Reparse Documentation

Release 1.0

Andy Chase

November 22, 2015

Reparse is simply a tool for combining regular expressions together and using a regular expression engine to scan/search/parse/process input for certain tasks.

Larger parsing tools like YACC/Bison, ANTLR, and others are really good for structured input like computer code or xml. They aren't specifically designed for scanning and parsing semi-structured data from unstructured text (like books, or internet documents, or diaries).

Reparse is designed to work with exactly that kind of stuff, (and is completely useless for the kinds of tasks any of the above is often used for).

Parsing Spectrum

Reparse isn't the first parser of it's kind. A hypothetical spectrum of parsers from pattern-finding only all the way to highly-featured, structured grammars might look something like this:

```
                v- Reparse            v- YACC/Bison
UNSTRUCTURED |--------------------------| STRUCTURED
             ^- Regex     ^- Parboiled/PyParsing
```

Reparse is in fact very featureless. It's only a little better than plain regular expressions. Still, you might find it ideal for the kinds of tasks it was designed to deal with (like dates and addresses).

CHAPTER 2

What kind of things might Reparse be useful for parsing?

Any kind of semi-structured formats:

- Uris
- Numbers
- Strings
- Dates
- Times
- Addresses
- Phone numbers

Or in other words, anything you might consider parsing with Regex, might consider Reparse, especially if you are considering combining multiple regular expressions together.

CHAPTER 3

Why Regular Expressions

PyParsing (Python) and Parboiled (JVM) also have use-cases very similar to Reparse, and they are much more feature-filled. They have their own (much more powerful) DSL for parsing text.

Reparse uses Regular Expressions which has some advantages:

- Short, minimal Syntax
- Universal (with some minor differences between different engines)
- Standard
- **Moderately Easy-to-learn (Though this is highly subjective)**
    - Many programmers already know the basics
    - Skills can be carried else where
- **Regular Expressions can be harvested elsewhere and used within Reparse**
- Decent performance over large inputs
- Ability to use fuzzy matching regex engines

CHAPTER 4

Limitations of Reparse

Regular Expressions have been known to catch input that was unexpected, or miss input that was expected due to unforeseen edge cases. Reparse provides tools to help alleviate this by checking the expressions against expected matching inputs, and against expected non-matching inputs.

This library is very limited in what it can parse, if you realize you need something like a recursive grammar, you might want to try PyParsing or something greater (though Reparse might be helpful as a ‘first step’ matching and transforming the parse-able data before it is properly parsed by a different library).

CHAPTER 5

Howto: How to use Reparse

5.1 You will need

1. A Python environment & some Regular Expression knowledge. Some resources: RegexOne, Regex-Info, W3Schools.
2. Some example texts that you will want to parse and their solutions. This will be useful to check your parser and will help you put together the expressions and patterns.

5.2 1. Setup Python & Reparse

See *Installation* for instructions on how to install Reparse

5.3 2. Layout of an example Reparse parser

Reparse needs 3 things in its operation:

1. Functions: A dictionary with String Key -> Function Value mapping.

```
{'cool func': func}
```

2. Expressions: A dictionary with Groups (Dict) -> Expressions (Dict) -> Expression (String/Regex). I typically format this in Yaml since it's easy to read and write, but you can do it in json, or even straight in Python.

```
{'my cool group': {'my cool expression': {'expression': '[a-z]+'}}}
```

3. Patterns: A dictionary with Patterns (Dict) -> Pattern (String/Regex) (Ditto about the Yaml).

```
{'my cool pattern': {'pattern': "<my cool group>.*" }}
```

As I mentioned I typically just use Yaml so my file structure looks like this:

```
my_parser.py <- The actual parser
functions.py <- All the functions and a dictionary mapping them at the end
expressions.yaml <- All of the Regex pieces nice and structured
patterns.yaml <- All the Regex patterns
```

This isn't forced, you could have them all in one file, or it split up in many, many files. This is just how I organize my parsing resources.

