parsy Documentation

Release 1.3.0

Jeanine Adkisson

Aug 03, 2019

Contents:

These are the docs for parsy 1.3.0. Check the *History and release notes* for significant changes.

CHAPTER 1

Installation

parsy can be installed with pip:

```
pip install parsy
```

Python 3.3 or greater is required.

CHAPTER 2

Overview

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.

If that means nothing, rest assured that parsy is a very straightforward and Pythonic solution for parsing text that doesn't require knowing anything about monads.

Parsy differentiates itself from other solutions with the following:

- it is not a parser generator, but a combinator based parsing library.
- a very clean implementation, only a few hundred lines, that borrows from the best of recent combinator libraries.
- free, good quality documentation, all in one place. (Please raise an issue on GitHub if you have any problems, or find the documentation lacking in any way).
- it avoids mutability, and therefore a ton of related bugs.
- it has monadic binding with a *nice syntax*. In plain English:
  - we can easily handle cases where later parsing depends on the value of something parsed earlier e.g. Hollerith constants.
  - it's easy to build up complex result objects, rather than returning lists of lists etc. which then need to be further processed.
  - there is no need for things like pyparsing's Forward class .
- it has a minimalist philosophy. It doesn't include built-in helpers for any specific grammars or languages, but provides building blocks for making these.

Basic usage looks like this:

Example 1 - parsing a set of alternatives:

```
>>> from parsy import string
>>> parser = (string('Dr.') | string('Mr.') | string('Mrs.')).desc("title")
>>> parser.parse('Mrs.')
```

(continues on next page)

(continued from previous page)

```
'Mrs.'
>>> parser.parse('Mr.')
'Mr.'

>>> parser.parse('Joe')
ParseError: expected title at 0:0

>>> parser.parse_partial('Dr. Who')
('Dr.', ' Who')
```

Example 2 - Parsing a dd-mm-yy 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')
datetime.date(2014, 5, 6)
```

To learn how to use parsy, you should continue with:

- the *tutorial*, especially if you are not familiar with this type of parser library.
- the *parser generator decorator*
- the *builtin parser primitives*
- the *method and combinator reference*

2.1 Other Python projects

- pyparsing. Also a combinator approach, but in general much less cleanly implemented, and rather scattered documentation, although it has more builtin functionality in terms of provided utilities for certain parsing tasks.
- PLY. A pure Python implementation of the classic lex/yacc parsing tools. It is well suited to large grammars that would be found in typical programming languages.
- funcparserlib - the most similar to parsy. It differs from parsy mainly in normally using a separate tokenization phase, lacking the convenience of the `generate()` method for creating parsers, and documentation that relies on understanding Haskell type annotations.
- Lark. With Lark you write a grammar definition in a separate mini-language as a string, and have a parser generated for you, rather than writing the grammar in Python. It has the advantage of speed and being able to use different parsing algorithms.

CHAPTER 3

Tutorial

First *install parsy*, and check that the documentation you are reading matches the version you just installed.

3.1 Building an ISO 8601 parser

In this tutorial, we are going to gradually build a parser for a subset of an ISO 8601 date. Specifically, we want to handle dates that look like this: `2017-09-25`.

A problem of this size could admittedly be solved fairly easily with regexes. But very quickly regexes don't scale, especially when it comes to getting the parsed data out, and for this tutorial we need to start with a simple example.

With parsy, you start by breaking the problem down into the smallest components. So we need first to match the 4 digit year at the beginning.

There are various ways we can do this, but a regex works nicely, and *`regex()`* is a built-in primitive of the parsy library:

```
>>> from parsy import regex
>>> year = regex(r'[0-9]{4}')
```

(For those who don't know regular expressions, the regex `[0-9]{4}` means "match any character from 0123456789 exactly 4 times".)

This has produced a *`Parser`* object which has various methods. We can immediately check that it works using the *`Parser.parse()`* method:

```
>>> year.parse('2017')
'2017'
>>> year.parse('abc')
ParseError: expected '[0-9]{4}' at 0:0
```

Notice first of all that a parser consumes input (the value we pass to `parse`), and it produces an output. In the case of `regex`, the produced output is the string that was matched, but this doesn't have to be the case for all parsers.

If there is no match, it raises a `ParseError`.

Notice as well that the *Parser.parse()* method expects to consume all the input, so if there are extra characters at the end, even if it is just whitespace, parsing will fail with a message saying it expected EOF (End Of File/Data):

```
>>> year.parse('2017 ')
ParseError: expected 'EOF' at 0:4
```

You can use *Parser.parse_partial()* if you want to just keep parsing as far as possible and not throw an exception.

To parse the data, we need to parse months, days, and the dash symbol, so we'll add those:

```
>>> from parsy import string
>>> month = regex('[0-9]{2}')
>>> day = regex('[0-9]{2}')
>>> dash = string('-')
```

We've added use of the *string()* primitive here, that matches just the string passed in, and returns that string.

Next we need to combine these parsers into something that will parse the whole date. The simplest way is to use the *Parser.then()* method:

```
>>> fulldate = year.then(dash).then(month).then(dash).then(day)
```

The `then` method returns a new parser that requires the first parser to succeed, followed by the second parser (the argument to the method).

We could also write this using the *>> operator* which does the same thing as *Parser.then()*:

```
>>> fulldate = year >> dash >> month >> dash >> day
```

This parser has some problems which we need to address, but it is already useful as a basic validator:

```
>>> fulldate.parse('2017-xx')
ParseError: expected '[0-9]{2}' at 0:5
>>> fulldate.parse('2017-01')
ParseError: expected '-' at 0:7
>>> fulldate.parse('2017-02-01')
'01'
```

If the parse doesn't succeed, we'll get `ParseError`, otherwise it is valid (at least as far as the basic syntax checks we've added).

The first problem with this parser is that it doesn't return a very useful value. Due to the way that *Parser.then()* works, when it combines two parsers to produce a larger one, the value from the first parser is discarded, and the value returned by the second parser is the overall return value. So, we end up getting only the 'day' component as the result of our parse. We really want the year, month and day packaged up nicely, and converted to integers.

A second problem is that our error messages are not very friendly.

Our first attempt at fixing these might be to use the *+ operator* instead of `then`. This operator is defined to combine the results of the two parsers using the normal plus operator, which will work fine on strings:

```
>>> fulldate = year + dash + month + dash + day
>>> fulldate.parse('2017-02-01')
'2017-02-01'
```

However, it won't help us if we want to split our data up into a set of integers.

Our first step should actually be to work on the year, month and day components using *Parser.map()*, which allows us to convert the strings to other objects - in our case we want integers.

We can also use the *Parser.desc()* method to give nicer error messages, so our components now look this this:

```
>>> year = regex('[0-9]{4}').map(int).desc('4 digit year')
>>> month = regex('[0-9]{2}').map(int).desc('2 digit month')
>>> day = regex('[0-9]{2}').map(int).desc('2 digit day')
```

We get better error messages now:

```
>>> year.then(dash).then(month).parse('2017-xx')
ParseError: expected '2 digit month' at 0:5
```

Notice that the `map` and `desc` methods, like all similar methods on `Parser`, return new parser objects - they do not modify the existing one. This allows us to build up parsers with a 'fluent' interface, and avoid problems caused by mutating objects.

However, we still need a way to package up the year, month and day as separate values.

The *seq()* combinator provides one easy way to do that. It takes the parsers that are passed in as arguments, and combines their results into a list:

```
>>> fulldate = seq(year, dash, month, dash, day)
>>> fulldate.parse('2017-01-02')
[2017, '-', 1, '-', 2]
```

Now, we don't need those dashes, so we can eliminate them using the *>> operator* or *<< operator*:

```
>>> fulldate = seq(year, dash >> month, dash >> day)
>>> fulldate.parse('2017-01-02')
[2017, 1, 2]
```

At this point, we could also convert this to a date object if we wanted using *Parser.combine()*:

