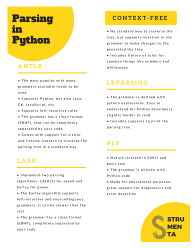
Example calc.g

```
// from the documentation
start: add;

// Rules
?add: (add add_symbol)? mul;
?mul: (mul mul_symbol)? atom;
// rules preceded by @ will not appear in the tree
@atom: neg | number | '\(' add '\)';
neg: '-' atom;

// Tokens
number: '[\d.]+';
```

Cheatsheet of Parsing in Python


Parsing in Python
CONTEXT-FREE
ANTLR
LRPARSING
PLY
LARK

A cheatsheet that sums up the strong and weak points of each parsing tool and tells you which tools to use for each parsing need

Sign up to download this FREE Cheatsheet

Email Address

GET THE CHEATSHEET

```
mul_symbol: '\*' | '/';
add_symbol: '\+' | '-';

WS: '[ \t]+' (%ignore);
```

PlyPlus include a function to draw an image of a parse tree based upon pydot and graphviz. PlyPlus has unique features, too. It allows you to select nodes in the AST using selectors similar to the CSS selectors used in web development. For instance, if you want to fill all terminal nodes that contain the letter 'n', you can find them like this:

Selectors example
```
// from the documentation
>>> x.select('/.*n.*/:is-leaf')
['Popen', 'isinstance', 'basestring', 'stdin']
```

This is a unique feature that can be useful, for example, if you are developing a static analysis or refactoring tool.

Pyleri

Python Left-Right Parser [https://github.com/transceptor-technology/pyleri] (pyleri) is part of a family of similar parser generators for JavaScript, Python, C, Go and Java.

A grammar for Pyleri must be defined in Python expressions that are part of a class. Once it is defined, the grammar can be exported as a file defining the grammar in Python or any other supported language. For example, you can define the grammar in Python, export it to JacaScript and then use the JavaScript version of pyleri to run it. You cannot do the inverse, i.e., you cannot create a grammar in JavaScript and export it to Python. So, even if you want to use another language, it is better to create the grammar in Python and then export it to that language.

Apart from this interesting feature, Pyleri is a simple and easy to use tool.

Pyleri example Python
```
# from the documentation
# Create a Grammar Class to define your language
class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    START = Sequence(k_hi, r_name)

# Compile your grammar by creating an instance of the Gramma
my_grammar = MyGrammar()

# Use the compiled grammar to parse 'strings'
print(my_grammar.parse('hi "Iris"').is_valid) # => True
print(my_grammar.parse('bye "Iris"').is_valid) # => False
```

In practical terms there are two kinds of parsing rules: simple and combinati ones. The simple ones are essentially tokens created with regular expressio complex ones are created using ready-to-use parsing functions (e.g., Seque sequence of elements).

So, it is a cross between a parser generator and a parser combinator. Howe powerful that a traditional parser combinator and can also generate a parse neat feature is that it provide a property `expecting`, that list the elements accept at that particular position. This is very useful if you are building auto- functionality.

This mixture of simplicity of syntax and powerful features can quite attractive something powerful, but are not used to a traditional parser generator.

PEG

After the CFG parsers is time to see the PEG parsers available for Python.

Arpeggio

> Arpeggio [http://www.igordejanovic.net/Arpeggio/] is recursive descent parser with backtracking and memoization (a.k.a. pacrat parser). Arpeggio grammars are based on PEG formalism.

The documentation defines Arpeggio as a parser interpreter, since parser are generated dynamically from a grammar. In any case it does not work any different from many other Python parser generators. A peculiarity of Arpeggio is that you can define a grammar in a textual PEG format or using Python expressions. Actually, there are two dialects of PEGs, one with a cleaner Python-like syntax and the other the traditional PEG one.

Arpeggio generate a simple parse tree, but it supports the use of a visitor. The visitor can also include a second action to perform after all the tree nodes have been processed. This is used for post-processing, for instance it can be used to deal with symbol reference.

An Arpeggio grammar defined with either a PEG notation or the Python one is usually quite readable. The following example uses Python notation.