5.4 3. Writing your first expressions.yaml file

I'll be parsing Phone numbers in this example.

Tip: You wanna know a secret? I don't do all the hard work myself, I often borrow Regexs off other sites like http://regexlib.com/.

```
Phone: # <- This is the expression group name, it's like the 'type' of your regex
       #    All the expressions in one group should be able to be substituted for each other
    Senthil Gunabalan: # <- This is the expression
                       # I use the authors name because I couldn't be asked to come up with names fo
        Expression: |
            [+]([0-9] \d{2}) - (\d{3}) - (\d{4})
        # Whitespace is ignored, so you can use it to make your regexs readable
        Description: This is a basic telephone number validation [...]
        Matches: +974-584-5656 | +000-000-0000 | +323-343-3453
        Non-Matches: 974-584-5656 | +974 000 0000
        Groups:
          - AreaCode
          - Prefix
          - Body
        # The keys in the Groups: field have to match up with the capture groups (stuff in parenthes
        # They are used as keyword arguments to the function that processes this expression
        # (Expression groups 'Phone' and capture groups () are different)
```

So that's a basic expression file. The hierarchy goes:

Group:

Expression: Detail: Detail: Detail:

Group:

Expression: Detail: Detail: Detail:

Expression: Detail:

5.5 4. Writing your first patterns.yaml file

There aren't any capture groups in patterns. All the capture groups should be done in expressions and merely *combined* in patterns.

```
Basic Phone:
    Pattern: <Phone>
    Order: 1

Fax Phone:
    Pattern: |
        Fax: \s <Phone>
    Order: 2
    # I could have used <Basic Phone> instead to use a pattern inside a pattern but it wouldn't have
```

The order field tells Reparse which pattern to pick if multiple patterns match. Generally speaking, the more specific patterns should be ordered higher than the lower ones (you wouldn't want someone to try and call a fax machine!).

I could have split the expression above into 4 expression groups: Country Code, Area Code, 3-digit prefix, 4-digit body, and combined them in the patterns file, and that would have looked like this:

```
Mega International:
    Pattern: [+]<Country Code>-<Area Code>-<3-digit prefix>-<4-digit body>
```

Done this way, I could have had 3 different formats for Area Code and the pattern would have matched on any of them. I didn't here because that'd be overkill for phone numbers.

5.6 5. Writing your functions.py file

Reparse matches text and also does some parsing using functions.

The order in which the functions are run and results passed are as follows:

1. The Function mapped to the Expression name is called with keyword arguments named in the `Groups:` key ('Senthil Gunabalan' in this example).
2. The output of that function is passed to the function mapped to the Expression Group ('Phone' in this example).
3. The output of that function is passed to the function mapped to the Pattern name ('Basic Phone' or 'Fax Phone').
4. (Optional) If you used *Patterns inside Patterns* then the output bubbles up to the top.
5. The output of that function is returned.

```python
from collections import namedtuple
Phone = namedtuple('phone', 'area_code prefix body fax')


def senthill(AreaCode, Prefix, Body):
    return Phone(area_code=AreaCode, prefix=Prefix, body=Body, fax=False)


def phone(p):
    return p[0]


def basic_phone(p):
    return p


def fax_phone(p):
    return p._replace(fax=True)

functions = {
   'Senthil Gunabalan' : senthill,
   'Phone' : phone,
   'Basic Phone' : basic_phone,
   'Fax Phone' : fax_phone
}
```

I used namedtuples here, but you can parse your output anyway you want to.

5.7 6. Combining it all together!

The builder.py module contains some functions to build a Reparse system together. Here's how I'd put together my phone number parser:

```python
from examples.phone.functions import functions
import reparse

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


print(phone_parser('+974-584-5656'))
# => [phone(area_code='974', prefix='584', body='5656', fax=False)]
print(phone_parser('Fax: +974-584-5656'))
# => [phone(area_code='974', prefix='584', body='5656', fax=True)]
```

5.8 7. More info

Yeah, so this was all basically straight out of the examples/phone directory where you can run it yourself and see if it actually works.