```
>>> from datetime import date
>>> fulldate = seq(year, dash >> month, dash >> day).combine(date)
```

We could have used *Parser.map()* here, but *Parser.combine()* is a bit nicer. It's especially succinct because the argument order to `date` matches the order of the values parsed (year, month, day), otherwise we could have passed a `lambda` to `combine`, or used *Parser.combine_dict()*.

3.2 Using previously parsed values

Now, sometimes we might want to do more complex logic with the values that are collected as parse results, and do so while we are still parsing.

To continue our example, the above parser has a problem that it will raise an exception if the day and month values are not valid. We'd like to be able to check this, and produce a parse error instead, which will make our parser play better with others if we want to use it to build something bigger.

Also, in ISO8601, strictly speaking you can just write the year, or the year and the month, and leave off the other parts. We'd like to handle that by returning a tuple for the result, and `None` for the missing data.

To do this, we need to allow the parse to continue if the later components (with their leading dashes) are missing - that is, we need to express optional components, and we need a way to be able to test earlier values while in the middle of parsing, to see if we should continue looking for another component.

The *Parser.bind()* method provides one way to do it (yay monads!). Unfortunately, it gets ugly pretty fast, and in Python we don't have Haskell's `do` notation to tidy it up. But thankfully we can use generators and the `yield` keyword to great effect.

We use a generator function and convert it into a parser by using the *generate()* decorator. The idea is that you `yield` every parser that you want to run, and receive the result of that parser as the value of the yield expression. You can then put parsers together using any logic you like, and finally return the value.

An equivalent parser to the one above can be written like this:

```
@generate
def full_date():
    y = yield year
    yield dash  # implicit skip, since we do nothing with the value
    m = yield month
    yield dash
    d = yield day
    return date(y, m, d)
```

This is more verbose than before, but provides a good starting point for our next set of requirements.

First of all, we need to express optional components - that is we need to be able to handle missing dashes, and return what we've got so far rather than failing the whole parse.

Parser has a set of methods that convert parsers into ones that allow multiples of the parser - including *Parser.many()*, *Parser.times()*, *Parser.at_most()* and *Parser.at_least()*. There is also *Parser.optional()* which allows matching zero times (in which case the parser will return `None`), or exactly once - just what we need in this case.

We also need to do checking on the month and the day. We'll take a shortcut and use the built-in `datetime.date` class to do the validation for us. However, rather than allow exceptions to be raised, we convert the exception into a parsing failure.

```
optional_dash = dash.optional()

@generate
def full_or_partial_date():
    d = None
    m = None
    y = yield year
    dash1 = yield optional_dash
    if dash1 is not None:
        m = yield month
        dash2 = yield optional_dash
        if dash2 is not None:
            d = yield day
    if m is not None:
       if m < 1 or m > 12:
           return fail("month must be in 1..12")
    if d is not None:
       try:
           datetime.date(y, m, d)
       except ValueError as e:
           return fail(e.args[0])

    return (y, m, d)
```

This works now works as expected:

```
>>> full_or_partial_date.parse('2017-02')
(2017, 2, None)
>>> full_or_partial_date.parse('2017-02-29')
ParseError: expected 'day is out of range for month' at 0:10
```

We could of course use a custom object in the final line to return a more convenient data type, if wanted.

3.3 Alternatives and backtracking

Suppose we are using our date parser to scrape dates off articles on a web site. We then discover that for recently published articles, instead of printing a timestamp, they write "X days ago".

We want to parse this, and we'll use a timedelta object to represent the value (to easily distinguish it from other values and consume it later). We can write a parser for this easily:

```
>>> days_ago = regex("[0-9]+").map(lambda d: timedelta(days=-int(d))) << string("␣
↪days ago")
>>> days_ago.parse("5 days ago")
datetime.timedelta(-5)
```

Now we need to combine it with our date parser, and allow either to succeed. This is done using the `|` *operator*, as follows:

```
>>> flexi_date = full_or_partial_date | days_ago
>>> flexi_date.parse('2012-01-05')
(2012, 1, 5)
>>> days_ago.parse("2 days ago")
datetime.timedelta(-2)
```

Notice that you still get good error messages from the appropriate parser, depending on which parser got furthest before returning a failure:

```
>>> flexi_date.parse('2012-')
ParseError: expected '2 digit month' at 0:5
>>> flexi_date.parse('2 years ago')
ParseError: expected ' days ago' at 0:1
```

When using backtracking, you need to understand that backtracking to the other option only occurs if the first parser fails. So, for example:

```
>>> a = string("a")
>>> ab = string("ab")
>>> c = string("c")
>>> a_or_ab_and_c = ((a | ab) + c)
>>> a_or_ab_and_c.parse('ac')
'ac'
>>> a_or_ab_and_c.parse('abc')
ParseError: expected 'c' at 0:1
```

The parse fails because the `a` parser succeeds, and so the `ab` parser is never tried. This is different from most regular expressions engines, where backtracking is done over the whole regex by default.

In this case we can get the parse to succeed by switching the order:

```
>>> ((ab | a) + c).parse('abc')
'abc'

>>> ((ab | a) + c).parse('ac')
'ac'
```

We could also fix it like this:

```
>>> ((a + c) | (ab + c)).parse('abc')
'abc'
```

3.4 Custom data structures

In the example shown so far, the result of parsing has been a native Python data type, such as a integer, string, datetime or tuple. In some cases that is enough, but very quickly you will find that for your parse result to be useful, you will need to use custom data structures (rather than ending up with nested lists etc.)

For defining custom data structures, you can use any method you like (e.g. simple classes). We recommend attrs. You can also use namedtuple from the standard library for simple cases or dataclasses.

For combining parsed data into these data structures, you can:

1. Use *`Parser.map()`*, *`Parser.combine()`* and *`Parser.combine_dict()`*, often in conjunction with *`seq()`*.

   See the *SQL SELECT and .proto file parser examples* for examples of this approach.

2. Use the `@generate` decorator as above, and manually call the data structure constructor with the pieces, as in `full_date` or `full_or_partial_date` above, but with your own data structure instead of a tuple or datetime in the final line.

3.5 Learn more

For further topics, see the *table of contents* for the rest of the documentation that should enable you to build parsers for your needs.

CHAPTER 4

API reference

4.1 Parsing primitives

These are the lowest level building blocks for creating parsers.

parsy.**string**(*expected_string*, *transform=None*)

Returns a parser that expects the `expected_string` and produces that string value.

Optionally, a transform function can be passed, which will be used on both the expected string and tested string. This allows things like case insensitive matches to be done. This function must not change the length of the string (as determined by `len`). The returned value of the parser will always be `expected_string` in its un-transformed state.

```
>>> parser = string("Hello", transform=lambda s: s.upper())
>>> parser.parse("Hello")
'Hello'
>>> parser.parse("hello")
'Hello'
>>> parser.parse("HELLO")
'Hello'
```

Changed in version 1.2: Added `transform` argument.

parsy.**regex**(*exp*, *flags=0*)

Returns a parser that expects the given `exp`, and produces the matched string. `exp` can be a compiled regular expression, or a string which will be compiled with the given `flags`.

Using a regex parser for small building blocks, instead of building up parsers from primitives like *`string()`*, *`test_char()`* and *`Parser.times()`* combinators etc., can have several advantages, including:

- It can be more succinct e.g. compare:

```
>>> (string('a') | string('b')).times(1, 4)
>>> regex(r'[ab]{1,4}')
```

- It will return the entire matched string as a single item, so you don't need to use *`Parser.concat()`*.

- It can be much faster.

parsy.**test_char**(*func*, *description*)

Returns a parser that tests a single character with the callable `func`. If `func` returns `True`, the parse succeeds, otherwise the parse fails with the description `description`.

```
>>> ascii = test_char(lambda c: ord(c) < 128,
...                   'ascii character')
>>> ascii.parse('A')
'A'
```

parsy.**test_item**(*func*, *description*)

Returns a parser that tests a single item from the list of items being consumed, using the callable `func`. If `func` returns `True`, the parse succeeds, otherwise the parse fails with the description `description`.

If you are parsing a string, i.e. a list of characters, you can use *`test_char()`* instead. (In fact the implementations are identical, these functions are aliases for the sake of clear code).

```
>>> numeric = test_item(str.isnumeric, 'numeric')
>>> numeric.many().parse(['123', '456'])
['123', '456']
```

parsy.**char_from**(*characters*)

Accepts a string and returns a parser that matches and returns one character from the string.

```
>>> char_from('abc').parse('a')
'a'
```

parsy.**string_from**(**strings*, *transform=None*)

Accepts a sequence of strings as positional arguments, and returns a parser that matches and returns one string from the list. The list is first sorted in descending length order, so that overlapping strings are handled correctly by checking the longest one first.