Arpeggio example Python

```python
# partial example from the documentation
def record():                   return field, ZeroOrMore(",", field)
def field():                    return [quoted_field, field_content]
def quoted_field():             return '"', field_content_quoted, '"'
def field_content():            return _(r'([^,\n])+')
def field_content_quoted():     return _(r'(("")|([^"]))+')
def csvfile():                  return OneOrMore([record, '\n']), EOF

[..]

def main(debug=False):
    # First we will make a parser - an instance of the CVS parser model.
    # Parser model is given in the form of python constructs therefore w
e
    # are using ParserPython class.
    # Skipping of whitespace will be done only for tabs and spaces. Newl
ines
    # have semantics in csv files. They are used to separate
parser = ParserPython(csvfile, ws='\t ', debug=debug)

[..]
```

There are a couple of options for debugging: verbose and informative ouput generation of DOT files of the parser. The DOT files can be used for creating of the parser, but you will have to call `graphviz` yourself. The documentati comprehensive and well-organized.

Arpeggio is the foundation of a more advanced tool for the creation of DSL c [https://github.com/igordejanovic/textX] . TextX is made by the same develop Arpeggio and it is inspired by the more famous XText.

Canopy

> Canopy [http://canopy.jcoglan.com/] is a parser compiler ta Java, JavaScript, Python and Ruby. It takes a file describin parsing expression grammar and compiles it into a parser the target language. The generated parsers have no runtin dependency on Canopy itself.

It also provides easy access to the parse tree nodes.

A Canopy grammar has the neat feature of using actions annotation to use c the parser. In practical terms. you just write the name of a function next to a you implement the function in your source code.

```
A Canopy grammar with actions
// the actions are prepended by %
grammar Maps
  map     <-  "{" string ":" value "}" %make_map
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

The Python file containing the action code.

```
class Actions(object):
    def make_map(self, input, start, end, elements):
        return {elements[1]: elements[3]}

   [..]
```

Parsimonious

> Parsimonious [https://github.com/erikrose/parsimonious] aims to be the fastest arbitrary-lookahead parser written in pure Python—and the most usable. It's based on parsing expression grammars (PEGs), which means you feed it a simplified sort of EBNF notation.

Parsimonious is a no-nonsense tool designed for speed and low usage of RAM. It is also a no-documentation tool, there are not even complete examples. Actually the short README file explain the basics and redirect you to Docstring [https://it.wikipedia.org/wiki/Docstring] for more specific information.

In any case Parsimonious is good working tool that allows you dynamically create a grammar defined in a file or a string. You can also define a visitor to traverse and transform the parsing tree. So, if you are already familiar with the PEG format you do not need to know anything else to use it at its fullest.

A Parsimonious grammar is readable like any other PEG grammar.

```
Parsimonious example
# example from the documentation
my_grammar = Grammar(r"""
    styled_text = bold_text / italic_text
    bold_text   = "((" text "))"
    italic_text = "''" text "''"
    text        = ~"[A-Z 0-9]*"i
    """)
```

pyPEG

> *pyPEG* [https://fdik.org/pyPEG/index.html] is a plain and si
> intrinsic parser interpreter framework for Python version 2.

PyPEG is a framework to parse and compose text. Which means that you de
grammar in a syntax as powerful as PEG, but you do it in Python code. And
this grammar to parse and/or compose a text based upon that grammar. Ob
compose a text you have to provide the data yourself. In this case it works a
system.

The syntax for a PyPEG is on the verbose side, frankly it is too verbose to b
you just want to use it for simple parsing. But it is a cool library if you want to
manipulate some document in a specific format. For instance, you could use
documentation in one format to another.

```
A pyPEG example
# from the documentation
```

```
from pypeg2 import *
class Type(Keyword):
grammar = Enum( K("int"), K("long") )

class Parameter:
grammar = attr("typing", Type), name()

class Parameters(Namespace):
grammar = optional(csl(Parameter))

class Instruction(str):
grammar = word, ";"

block = "{", maybe_some(Instruction), "}"
class Function(List):
grammar = attr("typing", Type), name(), \
        "(", attr("parms", Parameters), ")", block

f = parse("int f(int a, long b) { do_this; do_that; }", Function)
```

PyPEG does not produce a standard tree, but a structure based upon the defined grammar. Look at what happens for the previous example.