There's more (or at least one more) example in there for further insight.

Happy parsing!

CHAPTER 6

Best practices for Regular Expressions

As your Regexs grow and form into beautiful pattern-matching powerhouses, it is important to take good care of them, and they have a tendency to grow unruly as they mature.

Here are some several safe practices for handling your Regexs so that they can have a long productive life without getting out of control:

- Use whitespace and comments in your regular expressions. Whitespace is set to be ignored by default (use /s to match whitespace) so use spaces and newlines to break up sections freely. Regex comments look like (?# this).
- Never let a regex become too big to be easily understood. Split up big regex into smaller expressions. (Sensible splits won't hurt them).
- **Maintain a Matches and Non-Matches**
  - Reparse can use this to test your Regex to make sure they are matching properly
  - It helps maintainers see which regular expressions match what quickly
  - It helps show your intention with each expression, so that others can confidently improve or modify them
- Maintain a description which talks about what you are trying to match with each regex, what you are not matching and why, and possibly a url where they might learn more about that specific format.
- Having each regex author list his name can be a great boon. It gives them credit for their work, it encourages them to put forth their best effort, and is an easy way to name them. I often name the regex after the the author so I don't have to come up with unique names for all my regexs, since that are often really similar.

For more information about maintaining a regex-safe environment visit:

http://www.codinghorror.com/blog/2008/06/regular-expressions-now-you-have-two-problems.html

Here lies the embedded docblock documentation for the various parts of Reparse.

CHAPTER 7

expression

`reparse.expression.`**`AlternatesGroup`**(*expressions*, *final_function*, *name=u''*)

Group expressions using the OR character `|` >>> from collections import namedtuple >>> expr = namedtuple('expr', 'regex group_lengths run')('(1)', [1], None) >>> grouping = AlternatesGroup([expr, expr], lambda f: None, 'yeah') >>> grouping.regex # doctest: +IGNORE_UNICODE '(?:(1))|(?:(1))' >>> grouping.group_lengths [1, 1]

class `reparse.expression.`**`Expression`**(*regex*, *functions*, *group_lengths*, *final_function*, *name=u''*)

Expressions are the building blocks of parsers.

Each contains:

- A regex pattern (lazily compiled on first usage)
- Group lengths, functions and names
- a `final_function`

When an expression runs with `findall` or `scan`, it matches a string using its regex, and returns the results from the parsing functions.

`findall`(*string*)

Parse string, returning all outputs as parsed by functions

`run`(*matches*)

Run group functions over matches

`scan`(*string*)

Like findall, but also returning matching start and end string locations

`reparse.expression.`**`Group`**(*expressions*, *final_function*, *inbetweens*, *name=u''*)

Group expressions together with `inbetweens` and with the output of a `final_functions`.

builders

class `reparse.builders.`**`Expression_Builder`**(*expressions_dict*, *function_builder*)

Expression builder is useful for building regex bits with Groups that cascade down:

```
from  GROUP      (group).?,?(group)|(group) (group)
  to  EXPRESSION        (group)    |    (group)
  to  TYPE                      (group)
```

```
>>> dummy = lambda input: input
>>> get_function = lambda *_, **__: dummy
>>> function_builder = lambda: None
>>> function_builder.get_function = get_function
>>> expression = {'greeting':{'greeting':{'Expression': '(hey)|(cool)', 'Groups' : ['greeting',
>>> eb = Expression_Builder(expression, function_builder)
>>> eb.get_type("greeting").findall("hey, cool!")
[[('hey',), ('',)], [('',), ('cool',)]]
```

class `reparse.builders.`**`Function_Builder`**(*functions_dict*)

Function Builder is an on-the-fly builder of functions for expressions

```
>>> def t(_):
...     return _
>>> fb = Function_Builder({"hey":t})
>>> fb.get_function("hey", "") is t
True
```

CHAPTER 9

tools

9.1 expression_checker

This module contains a useful function that aids in validating expression files by making sure each expression in a group are valid against the 'Expression', 'Matches', & 'Non-Matches' fields.