```
>>> string_from('y', 'yes').parse('yes')
'yes'
```

Optionally accepts `transform`, which is passed to *`string()`* (see the documentation there).

Changed in version 1.2: Added `transform` argument.

parsy.**match_item**(*item*, *description=None*)

Returns a parser that tests the next item (or character) from the stream (or string) for equality against the provided item. Optionally a string description can be passed.

Parsing a string:

```
>>> letter_A = match_item('A')
>>> letter_A.parse_partial('ABC')
('A', 'BC')
```

Parsing a list of tokens:

```
>>> hello = match_item('hello')
>>> hello.parse_partial(['hello', 'how', 'are', 'you'])
('hello', ['how', 'are', 'you'])
```

parsy.**success**(*val*)

Returns a parser that does not consume any of the stream, but produces `val`.

parsy.**fail**(*expected*)
Returns a parser that always fails with the provided error message.

parsy.**from_enum**(*enum_cls*, *transform=None*)
Given a class that is an enum.Enum class, returns a parser that will parse the values (or the string representations of the values) and return the corresponding enum item.

```
>>> from enum import Enum
>>> class Pet(Enum):
...     CAT = "cat"
...     DOG = "dog"
>>> pet = from_enum(Pet)
>>> pet.parse("cat")
<Pet.CAT: 'cat'>
```

`str` is first run on the values (for the case of values that are integers etc.) to create the strings which are turned into parsers using *`string()`*.

If `transform` is provided, it is passed to *`string()`* when creating the parser (allowing for things like case insensitive parsing).

parsy.**peek**(*parser*)
Returns a lookahead parser that parse the input stream without consuming chars.

4.1.1 Pre-built parsers

Some common, pre-built parsers (all of these are *`Parser`* objects created using the primitives above):

parsy.**any_char**
A parser that matches any single character.

parsy.**whitespace**
A parser that matches and returns one or more whitespace characters.

parsy.**letter**
A parser that matches and returns a single letter, as defined by str.isalpha.

parsy.**digit**
A parser that matches and returns a single digit, as defined by str.isdigit. Note that this includes various unicode characters outside of the normal 0-9 range, such as ¹²³.

parsy.**decimal_digit**
A parser that matches and returns a single decimal digit, one of "0123456789".

parsy.**line_info**
A parser that consumes no input and always just returns the current line information, a tuple of (line, column), zero-indexed, where lines are terminated by `\n`. This is normally useful when wanting to build more debugging information into parse failure error messages.

parsy.**index**
A parser that consumes no input and always just returns the current stream index. This is normally useful when wanting to build more debugging information into parse failure error messages.

4.2 Parser methods, operators and combinators

4.2.1 Parser methods

Parser objects are returned by any of the built-in parser *Parsing primitives*. They can be used and manipulated as below.

class parsy.**Parser**

__init__(*wrapped_fn*)
This is a low level function to create new parsers that is used internally but is rarely needed by users of the parsy library. It should be passed a parsing function, which takes two arguments - a string/list to be parsed and the current index into the list - and returns a `Result` object, as described in *Creating new Parser instances*.

The following methods are for actually **using** the parsers that you have created:

parse(*string_or_list*)
Attempts to parse the given string (or list). If the parse is successful and consumes the entire string, the result is returned - otherwise, a `ParseError` is raised.

Instead of passing a string, you can in fact pass a list of tokens. Almost all the examples assume strings for simplicity. Some of the primitives are also clearly string specific, and a few of the combinators (such as `Parser.concat()`) are string specific, but most of the rest of the library will work with tokens just as well. See *Separate lexing/tokenization phases* for more information.

parse_partial(*string_or_list*)
Similar to `parse`, except that it does not require the entire string (or list) to be consumed. Returns a tuple of `(result, remainder)`, where `remainder` is the part of the string (or list) that was left over.

The following methods are essentially **combinators** that produce new parsers from the existing one. They are provided as methods on `Parser` for convenience. More combinators are documented below.

desc(*string*)
Adds a description to the parser, which is used in the error message if parsing fails.

```
>>> year = regex(r'[0-9]{4}').desc('4 digit year')
>>> year.parse('123')
ParseError: expected 4 digit year at 0:0
```

then(*other_parser*)
Returns a parser which, if the initial parser succeeds, will continue parsing with `other_parser`. This will produce the value produced by `other_parser`.

```
>>> string('x').then(string('y')).parse('xy')
'y'
```

See also `>>` *operator*.

skip(*other_parser*)
Similar to `Parser.then()`, except the resulting parser will use the value produced by the first parser.

```
>>> string('x').skip(string('y')).parse('xy')
'x'
```

See also `<<` *operator*.

many()
Returns a parser that expects the initial parser 0 or more times, and produces a list of the results. Note that this parser does not fail if nothing matches, but instead consumes nothing and produces an empty list.

```
>>> parser = regex(r'[a-z]').many()
>>> parser.parse('')
[]
>>> parser.parse('abc')
['a', 'b', 'c']
```

times(*min*[, *max*=*min*])
Returns a parser that expects the initial parser at least `min` times, and at most `max` times, and produces a list of the results. If only one argument is given, the parser is expected exactly that number of times.

at_most(*n*)
Returns a parser that expects the initial parser at most `n` times, and produces a list of the results.

at_least(*n*)
Returns a parser that expects the initial parser at least `n` times, and produces a list of the results.

optional()
Returns a parser that expects the initial parser zero or once, and maps the result to `None` in the case of no match.

```
>>> string('A').optional().parse('A')
'A'
>>> string('A').optional().parse('')
None
```

map(*fn*)
Returns a parser that transforms the produced value of the initial parser with `fn`.

```
>>> regex(r'[0-9]+').map(int).parse('1234')
1234
```

This is the simplest way to convert parsed strings into the data types that you need. See also *combine()* and *combine_dict()* below.

combine(*fn*)
Returns a parser that transforms the produced values of the initial parser with `fn`, passing the arguments using `*args` syntax.

Where the current parser produces an iterable of values, this can be a more convenient way to combine them than *map()*.

Example 1 - the argument order of our callable already matches:

```
>>> from datetime import date
>>> yyyymmdd = seq(regex(r'[0-9]{4}').map(int),
...                regex(r'[0-9]{2}').map(int),
...                regex(r'[0-9]{2}').map(int)).combine(date)
>>> yyyymmdd.parse('20140506')
datetime.date(2014, 5, 6)
```

Example 2 - the argument order of our callable doesn't match, and we need to adjust a parameter, so we can fix it using a lambda.

```
>>> ddmmyy = regex(r'[0-9]{2}').map(int).times(3).combine(
...                lambda d, m, y: date(2000 + y, m, d))
```

(continues on next page)

(continued from previous page)

```
>>> ddmmyy.parse('060514')
datetime.date(2014, 5, 6)
```

The equivalent `lambda` to use with `map` would be `lambda res: date(2000 + res[2], res[1], res[0])`, which is less readable. The version with `combine` also ensures that exactly 3 items are generated by the previous parser, otherwise you get a `TypeError`.

`combine_dict` (*fn*)

Returns a parser that transforms the value produced by the initial parser using the supplied function/callable, passing the arguments using the `**kwargs` syntax.

The value produced by the initial parser must be a mapping/dictionary from names to values, or a list of two-tuples, or something else that can be passed to the `dict` constructor.

If `None` is present as a key in the dictionary it will be removed before passing to `fn`, as will all keys starting with _.

Motivation:

For building complex objects, this can be more convenient, flexible and readable than *`map()`* or *`combine()`*, because by avoiding positional arguments we can avoid a dependence on the order of components in the string being parsed and in the argument order of callables being used. It is especially designed to be used in conjunction with *`seq()`* and *`tag()`*.

For Python 3.6 and above, we can make use of the `**kwargs` version of *`seq()`* to produce a very readable definition:

```
>>> ddmmyyyy = seq(
...     day=regex(r'[0-9]{2}').map(int),
...     month=regex(r'[0-9]{2}').map(int),
...     year=regex(r'[0-9]{4}').map(int),
... ).combine_dict(date)
>>> ddmmyyyy.parse('04052003')
datetime.date(2003, 5, 4)
```

(If that is hard to understand, use a Python REPL, and examine the result of the `parse` call if you remove the `combine_dict` call).

Here we used `datetime.date` which accepts keyword arguments. For your own parsing needs you will often use custom data types. You can create these however you like, but we recommend attrs. You can also use namedtuple from the standard library for simple cases, or dataclasses

The following example shows the use of _ as a prefix to remove elements you are not interested in, and the use of `namedtuple` to create a simple data-structure.