```
# execute the example
>>> f.name
Symbol('f')
>>> f.typing
Symbol('int')
>>> f.parms["b"].typing
Symbol('long')
>>> f[0]
'do_this'
>>> f[1]
'do_that'
```

TatSu

> TatSu (for grammar compiler) is a tool that takes grammars in a variation of EBNF as input, and outputs memoizing (Packrat) PEG parsers in Python.

TatSu is the successor of Grako, another parser generator tool, and it has a compatibility [http://tatsu.readthedocs.io/en/stable/grako.html] with it. It can c dynamically from a grammar or compiling into a Python module.

TatSu generate PEG parsers, but grammars are defined in a variant of EBNI order of rules matters as it is usual for PEG grammars. So it is actually a sor between the two. This variant includes support for dealing with associativity the generated tree or model (more on that later). Support for left-recursive ru but experimental.

calc.ebnf

```
// TatSu example grammar from the tutorial
@@grammar::CALC

start
    =
    expression $
    ;

expression
    =
    | expression '+' term
    | expression '-' term
    | term
    ;

term
    =
    | term '*' factor
    | term '/' factor
    | factor
    ;

factor
    =
    | '(' expression ')'
    | number
    ;

number
    =
    /\d+/
```

TatSu grammars cannot include actions, that can be defined in a separate Python class. Instead you have to annotate the grammar if you want to use an object model in place of semantic actions. An object model is a way to separate the parsing process from the entity that is parsed. In practical terms instead of doing something when a certain rule is matched you do something when a certain object is defined. This object may be defined by more than one rule.

The following extract example defines an object **Multiply** that corresponds to the rule **multiplication**.

```
multiplication::Multiply
    =
    left:factor op:'*' ~ right:term
    ;
```

The object model can then be used for what TatSu calls *walker* (essentially a visitor for the model).

TatSu example Python

```python
from tatsu.walkers import NodeWalker

class CalcWalker(NodeWalker):
    def walk_object(self, node):
        return node

    def walk__add(self, node):
        return self.walk(node.left) + self.walk(node.right)

    def walk__subtract(self, node):
        return self.walk(node.left) - self.walk(node.right)

    def walk__multiply(self, node):
        return self.walk(node.left) * self.walk(node.right)

    def walk__divide(self, node):
        return self.walk(node.left) / self.walk(node.right)

def parse_and_walk_model():
    grammar = open('grammars/calc_model.ebnf').read()

    parser = tatsu.compile(grammar, asmodel=True)
    model = parser.parse('3 + 5 * ( 10 - 20 )')

    print('# WALKER RESULT IS:')
    print(CalcWalker().walk(model))
    print()
```

The same object model can also be used for code generation, for instance t
format into another one. But for that you obviously cannot reuse the walker,
define a template class for each object.

TatSu provides also: a tool to translate ANTLR grammars, complex trace ou
graphical representation of the tree using pygraphviz. ANLTR grammar may
manually adapted to respect PEG constraints.

The documentation is complete: it shows all the features, provide examples
basic introduction to parsing concepts, like AST.

Cheatsheet of Parsing in Python

A cheatsheet that sums up the strong and weak points of each parsing tool and tells you which tools to use for each parsing need

Sign up to download this FREE Cheatsheet

Email Address

GET THE CHEATSHEET

Waxeye

> Waxeye [https://waxeye.org/] is a parser generator based on parsing expression grammars (PEGs). It supports C, Java, Javascript, Python, Ruby and Scheme.

Waxeye can facilitate the creation of an AST by defining nodes in the grammar that will not be included in the generated tree. That is quite useful, but a drawback of Waxeye is that it only generates a AST. In the sense that there is no way to automatically execute an action when you match a node. You have to traverse and execute what you need manually.

One positive side-effect of this limiation is that grammars are easily readable and clean. They are also independent from any language.

Calc.waxeye

```
// from the manual

calc  <- ws sum

sum   <- prod *([+-] ws prod)

prod  <- unary *([*/] ws unary)