It is intended to be used against a `testing_framework` that contains assertIn, assertTrue, assertFalse (such as unittest).

Example Usage:

```
from reparse.tools.expression_checker import check_expression
import unittest

class cool_test(unittest.TestCase):
    def test_coolness(self):
        check_expression(self, load_yaml("parse/cool/expressions.yaml"))
```

`reparse.tools.expression_checker.`**`check_expression`**`(`*testing_framework*, *expression_dict*`)`

```
>>> class mock_framework:
...   def assertIn(self, item, list, msg="Failed asserting item is in list"):
...     if item not in list: raise Exception(msg)
...   def assertTrue(self, value, msg="Failed asserting true"):
...     if not value: raise Exception(msg)
...   def assertFalse(self, value, msg): self.assertTrue(not value, msg)
>>> check_expression(mock_framework(),
...   {'class': {'group' :{'Matches': " 0 | 1", 'Non-Matches': "2 | 0 2", 'Expression': "[0-1]"}
```

CHAPTER 10

Reparse

Python library/tools for combining and parsing using Regular Expressions in a maintainable way

[Download/View Source on Github] [Docs at ReadtheDocs]

This library also allows you to:

- Maintain a database of Regular Expressions
- Combine them together using Patterns
- Search, Parse and Output data matched by combined Regex using Python functions.

This library basically just gives you a way to combine Regular Expressions together and hook them up to some callback functions in Python.

A Taste / Getting Started

(See the examples/ directory for a full code examples)

Say your fashionista friend must know what colors their friends like at certain times. Luckily for you two, your friend's friends are blogging fanatics and you have downloaded thousands of text documents containing their every thought.

So you want to get (color and time) or `[('green', datetime.time(23, 0))]` out of text like:

```
blah blah blah go to the store to buy green at 11pm! blah blah
```

If you need scan/search/parse/transform some unstructured input and get some semi-structured data out of it Reparse might be able to help.

11.1 First structure some Regular Expressions (Here, in Yaml)

```
Color:
    Basic Color:
        Expression: (Red|Orange|Yellow|Green|Blue|Violet|Brown|Black)
        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
```

11.2 Then structure some Patterns with those expressions (Yaml)

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

11.3 Some callback functions (in Python)

```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
```

11.4 Build your parser

```python
from examples.colortime.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! ~ ~"))
```

11.5 Result

```
[('green', datetime.time(23, 0))]
```

Cool!

Intrigued? Learn more how to make the magic happen in Howto: How to use Reparse.

Want to read more about what Reparse is and what it can do? More info in About: Why another tool for parsing?

CHAPTER 12

Info

12.1 Installation

12.1.1 pip

```
pip install reparse
```

12.1.2 manually

1. If you don't have them already, Reparse depends on REGEX, and PyYaml. Download those and `python setup.py install` in their directories. If you are on windows, you may have to find binary installers for these, since they contain modules that have to be compiled.
2. Download the Zip off of Github (or clone the repo).
3. Extract and do `python setup.py install` in the reparse containing the setup.py file directory. You can also just have the reparse/reparse directory in the source tree of your project if you don't want to install it.
4. Test with `python -c "import reparse"`, no output means it is probably installed. If you get `ImportError: No module named reparse` then you might want to recheck your steps.

12.2 Support

Need some help? Send me an email at theandychase@gmail.com and I'll do my best to help you.

12.3 Contribution

The code is located on Github. Send me suggestions, issues, and pull requests and I'll gladly review them!

12.4 Versions

- *3.0* InvalidPattern Exception, Allow monkey patching regex arguments. RE|PARSE -> Reparse.
- *2.1* Change *yaml.load* to *yaml.safe_load* for security

- *2.0* Major Refactor, Python 3, Better Parser builders
- *1.1* Fix setup.py
- *1.0* Release

12.5 Licence

MIT Licensed! See LICENSE file for the full text.