```
>>> from collections import namedtuple
>>> Pair = namedtuple('Pair', ['name', 'value'])
>>> name = regex("[A-Za-z]+")
>>> int_value = regex("[0-9]+").map(int)
>>> bool_value = string("true").result(True) | string("false").result(False)
>>> pair = seq(
...     name=name,
...     __eq=string('='),
...     value=int_value | bool_value,
...     __sc=string(';'),
... ).combine_dict(Pair)
>>> pair.parse("foo=123;")
Pair(name='foo', value=123)
```

(continues on next page)

(continued from previous page)

```
>>> pair.parse("BAR=true;")
Pair(name='BAR', value=True)
```

You could also use << or >> for the unwanted parts instead of `.tag(None)` (but in some cases this is less convenient):

```
>>> pair = seq(
...     name=name << string('='),
...     value=(int_value | bool_value) << string(';')
... ).combine_dict(Pair)
```

For Python 3.5 and below, kwargs usage is not possible (because keyword arguments produce a dictionary that does not have a guaranteed order). Instead, use *`tag()`* to produce a list of name-value pairs:

```
>>> ddmmyyyy = seq(
...     regex(r'[0-9]{2}').map(int).tag('day'),
...     regex(r'[0-9]{2}').map(int).tag('month'),
...     regex(r'[0-9]{4}').map(int).tag('year'),
... ).combine_dict(date)
>>> ddmmyyyy.parse('04052003')
datetime.date(2003, 5, 4)
```

The following example shows the use of `tag(None)` to remove elements you are not interested in, and the use of `namedtuple` to create a simple data-structure.

```
>>> from collections import namedtuple
>>> Pair = namedtuple('Pair', ['name', 'value'])
>>> name = regex("[A-Za-z]+")
>>> int_value = regex("[0-9]+").map(int)
>>> bool_value = string("true").result(True) | string("false").result(False)
>>> pair = seq(
...     name.tag('name'),
...     string('=').tag(None),
...     (int_value | bool_value).tag('value'),
...     string(';').tag(None),
... ).combine_dict(Pair)
>>> pair.parse("foo=123;")
Pair(name='foo', value=123)
>>> pair.parse("BAR=true;")
Pair(name='BAR', value=True)
```

You could also use << for the unwanted parts instead of `.tag(None)`:

```
>>> pair = seq(
...     name.tag('name') << string('='),
...     (int_value | bool_value).tag('value') << string(';')
... ).combine_dict(Pair)
```

Changed in version 1.2: Allow lists as well as dicts to be consumed, and filter out `None`.

Changed in version 1.3: Stripping of args starting with _

`tag`(*name*)

Returns a parser that wraps the produced value of the initial parser in a 2 tuple containing `(name, value)`. This provides a very simple way to label parsed components. e.g.:

```
>>> day = regex(r'[0-9]+').map(int)
>>> month = string_from("January", "February", "March", "April", "May",
...                     "June", "July", "August", "September", "October",
...                     "November", "December")
>>> day.parse("10")
10
>>> day.tag("day").parse("10")
('day', 10)

>>> seq(day.tag("day") << whitespace,
...     month.tag("month")
...     ).parse("10 September")
[('day', 10), ('month', 'September')]
```

It also works well when combined with `.map(dict)` to get a dictionary of values:

```
>>> seq(day.tag("name") << whitespace,
...     month.tag("month")
...     ).map(dict).parse("10 September")
{'day': 10, 'month': 'September'}
```

... and with *combine_dict()* to build other objects.

concat()
: Returns a parser that concatenates together (as a string) the previously produced values. Usually used after *many()* and similar methods that produce multiple values.

```
>>> letter.at_least(1).parse("hello")
['h', 'e', 'l', 'l', 'o']
>>> letter.at_least(1).concat().parse("hello")
'hello'
```

result(*val*)
: Returns a parser that, if the initial parser succeeds, always produces `val`.

```
>>> string('foo').result(42).parse('foo')
42
```

should_fail(*description*)
: Returns a parser that fails when the initial parser succeeds, and succeeds when the initial parser fails (consuming no input). A description must be passed which is used in parse failure messages.

  This is essentially a negative lookahead:

```
>>> p = letter << string(" ").should_fail("not space")
>>> p.parse('A')
'A'
>>> p.parse('A ')
ParseError: expected 'not space' at 0:1
```

  It is also useful for implementing things like parsing repeatedly until a marker:

```
>>> (string(";").should_fail("not ;") >> letter).many().concat().parse_
↪partial('ABC;')
('ABC', ';')
```

bind(*fn*)
: Returns a parser which, if the initial parser is successful, passes the result to `fn`, and continues with the

parser returned from `fn`. This is the monadic binding operation. However, since we don't have Haskell's `do` notation in Python, using this is very awkward. Instead, you should look at *Generating a parser* which provides a much nicer syntax for that cases where you would have needed `do` notation in Parsec.

sep_by(*sep*, *min=0*, *max=inf*)
Like *`Parser.times()`*, this returns a new parser that repeats the initial parser and collects the results in a list, but in this case separated by the parser `sep` (whose return value is discarded). By default it repeats with no limit, but minimum and maximum values can be supplied.

```
>>> csv = letter.at_least(1).concat().sep_by(string(","))
>>> csv.parse("abc,def")
['abc', 'def']
```

mark()
Returns a parser that wraps the initial parser's result in a value containing column and line information of the match, as well as the original value. The new value is a 3-tuple:

```
((start_row, start_column),
 original_value,
 (end_row, end_column))
```

This is useful for being able to report problems with parsing more accurately, especially if you are using parsy as a *lexer* and want subsequent parsing of the token stream to be able to report original positions in error messages etc.

4.2.2 Parser operators

This section describes operators that you can use on *`Parser`* objects to build new parsers.

4.2.2.1 | operator

`parser | other_parser`

Returns a parser that tries `parser` and, if it fails, backtracks and tries `other_parser`. These can be chained together.

The resulting parser will produce the value produced by the first successful parser.

```
>>> parser = string('x') | string('y') | string('z')
>>> parser.parse('x')
'x'
>>> parser.parse('y')
'y'
>>> parser.parse('z')
'z'
```

Note that `other_parser` will only be tried if `parser` cannot consume any input and fails. `other_parser` is not used in the case that **later** parser components fail. This means that the order of the operands matters - for example:

```
>>> ((string('A') | string('AB')) + string('C')).parse('ABC')
ParseEror: expected 'C' at 0:1
>>> ((string('AB') | string('A')) + string('C')).parse('ABC')
'ABC'
>>> ((string('AB') | string('A')) + string('C')).parse('AC')
'AC'
```

4.2.2.2 << operator

`parser << other_parser`

The same as `parser.skip(other_parser)` - see *Parser.skip()*.

(Hint - the arrows point at the important parser!)

```
>>> (string('x') << string('y')).parse('xy')
'x'
```

4.2.2.3 >> operator

`parser >> other_parser`

The same as `parser.then(other_parser)` - see *Parser.then()*.

(Hint - the arrows point at the important parser!)

```
>>> (string('x') >> string('y')).parse('xy')
'y'
```

4.2.2.4 + operator

`parser1 + parser2`

Requires both parsers to match in order, and adds the two results together using the + operator. This will only work if the results support the plus operator (e.g. strings and lists):

```
>>> (string("x") + regex("[0-9]")).parse("x1")
"x1"

>>> (string("x").many() + regex("[0-9]").map(int).many()).parse("xx123")
['x', 'x', 1, 2, 3]
```

The plus operator is a convenient shortcut for:

```
>>> seq(parser1, parser2).combine(lambda a, b: a + b)
```

4.2.2.5 * operator

`parser1 * number`

This is a shortcut for doing *Parser.times()*:

```
>>> (string("x") * 3).parse("xxx")
["x", "x", "x"]
```

You can also set both upper and lower bounds by multiplying by a range:

```
>>> (string("x") * range(0, 3)).parse("xxx")
ParseError: expected EOF at 0:2
```

(Note the normal semantics of `range` are respected - the second number is an *exclusive* upper bound, not inclusive).

4.2.3 Parser combinators

parsy.**alt**(**parsers*)

Creates a parser from the passed in argument list of alternative parsers, which are tried in order, moving to the next one if the current one fails, as per the | *operator* - in other words, it matches any one of the alternative parsers.

Example using **arg* syntax to pass a list of parsers that have been generated by mapping *string()* over a list of characters:

```
>>> hexdigit = alt(*map(string, "0123456789abcdef"))
```

(In this case you would be better off using *char_from()*)

Note that the order of arguments matter, as described in | *operator*.

parsy.**seq**(**parsers*, ***kw_parsers*)

Creates a parser that runs a sequence of parsers in order and combines their results in a list.