unary <= '-' ws unary
       | :'(' ws sum :')' ws
       | num

num   <- +[0-9] ?('.' +[0-9]) ws

ws    <: *[ \t\n\r]
```

A particular feature of Waxeye is that it provides some help to compose different grammars together and then it facilitates modularity. For instance, you could create a common grammar for identifiers, that are usually similar in many languages.

Waxeye has a great documentation in the form of a manual that explains basic concepts and how to use the tool for all the languages it supports. There are a few exa grammars.

Parser Combinators

They allow you to create a parser by combining different pattern matching fu are equivalent to grammar rules. They are generally considered best suited parsing needs.

In practice this means that they are very useful for all the little parsing proble the typical developer encounters a problem, that is too complex for a simple expression, these libraries are usually the solution. In short, if you need to bu but you don't actually want to, a parser combinator may be your best option.

Some readers have indicated us funcparserlib [https://github.com/vlasovskik , but we decided to not include it because it has been unmantained for a few

Parsec.py, Parsy and Pyparsing

> A universal Python parser combinator library inspired by P library of Haskell.

That is basically the extent of the documentation on Parsec.py [https://github.com/sighingnow/parsec.py] . Though there are a couple of exa already know how to use the original Parsec library or one of its many clones you can try to use it. It does not look bad, but the lack of documentation is a problem for new users.

> Parsy [https://github.com/python-parsy/parsy] is an easy way to combine simple, small parsers into complex, larger parsers. If it means anything to you, it's a monadic parser combinator library for LL(infinity) grammars in the spirit of Parsec, Parsnip, and Parsimmon.

Parsy was an abandoned project for a while, but it was recently recovered and taken up by a new maintainer and it is now in a good shape. Among other things the new developer brought the project to recent coding practices (e.g., testing coverage).

The project might not be as powerful as an "industrial-strength" parser combinator such Parsec (the original one), but it has a few nice features. For instance, you can create a generator function to create a parser [http://parsy.readthedocs.io/en/latest/ref/generating.html] . It now requires Python 3.3 or later, which should only be a problem for people stuck with Python 2.

The project now has ample documentation, examples and a tutorial. The following example comes from the documentation and shows how to parse a date.

Parsy example Python
```python
# from the documentation
# parsing a date
from parsy import string, regex
from datetime import date
ddmmyy = regex(r'[0-9]{2}').map(int).sep_by(string("-"), min=3, max=3).combine(
               lambda d, m, y: date(2000 + y, m, d))
ddmmyy.parse('06-05-14')
```

> The pyparsing [http://pyparsing.wikispaces.com/home] module is an alternative approach to creating and executing simple gram
> the traditional lex/yacc approach, or the use of regular exp
> The pyparsing module provides a library of classes that cli
> uses to construct the grammar directly in Python code.

Pyparsing is a stable and mature software developed for more than 14 years many examples [http://pyparsing.wikispaces.com/Examples] , but still a conf lacking documentation. While Pyparsing is as equally powerful as a tradition combinator, it works a bit differently and this lack in the proper documentatio frustrating.

However, if you take the time to learn on its own, the following example shov easy to use.

greeeting.py
```python
# example from the documentation
# define grammar
greet = Word( alphas ) + "," + Word( alphas ) + "!"

# input string
hello = "Hello, World!"

# parse input string
print hello, "->", greet.parseString( hello )
```

Python Libraries Related to Parsing

Python offers also some other libraries or tools related to parsing.

Cheatsheet of Parsing in Python

A cheatsheet that sums up the strong and weak points of each parsing tool and tells you which tools to use for each parsing need

Sign up to download this FREE Cheatsheet

Email Address

GET THE CHEATSHEET

Parsing Python Inside Python

There is one special case that could be managed in more specific way: the case in which you want to parse Python code in Python. When it comes to Python the best choice is to rely on your own Python interpreter.

The standard reference implementation of Python, known as CPython, include a few modules to access its internals for parsing: tokenize [https://docs.python.org/release/3.6.1/library/tokenize.html] , parser [https://docs.python.org/release/3.6.1/library/parser.html] and ast [https://docs.python.org/release/3.6.1/library/ast.html] . You may also be able to use the parser in the PyPy interpreter [http://doc.pypy.org/en/latest/parser.html] .

Parsing with Regular Expressions and The Like

Usually you resort to parsing libraries and tools when regular expression are not enough. However, there is a good library for Python than can extend the life and usefulness of regular expressions or using elements of similar complexity.

> Regular Expression based parsers for extracting data from natural languages [..]
>
> This library basically just gives you a way to combine Regular Expressions together and hook them up to some callback functions in Python.

Reparse [http://reparse.readthedocs.io/en/latest/] is a simple tool that can nonetheless quite useful in certain scenarios. The author himself says that it is much simpler and with less feature than PyParsing or Parboiled.

The basic idea is that you define regular expressions, the patterns in which they can combine and the functions that are called when an expression or pattern is found. You must define functions in Python, but expressions and pattern can be defined in Ya
Python.

In this example from the documentation expressions and patterns are define

```
expressions.yaml
Color:
    Basic Color:
        Expression: (Red|Orange|Yellow|Green|Blue|Violet|Bro
        Matches: Orange | Green
        Non-Matches: White
        Groups:
          - Color

Time:
    Basic Time:
        Expression: ([0-9]|[1][0-2]) \s? (am|pm)
        Matches: 8am | 3 pm
        Non-Matches: 8a | 8:00 am | 13pm
        Groups:
          - Hour
          - AMPM
```

Fields like Matches are there for humans, but can be used for testing by Rep

```
patterns.yaml
BasicColorTime:
  Order: 1
  Pattern: |
    <Color> \s? at \s? <Time>
  # Angle brackets detonate expression groups
  # Multiple expressions in one group are combined together
```

An example function in Python for the pattern.