```
>>> x_bottles_of_y_on_the_z = \
...     seq(regex(r"[0-9]+").map(int) << string(" bottles of "),
...         regex(r"\S+") << string(" on the "),
...         regex(r"\S+")
...         )
>>> x_bottles_of_y_on_the_z.parse("99 bottles of beer on the wall")
[99, 'beer', 'wall']
```

In Python 3.6, you can also use `seq` with keyword arguments instead of positional arguments. In this case, the produced value is a dictionary of the individual values, rather than a sequence. This can make the produced value easier to consume.

```
>>> name = seq(first_name=regex("\S+") << whitespace,
...            last_name=regex("\S+")
>>> name.parse("Jane Smith")
{'first_name': 'Jane',
 'last_name': 'Smith'}
```

Changed in version 1.1: Added `**kwargs` option.

Note: The `**kwargs` feature is for Python 3.6 and later only, because keyword arguments do not keep their order in earlier versions.

For earlier versions, see *Parser.tag()* for a way of labelling parsed components and producing dictionaries.

4.2.4 Other combinators

Parsy does not try to include every possible combinator - there is no reason why you cannot create your own for your needs using the built-in combinators and primitives. If you find something that is very generic and would be very useful to have as a built-in, please *submit* as a PR!

4.3 Generating a parser

parsy.**generate**()

`generate` converts a generator function (one that uses the `yield` keyword) into a parser. The generator function must yield parsers. These parsers are applied successively and their results are sent back to the generator using the `.send()` protocol. The generator function should return the final result of the parsing. Alternatively it can return another parser, which is equivalent to applying it and returning its result.

4.3.1 Motivation and examples

Constructing parsers by using combinators and *Parser* methods to make larger parsers works well for many simpler cases. However, for more complex cases the `generate` function decorator is both more readable and more powerful. (For those coming from Haskell/Parsec, this method provides an acceptable substitute for `do` notation).

4.3.1.1 Alternative syntax to combinators

The first example just shows a different way of building a parser that could have easily been built using combinators:

```
from parsy import generate

@generate("form")
def form():
    """
    Parse an s-expression form, like (a b c).
    An equivalent to lparen >> expr.many() << rparen
    """
    yield lparen
    exprs = yield expr.many()
    yield rparen
    return exprs
```

In the example above, the parser was given a string name `"form"`, which does the same as *Parser.desc()*. This is not required, as per the examples below.

Note that there is no guarantee that the entire function is executed: if any of the yielded parsers fails, the function will not complete, and parsy will try to backtrack to an alternative parser if there is one.

4.3.1.2 Building complex objects

The second example shows how you can use multiple parse results to build up a complex object:

```
from datetime import date

from parsy import generate, regex, string

@generate
def date():
    """
    Parse a date in the format YYYY-MM-DD
    """
    year = yield regex("[0-9]{4}").map(int)
    yield string("-")
    month = yield regex("[0-9]{2}").map(int)
    yield string("-")
    day = yield regex("[0-9]{2}").map(int)

    return date(year, month, day)
```

This could also have been achieved using *seq()* and *Parser.combine()*.

4.3.1.3 Using values already parsed

The third example shows how we can use an earlier parsed value to influence the subsequent parsing. This example parses Hollerith constants. Hollerith constants are a way of specifying an arbitrary set of characters by first writing the integer that specifies the length, followed by the character H, followed by the set of characters. For example, `pancakes` would be written `8Hpancakes`.

```
from parsy import generate, regex, string, any_char

@generate
def hollerith():
    num = yield regex(r'[0-9]+').map(int)
    yield string('H')
    return any_char.times(num).concat()
```

(You may want to compare this with an implementation of Hollerith constants that uses pyparsing, originally by John Shipman from his pyparsing docs.)

There are also more complex examples in the *tutorial* of using the `generate` decorator to create parsers where there is logic that is conditional upon earlier parsed values.

4.3.1.4 Implementing recursive definitions

A fourth examples shows how you can use this syntax for grammars that you would like to define recursively (or mutually recursively).

Say we want to be able to pass an s-expression like syntax which uses parenthesis for grouping items into a tree structure, like the following:

```
(0 1 (2 3) (4 5 6) 7 8)
```

A naive approach would be:

```
simple = regex('[0-9]+').map(int)
group = string('(') >> expr.sep_by(string(' ')) << string(')')
expr = simple | group
```

The problem is that the second line will get a `NameError` because `expr` is not defined yet.

Using the `@generate` syntax will introduce a level of laziness in resolving `expr` that allows things to work:

```
simple = regex('[0-9]+').map(int)

@generate
def group():
    return (yield string('(') >> expr.sep_by(string(' ')) << string(')'))

expr = simple | group
```

```
>>> expr.parse("(0 1 (2 3) (4 5 6) 7 8)")
[0, 1, [2, 3], [4, 5, 6], 7, 8]
```

4.4 Creating new Parser instances

Normally you will create Parser instances using the provided *primitives* and *combinators*.

However it is also possible to create them manually, as below.

The *Parser* constructor should be passed a function that takes the string/list to be parsed and an index into that string, and returns a *Result* object. The Result object will be created either using *Result.success()* or *Result.failure()* to indicate success or failure respectively. *Result.success()* should be passed the next index to continue parsing with, and the value that is returned from the parsing. *Result.failure()* should return the index at which failure occurred i.e. the index passed in, and a string indicating what the parser expected to find.

The Parser constructor will usually be called using decorator syntax. In order to pass parameters to the Parser instance, it is typically created using a closure. In the example below, we create a parser that matches any string/list of tokens of a given length. This could also be written as something like `any_char.times(n).concat()` but the following will be more efficient:

```
def consume(n):

    @Parser
    def consumer(stream, index):
        items = stream[index:index + n]
        if len(items) == n:
            return Result.success(index + n, items)
        else:
            return Result.failure(index, "{0} items".format(n))

    return consumer
```

```
>>> consume(3).many().parse('abc123def')
['abc', '123', 'def']
```

4.4.1 Result objects

class parsy.**Result**

static success(*next_index*, *value*)
Creates a Result object indicating parsing succeeded. The index to continue parsing at, and the value retrieved from the parsing, should be passed.

static failure(*index*, *expected*)
Creates a Result object indicating parsing failed. The index to continue parsing at, and a string representing what the parser expected to find, should be passed.

CHAPTER 5

Howto's, cookbooks and examples

5.1 Separate lexing/tokenization phases

Most of the documentation in parsy assumes that when you call `Parser.parse()` you will pass a string, and will get back your final parsed, constructed object (of whatever type you desire).

A more classical approach to parsing is that you first have a lexing/tokenization phase, the result of which is a simple list of tokens. These tokens could be strings, or other objects.

You then have a separate parsing phase that consumes this list of tokens, and produces your final object, which is very often a tree-like structure or other complex object.

Parsy can actually work with either approach. Further, for the split lexing/parsing approach, parsy can be used either to implement the lexer, or the parser, or both! The following examples use parsy to do both lexing and parsing.

However, parsy's features for this use case are not as developed as some other Python tools. If you are building a parser for a full language that needs the split lexing/parsing approach, you might be better off with PLY.

5.1.1 Turtle Logo

For our first example, we'll do a very stripped down Turtle Logo parser. First, the lexer:

```
"""
Stripped down logo lexer, for tokenizing Turtle Logo programs like:

   fd 1
   bk 2
   rt 90

etc.
"""

from parsy import eof, regex, seq, string, string_from, whitespace
```

(continues on next page)

(continued from previous page)

```
command = string_from("fd", "bk", "rt", "lt")
number = regex(r'[0-9]+').map(int)
optional_whitespace = regex(r'\s*')
eol = string("\n")
line = seq(optional_whitespace >> command,
           whitespace >> number,
           (eof | eol | (whitespace >> eol)).result("\n"))
flatten_list = lambda ls: sum(ls, [])
lexer = line.many().map(flatten_list)
```

We are not interested in whitespace, so our lexer removes it all, apart from newlines. We can now parse a program into the tokens we are interested in:

```
>>> l = lexer.parse("fd 1\nbk 2")
>>> l
['fd', 1, '\n', 'bk', 2, '\n']
```

The `line` parser produces a list, so after applying `many` which also produces a list, we applied a level of flattening so that we end up with a simple list of tokens. We also chose to convert the parameters to integers while we were at it, so in this case our list of tokens is not a list of strings, but heterogeneous.

The next step is the parser. We create some classes to represent different commands, and then use parsy again to create a parser which is very simple because this is a very limited language:

```
from parsy import generate, match_item, test_item


class Command:
    def __init__(self, parameter):
        self.parameter = parameter

    def __repr__(self):
        return "{0}({1})".format(self.__class__.__name__, self.parameter)


class Forward(Command):
    pass


class Backward(Command):
    pass


class Right(Command):
    pass


class Left(Command):
    pass


commands = {
    'fd': Forward,
    'bk': Backward,
    'rt': Right,
    'lt': Left,
}
```

(continues on next page)

(continued from previous page)

```
@generate
def statement():
    cmd_name = yield test_item(lambda i: i in commands.keys(), "command")
    parameter = yield test_item(lambda i: isinstance(i, int), "number")
    yield match_item('\n')
    return commands[cmd_name](int(parameter))


program = statement.many()
```

To use it, we pass the the list of tokens generated above into `program.parse`:

```
>>> program.parse(l)
[Forward(1), Backward(2)]
```

In a real implementation, we could then have `execute` methods on the `Command` sub-classes if we wanted to implement an interpreter, for example.

5.1.2 Calculator

Our second example illustrates lexing and then parsing a sequence of mathematical operations, e.g "1 + 2 * (3 - 4.5)", with precedence.

In this case, while doing the parsing stage, instead of building up an AST of objects representing the operations, the parser actually evaluates the expression.

```
from parsy import digit, generate, match_item, regex, string, success, test_item


def lexer(code):
    whitespace = regex(r'\s*')
    integer = digit.at_least(1).concat().map(int)
    float_ = (
        digit.many() + string('.').result(['.']) + digit.many()
    ).concat().map(float)
    parser = whitespace >> ((
        float_ | integer  | regex(r'[()*/+-]')
    ) << whitespace).many()
    return parser.parse(code)


def eval_tokens(tokens):
    # This function parses and evaluates at the same time.

    lparen = match_item('(')
    rparen = match_item(')')

    @generate
    def additive():
        res = yield multiplicative
        sign = match_item('+') | match_item('-')
        while True:
            operation = yield sign | success('')
```

(continues on next page)

(continued from previous page)

```
            if not operation:
                break
            operand = yield multiplicative
            if operation == '+':
                res += operand
            elif operation == '-':
                res -= operand
        return res

    @generate
    def multiplicative():
        res = yield simple
        op = match_item('*') | match_item('/')
        while True:
            operation = yield op | success('')
            if not operation:
                break
            operand = yield simple
            if operation == '*':
                res *= operand
            elif operation == '/':
                res /= operand
        return res

    @generate
    def number():
        sign = yield match_item('+') | match_item('-') | success('+')
        value = yield test_item(
            lambda x: isinstance(x, (int, float)), 'number')
        return value if sign == '+' else -value

    expr = additive
    simple = (lparen >> expr << rparen) | number

    return expr.parse(tokens)


def simple_eval(expr):
    return eval_tokens(lexer(expr))


if __name__ == '__main__':
    print(simple_eval(input()))
```

5.2 Other examples

This section has some further example parsers that you can study. There are also examples in the *Tutorial* and in *Generating a parser*.

5.2.1 SQL SELECT statement parser

This shows a very simplified parser for a SQL `SELECT` statement, using custom data structures, and the convenient keyword argument syntax for *`seq()`* (usable with Python 3.6 and later), followed by *`Parser.combine_dict()`*.

```
# A very limited parser for SQL SELECT statements,
# for demo purposes. Supports:
# 1. A simple list of columns (or number/string literals)
# 2. A simple table name
# 3. An optional where condition,
#    which has the form of 'A op B' where A and B are columns, strings or number,
#    and op is a comparison operator
#
# We demonstrate the use of `map` to create AST nodes with a single arg,
# and `seq` for AST nodes with more than one arg.


import attr

from parsy import regex, seq, string, string_from

# -- AST nodes.


@attr.s
class Number:
    value = attr.ib()


@attr.s
class String:
    value = attr.ib()


@attr.s
class Field:
    name = attr.ib()


@attr.s
class Table:
    name = attr.ib()


@attr.s
class Comparison:
    left = attr.ib()
    operator = attr.ib()
    right = attr.ib()


@attr.s
class Select:
    columns = attr.ib()
    table = attr.ib()
    where = attr.ib()


# -- Parsers

number_literal = regex(r'-?[0-9]+').map(int).map(Number)
```

(continues on next page)

(continued from previous page)

```
# We don't support ' in strings or escaping for simplicity
string_literal = regex(r"'[^']*'").map(lambda s: String(s[1:-1]))

identifier = regex('[a-zA-Z][a-zA-Z0-9_]*')

field = identifier.map(Field)

table = identifier.map(Table)

space = regex(r'\s+')  # non-optional whitespace
padding = regex(r'\s*')  # optional whitespace

column_expr = field | string_literal | number_literal

operator = string_from('=', '<', '>', '<=', '>=')

comparison = seq(
    left=column_expr << padding,
    operator=operator,
    right=padding >> column_expr,
).combine_dict(Comparison)

SELECT = string('SELECT')
FROM = string('FROM')
WHERE = string('WHERE')

# Here we demonstrate use of leading underscore to discard parts we don't want,
# which is more readable and convenient than `<<` and `>>` sometimes.
select = seq(
    _select=SELECT + space,
    columns=column_expr.sep_by(padding + string(',') + padding, min=1),
    _from=space + FROM + space,
    table=table,
    where=(space >> WHERE >> space >> comparison).optional(),
    _end=padding + string(';')
).combine_dict(Select)


def test_select():
    assert select.parse(
        "SELECT thing, stuff, 123, 'hello' FROM my_table WHERE id = 1;"
    ) == Select(
        columns=[Field("thing"), Field("stuff"), Number(123), String("hello")],
        table=Table("my_table"),
        where=Comparison(
            left=Field("id"),
            operator="=",
            right=Number(1)
        )
    )


def test_optional_where():
    assert select.parse(
        "SELECT 1 FROM x;"
    ) == Select(
        columns=[Number(1)],
```

(continues on next page)

(continued from previous page)

```
        table=Table("x"),
        where=None,
    )
```

5.2.2 JSON parser

```
from sys import stdin

from parsy import generate, regex, string

whitespace = regex(r'\s*')
lexeme = lambda p: p << whitespace
lbrace = lexeme(string('{'))
rbrace = lexeme(string('}'))
lbrack = lexeme(string('['))
rbrack = lexeme(string(']'))
colon  = lexeme(string(':'))
comma  = lexeme(string(','))
true   = lexeme(string('true')).result(True)
false  = lexeme(string('false')).result(False)
null   = lexeme(string('null')).result(None)
number = lexeme(
    regex(r'-?(0|[1-9][0-9]*)([.][0-9]+)?([eE][+-]?[0-9]+)?')
).map(float)
string_part = regex(r'[^"\\]+')
string_esc = string('\\') >> (
    string('\\')
    | string('/')
    | string('"')
    | string('b').result('\b')
    | string('f').result('\f')
    | string('n').result('\n')
    | string('r').result('\r')
    | string('t').result('\t')
    | regex(r'u[0-9a-fA-F]{4}').map(lambda s: chr(int(s[1:], 16)))
)
quoted = lexeme(string('"') >> (string_part | string_esc).many().concat() << string('"
↪'))


# Circular dependency between array and value means we use `generate` form here
@generate
def array():
    yield lbrack
    elements = yield value.sep_by(comma)
    yield rbrack
    return elements


@generate
def object_pair():
    key = yield quoted
    yield colon
    val = yield value
    return (key, val)
```

(continues on next page)

(continued from previous page)