```
reparse_functions.py                                                Python
from datetime import time
def color_time(Color=None, Time=None):
    Color, Hour, Period = Color[0], int(Time[0]), Time[1]
    if Period == 'pm':
        Hour += 12
    Time = time(hour=Hour)

    return Color, Time

functions = {
   'BasicColorTime' : color_time,
}
```

The file that puts everything together.

```
Reparse_example.py                                                  Python
from reparse_functions import functions
import reparse

colortime_parser = reparse.parser(
    parser_type=reparse.basic_parser,
    expressions_yaml_path=path + "expressions.yaml",
    patterns_yaml_path=path + "patterns.yaml",
    functions=functions
)

print(colortime_parser("~ ~ ~ go to the store ~ buy green at 11pm! ~ ~")
)
```

Parsing Binary Data: Construct

> Instead of writing *imperative code* to parse a piece of data, you declaratively define a *data structure* that describes your data. As this data structure is not code, you can use it in one direction to *parse* data into Pythonic objects, and in the other direction, to *build* objects into binary data.

And that is it: Construct [https://construct.readthedocs.io/en/latest/] . You cou
data even with some parser generators (e.g. ANTLR), but Constuct make it
is a sort of DSL combined with a parser combinator to parse binary formats.
bunch of fields to manage binary data: apart from the obvious ones (e.g. floa
etc.), there are a few specialized to manage sequences of fields (sequence)
(struct) and a few conditional statements.

It also makes available functions to adapt or validate (test) the data and deb
you found.

As you can see in the following example, it is quite easy to use.

```
Construct GIF example
# from the documentation

gif_logical_screen = Struct("logical_screen",
    ULInt16("width"),
    ULInt16("height"),
    [..]
    If(lambda ctx: ctx["flags"]["global_color_table"],
        Array(lambda ctx: 2**(ctx["flags"]["global_color_tab
),
            Struct("palette",
                ULInt8("R"),
                ULInt8("G"),
                ULInt8("B")
            )
        )
    )
)

gif_header = Struct("gif_header",
    Const("signature", b"GIF"),
    Const("version", b"89a"),
)

[..]

gif_file = Struct("gif_file",
    gif_header,
    gif_logical_screen,
    [..]
)

if __name__ == "__main__":
    f = open("../../../tests/sample.gif", "rb")
    s = f.read()
    f.close()
    # if you want to build the file, you just have to provide the data
    # to the build() function
    print(gif_file.parse(s))
```

There is a nice amount of documentation and even many example grammars for different kinds of format [https://github.com/construct/construct/tree/master/construct/examples/formats] , such as filesystems or graphics files.

Summary

Any programming language has a different community with its peculiarities. These differences remains even when we compare the same interests across the languages. For instance, when we compare parsers tools we can see how Java and Python developers live in a different world.

The parsing tools and libraries for Python for the most part use very readable grammars and are simple to use. But the most interesting thing is that they cover a very wide spectrum of competence and use cases. There seems to be an uninterrupte available from regular expression, passing through Reparse to end with TatS

Sometimes this means that it can be confusing, if you are a parsing expert c different language. Because few parser generators actually generate parsers mostly interpret them at runtime. On the other hand, with Python you can rea perfect library, or tools, for your needs. And to help you with that we hope tha comparison has been useful for you.

Tags: Python

You might also like

DO YOU NEED A PARSER?

We can design parsers for new languages, or rewrite parsers for existing languages built in house.

On top of parsers we can then help building interpreters, compilers, code generators, documentation generators, or translators (code converters) to other languages.

Let's Talk!

Cheatsheet of Parsing in Python

A cheatsheet that sums up the strong and weak points of each parsing tool and tells you which tools to use for each parsing need

Sign up to download this FREE Cheatsheet

Email Address

GET THE CHEATSHEET