```
json_object = lbrace >> object_pair.sep_by(comma).map(dict) << rbrace
value = quoted | number | json_object | array | true | false | null
json = whitespace >> value

if __name__ == '__main__':
    print(repr(json.parse(stdin.read())))
```

5.2.3 .proto file parser

A parser for the `.proto` files for Protocol Buffers, version 3.

This example is useful in showing lots of simple custom data structures for holding the result of the parse. It uses the *Parser.tag()* method for labelling parts, followed by *Parser.combine_dict()*.

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

# Parser for protocol buffer .proto files
import enum as stdlib_enum
from string import ascii_letters, digits, hexdigits, octdigits

import attr

from parsy import char_from, from_enum, generate, regex, seq, string

# This file follows the spec at
# https://developers.google.com/protocol-buffers/docs/reference/proto3-spec
# very closely.

# However, because we are parsing into useful objects, we do transformations
# along the way e.g. turning into integers, strings etc. and custom objects.
# Some of the lowest level items have been implemented using 'regex' and converting
# the descriptions to regular expressions. Higher level constructs have been
# implemented using other parsy primitives and combinators.

# Notes:

# 1. Whitespace is very badly defined in the 'spec', so we guess what is meant.
# 2. The spec doesn't allow for comments, and neither does this parser.
#    Other places mention that C++ style comments are allowed. To support that,
#    this parser would need to be changed into split lexing/parsing stages
#    (otherwise you hit issues with comments start markers within string literals).
# 3. Other notes inline.


# Our utilities
optional_string = lambda s: string(s).times(0, 1).concat()
convert_decimal = int
convert_octal = lambda s: int(s, 8)
convert_hex = lambda s: int(s, 16)
exclude_none = lambda l: [i for i in l if i is not None]


def lexeme(p):
```

(continues on next page)

(continued from previous page)

```
    """
    From a parser (or string), make a parser that consumes
    whitespace on either side.
    """
    if isinstance(p, str):
        p = string(p)
    return regex(r'\s*') >> p << regex(r'\s*')


def is_present(p):
    """
    Given a parser or string, make a parser that returns
    True if the parser matches, False otherwise
    """
    return lexeme(p).optional().map(lambda v: False if v is None else True)


# Our data structures
@attr.s
class Import:
    identifier = attr.ib()
    option = attr.ib()


@attr.s
class Package:
    identifer = attr.ib()


@attr.s
class Option:
    name = attr.ib()
    value = attr.ib()


@attr.s
class Field:
    repeated = attr.ib()
    type = attr.ib()
    name = attr.ib()
    number = attr.ib()
    options = attr.ib()


@attr.s
class OneOfField:
    type = attr.ib()
    name = attr.ib()
    number = attr.ib()
    options = attr.ib()


@attr.s
class OneOf:
    name = attr.ib()
    fields = attr.ib()
```

(continues on next page)

(continued from previous page)

```
@attr.s
class Map:
    key_type = attr.ib()
    type = attr.ib()
    name = attr.ib()
    number = attr.ib()
    options = attr.ib()


@attr.s
class Reserved:
    items = attr.ib()


@attr.s
class Range:
    from_ = attr.ib()
    to = attr.ib()


@attr.s
class EnumField:
    name = attr.ib()
    value = attr.ib()
    options = attr.ib()


@attr.s
class Enum:
    name = attr.ib()
    body = attr.ib()


@attr.s
class Message:
    name = attr.ib()
    body = attr.ib()


@attr.s
class Service:
    name = attr.ib()
    body = attr.ib()


@attr.s
class Rpc:
    name = attr.ib()
    request_stream = attr.ib()
    request_message_type = attr.ib()
    response_stream = attr.ib()
    response_message_type = attr.ib()
    options = attr.ib()


@attr.s
```

(continues on next page)

(continued from previous page)

```
class Proto:
    syntax = attr.ib()
    statements = attr.ib()


# Enums:
class ImportOption(stdlib_enum.Enum):
    WEAK = "weak"
    PUBLIC = "public"


class Type(stdlib_enum.Enum):
    DOUBLE = "double"
    FLOAT = "float"
    INT32 = "int32"
    INT64 = "int64"
    UINT32 = "uint32"
    UINT64 = "uint64"
    SINT32 = "sint32"
    SINT64 = "sint64"
    FIXED32 = "fixed32"
    FIXED64 = "fixed64"
    SFIXED32 = "sfixed32"
    SFIXED64 = "sfixed64"
    BOOL = "bool"
    STRING = "string"
    BYTES = "bytes"


class KeyType(stdlib_enum.Enum):
    INT32 = "int32"
    INT64 = "int64"
    UINT32 = "uint32"
    UINT64 = "uint64"
    SINT32 = "sint32"
    SINT64 = "sint64"
    FIXED32 = "fixed32"
    FIXED64 = "fixed64"
    SFIXED32 = "sfixed32"
    SFIXED64 = "sfixed64"
    BOOL = "bool"
    STRING = "string"


# Some extra constants to avoid typing
SEMI = lexeme(";")
EQ = lexeme("=")
LPAREN = lexeme("(")
RPAREN = lexeme(")")
LBRACE = lexeme("{")
RBRACE = lexeme("}")

# -- Beginning of following spec --
# Letters and digits
letter = char_from(ascii_letters)
decimalDigit = char_from(digits)
octalDigit = char_from(octdigits)
```

(continues on next page)

(continued from previous page)

```
hexDigit = char_from(hexdigits)

# Identifiers

# Compared to spec, we add some '_' prefixed items which are not wrapped in `lexeme`,
# on the assumption that spaces in the middle of identifiers are not accepted.
_ident = (letter + (letter | decimalDigit | string("_")).many().concat()).desc('ident
↪')
ident = lexeme(_ident)
fullIdent = lexeme(ident + (string(".") + ident).many().concat()).desc('fullIdent')
_messageName = _ident
messageName = lexeme(ident).desc('messageName')
_enumName = ident
enumName = lexeme(_enumName).desc('enumName')
fieldName = ident.desc('fieldName')
oneofName = ident.desc('oneofName')
mapName = ident.desc('mapName')
serviceName = ident.desc('serviceName')
rpcName = ident.desc('rpcName')
messageType = optional_string(".") + (_ident + string(".")).many().concat() + _
↪messageName
enumType = optional_string(".") + (_ident + string(".")).many().concat() + _enumName

# Integer literals
decimalLit = regex("[1-9][0-9]*").desc('decimalLit').map(convert_decimal)
octalLit   = regex("0[0-7]*").desc('octalLit').map(convert_octal)
hexLit     = regex("0[x|X][0-9a-fA-F]+").desc('octalLit').map(convert_hex)
intLit     = decimalLit | octalLit | hexLit


# Floating-point literals
decimals = r'[0-9]+'
exponent  = r'[e|E][+|-]?' + decimals
floatLit = regex(r'({decimals}\.({decimals})?({exponent})?)|{decimals}{exponent}|\.
↪{decimals}({exponent})?'
                 .format(decimals=decimals, exponent=exponent)).desc('floatLit').
↪map(float)


# Boolean
boolLit = (string("true").result(True) | string("false").result(False)).desc('boolLit
↪')


# String literals
hexEscape = regex(r"\\[x|X]") >> regex("[0-9a-fA-F]{2}").map(convert_hex).map(chr)
octEscape = regex(r"\\") >> regex('[0-7]{2}').map(convert_octal).map(chr)
charEscape = regex(r"\\") >> (
    string("a").result("\a")
    | string("b").result("\b")
    | string("f").result("\f")
    | string("n").result("\n")
    | string("r").result("\r")
    | string("t").result("\t")
    | string("v").result("\v")
    | string("\\").result("\\")
    | string("'").result("'")
```

(continues on next page)

(continued from previous page)

```
    | string('"').result('"')
)
escapes = hexEscape | octEscape | charEscape
# Correction to spec regarding " and ' inside quoted strings
strLit = (string("'") >> (escapes | regex(r"[^\0\n\'\\]")).many().concat() << string("
↪'")
          | string('"') >> (escapes | regex(r"[^\0\n\"\\]")).many().concat() <<␣
↪string('"')).desc('strLit')
quote = string("'") | string('"')

# EmptyStatement
emptyStatement = string(";").result(None)

# Signed numbers:
# (Extra compared to spec, to cope with need to produce signed numeric values)
signedNumberChange = lambda s, num: (-1) if s == "-" else (+1)
sign = regex("[-+]?")
signedIntLit = seq(sign, intLit).combine(signedNumberChange)
signedFloatLit = seq(sign, floatLit).combine(signedNumberChange)


# Constant
# put fullIdent at end to disabmiguate from boolLit
constant = signedIntLit | signedFloatLit | strLit | boolLit | fullIdent

# Syntax
syntax = lexeme("syntax") >> EQ >> quote >> string("proto3") << quote + SEMI

# Import Statement
import_option = from_enum(ImportOption)

import_ = seq(lexeme("import") >> import_option.optional().tag('option'),
              lexeme(strLit).tag('identifier') << SEMI).combine_dict(Import)

# Package
package = seq(lexeme("package") >> fullIdent << SEMI).map(Package)

# Option
optionName = (ident | (LPAREN >> fullIdent << RPAREN)) + (string(".") + ident).many().
↪concat()
option = seq(lexeme("option") >> optionName.tag('name'),
             EQ >> constant.tag('value') << SEMI,
             ).combine_dict(Option)

# Normal field
type_ = lexeme(from_enum(Type) | messageType | enumType)
fieldNumber = lexeme(intLit)

fieldOption = seq(optionName.tag('name'),
                  EQ >> constant.tag('value')).combine_dict(Option)
fieldOptions = fieldOption.sep_by(lexeme(","), min=1)
fieldOptionList = (lexeme("[") >> fieldOptions << lexeme("]")).optional().map(
    lambda o: [] if o is None else o)

field = seq(is_present("repeated").tag('repeated'),
            type_.tag('type'),
            fieldName.tag('name') << EQ,
```

(continues on next page)

(continued from previous page)

```
            fieldNumber.tag('number'),
            fieldOptionList.tag('options') << SEMI,
            ).combine_dict(Field)


# Oneof and oneof field
oneofField = seq(type_.tag('type'),
                 fieldName.tag('name') << EQ,
                 fieldNumber.tag('number'),
                 fieldOptionList.tag('options') << SEMI,
                 ).combine_dict(OneOfField)
oneof = seq(lexeme("oneof") >> oneofName.tag('name'),
            LBRACE
            >> (oneofField | emptyStatement).many().map(exclude_none).tag('fields')
            << RBRACE
            ).combine_dict(OneOf)

# Map field
keyType = lexeme(from_enum(KeyType))
mapField = seq(lexeme("map") >> lexeme("<") >> keyType.tag('key_type'),
               lexeme(",") >> type_.tag('type'),
               lexeme(">") >> mapName.tag('name'),
               EQ >> fieldNumber.tag('number'),
               fieldOptionList.tag('options') << SEMI
               ).combine_dict(Map)

# Reserved
range_ = seq(lexeme(intLit).tag('from_'),
             (lexeme("to") >> (intLit | lexeme("max"))).optional().tag('to')
             ).combine_dict(Range)
ranges = range_.sep_by(lexeme(","), min=1)
# The spec for 'reserved' indicates 'fieldName' here, which is never a quoted string.
# But the example has a quoted string. We have changed it to 'strLit'
fieldNames = strLit.sep_by(lexeme(","), min=1)
reserved = seq(lexeme("reserved") >> (ranges | fieldNames) << SEMI
               ).combine(Reserved)

# Enum definition
enumValueOption = seq(optionName.tag('name') << EQ,
                      constant.tag('value')
                      ).combine_dict(Option)
enumField = seq(ident.tag('name') << EQ,
                lexeme(intLit).tag('value'),
                (lexeme("[") >> enumValueOption.sep_by(lexeme(","), min=1) << lexeme(
↪"]")).optional()
                .map(lambda o: [] if o is None else o).tag('options')
                << SEMI
                ).combine_dict(EnumField)
enumBody = (LBRACE
            >> (option | enumField | emptyStatement).many().map(exclude_none)
            << RBRACE)
enum = seq(lexeme("enum") >> enumName.tag('name'),
           enumBody.tag('body')
           ).combine_dict(Enum)


# Message definition
```

(continues on next page)

(continued from previous page)

```
@generate
def message():
    yield lexeme("message")
    name = yield messageName
    body = yield messageBody
    return Message(name=name, body=body)


messageBody = (LBRACE
               >> (field | enum | message | option | oneof | mapField
                   | reserved | emptyStatement).many()
               << RBRACE)


# Service definition
rpc = seq(lexeme("rpc") >> rpcName.tag('name'),
          LPAREN
          >> (is_present("stream").tag("request_stream")),
          messageType.tag("request_message_type") << RPAREN,
          lexeme("returns") >> LPAREN
          >> (is_present("stream").tag("response_stream")),
          messageType.tag("response_message_type")
          << RPAREN,
          ((LBRACE
           >> (option | emptyStatement).many()
           << RBRACE)
           | SEMI.result([])
           ).optional().map(exclude_none).tag('options')
          ).combine_dict(Rpc)

service = seq(lexeme("service") >> serviceName.tag('name'),
              LBRACE
              >> (option | rpc | emptyStatement).many().map(exclude_none).tag('body')
              << RBRACE
              ).combine_dict(Service)


# Proto file
topLevelDef = message | enum | service
proto = seq(syntax.tag('syntax'),
            (import_ | package | option | topLevelDef | emptyStatement
             ).many().map(exclude_none).tag('statements')
            ).combine_dict(Proto)


EXAMPLE = """
syntax = "proto3";
import public "other.proto";
option java_package = "com.example.foo";
enum EnumAllowingAlias {
  option allow_alias = true;
  UNKNOWN = 0;
  STARTED = 1;
  RUNNING = 2 [(custom_option) = "hello world"];
}
message outer {
  option (my_option).a = true;
```

(continues on next page)

(continued from previous page)

```
  message inner {
    int64 ival = 1;
  }
  repeated inner inner_message = 2;
  EnumAllowingAlias enum_field =3;
  map<int32, string> my_map = 4;
}
"""
# Smoke test - should find 4 top level statements in the example:
assert len(proto.parse(EXAMPLE).statements) == 4
```

CHAPTER 6

History and release notes

6.1 1.3.0 - 2019-08-03

- Documentation improvements.
- Added *peek()* - thanks @lisael.
- Removed Python 3.3 support
- Added Python 3.7 support
- *Parser.combine_dict()* now strips keys that start with _.

6.2 1.2.0 - 2017-11-15

- Added `transform` argument to *string()* and *string_from()*.
- Made *Parser.combine_dict()* accept lists of name value pairs, and filter out keys with value `None`.
- Added *from_enum()*.

6.3 1.1.0 - 2017-11-05

- Added *Parser.optional()*.
- Added *Parser.tag()*.
- Added *seq()* keyword argument version (Python 3.6)
- Added *Parser.combine_dict()*.
- Documented *Parser.mark()*.
- Documentation improvements.

6.4 1.0.0 - 2017-10-10

- Improved parse failure messages of `@generate` parsers. Previously the parser was given a default description of the function name, which hides all useful internal info there might be.
- Added *Parser.sep_by()*
- Added *test_char()*
- Added *char_from()*
- Added *string_from()*
- Added *any_char*
- Added *decimal_digit*
- Added *Parser.concat()*
- Fixed parsy so that it can again work with tokens as well as strings, allowing it to be used as both a *lexer or parser or both*, with docs and tests.
- Added *test_item()*
- Added *match_item()*
- Added *Parser.should_fail()*

6.5 0.9.0 - 2017-09-28

- Better error reporting of failed parses.
- Documentation overhaul and expansion.
- Added *Parser.combine()*.

6.6 0.0.4 - 2014-12-28

- See git logs for changes before this point.

CHAPTER 7

Contributing to parsy

Contributions to parsy, whether code or docs, are very welcome. Please contribute by making a fork, and submitting a PR on GitHub.

We have a high standard in terms of quality. All contributions will need to be fully covered by unit tests and documentation. Code should be formatted according to pep8, and the formatting defined by the `../.editorconfig` file (see EditorConfig).

To run the test suite:

```
pip install pytest
pytest
```

To run the test suite on all supported Python versions, and code quality checks, first install the various Python versions, then:

```
pip install tox
tox
```

To build the docs, do:

```
pip install sphinx
cd docs
make html
```

We also require that flake8, isort and checkmanifest report zero errors (these are run by tox).

When writing documentation, please keep in mind Daniele Procida's great article on documentation. To summarise, there are 4 types of docs:

- Tutorials (focus: learning, analogy: teaching a child to cook)
- How-to guides (focus: goals, analogy: a recipe in a cook book)
- Discussions (focus: understanding, analogy: an article on culinary history)
- Reference (focus: information, analogy: encyclopedia article)

We do not (yet) have documentation that fits into the "Discussions" category, but we do have the others, and when adding new features, documentation of the right sort(s) should be added. With parsy, where code is often very succinct, writing good docs often takes several times longer than writing the code.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index