It slices, dices, and makes julienne data!

or, *Processing data with RecordStream, also known simply as* recs

Thomas Sibley — YAPC::NA 2015

Hi! Thanks for coming to my talk today.

Who am I?

My name is Thomas Sibley. I'm TSIBLEY on CPAN and trs on IRC and Twitter.

Mullins Lab @ the University of Washington

I'm Thomas Sibley, or TSIBLEY on CPAN and trs on IRC. Currently I work more or less as a staff programmer in the Mullins Lab, a microbiology research lab at the University of Washington. I handle a lot of poorly curated, but nevertheless important, datasets, and also have lots of ad-hoc day-to-day data processing needs.

recs will...

- bring consistency to data manipulation
- answer questions of your data quicker
- enhance correctness thru increased insight
- erase the guilt of using split and join instead of Text::CSV

I'm really excited to show you a better, faster, more consistent way to work with data, and I want you to come away with the knowledge to start using this fantastic toolset called recs. It'll save you time, provide tools to better validate your data, *and* you'll get to feel good about finally using Text::CSV for all those times you used split and join despite knowing all the ways it would break.

recs is something I came across right around the time I was starting a new job almost two years ago. It happened to be the perfect time to discover recs for me, because in my new job I was responsible for most of the data stewardship for the biology research lab I now work it. My primary job description was maintaining and developing an in-house clinical, research, and analysis data repository called Viroverse, but there was also a lot of ad-hoc support for the scientists dealing with smaller, on-the-fly datasets and transformations on those. I was quickly getting tired of writing giant messes of one liners and directories of scripts only used a few times and wanted something that let me build better pipelines. I shortly became involved maintaining and developing recs as I used it every day.

Let's look at what makes recs so useful.

RecordStream, or recs

is “a collection of command line tools for processing, analysing, and transforming data as streams of JSON records.”[1]

[1] https://metacpan.org/pod/App::RecordStream#SYNOPSIS

recs is the Unix coreutils for data. It’s a tool box to deal with the data you have — not the data you wish you had — and it strives to Just Work. Whereas many standard Unix tools like cut and sort and grep support tabular, delimited data, recs embraces a JSON stream format and provides a number of tools which consume and produce that format.

Stream format

- One JSON record per line

```
{"year": 2013, "city": "Austin"}
{"year": 2014, "city": "Orlando"}
{"year": 2015, "city": "Salt Lake City"}
```

The stream format itself super simple, and the *lingua franca* of all recs commands. Each record is a single-line JSON Object, and one line equals one record. That's all! This makes the record stream dead simple to redirect to a file or send across a network connection. It's important to note that the JSON records must be capital O Objects, or what we more sensibly call hashes in Perl. There's no limitation on the values of your keys, however, and it's perfectly fine to have nested data structures.

Commands at the edges of your pipeline handle marshalling of data to and from JSON streams and other formats, such as CSV, SQL, or access logs.

Commands

- Any data → Records (from*)
- Records → Records
- Records → Any data (to*)

The commands which make up recs can be classified into three primary groups: those that produce records from other formats (the *from* commands), those that produce other formats from records (the *to* commands), and those in between which transform records in some way. Let's look at the first group.

From commands

fromapache
fromatomfeed
fromcsv
fromdb
fromjsonarray
fromkv
frommongo
frommultire
fromps
fromre
fromsplit
fromtcpdump
fromxferlog
fromxml

These are the from commands, and they'll be your first step of using recs to read your existing data. recs comes with built-in support for a slew of formats and data sources, and I highlighted a few of the ones that I get the most use out of, but your mileage may vary! Since I work in a biology research lab, most of my day-to-day data is spreadsheets, databases as glorified spreadsheets, and ad-hoc formats. I left out a few custom formats specific to bioinformatics that aren't in core recs but that were easy for me to write commands to support. fromcsv is the real workhorse for me; it also handles TSV and anything else the Text::CSV module can parse. If all you use from recs is fromcsv, you're still better off than you were before!

To commands

tocsv
todb
togdgraph
tognuplot
tohtml
toprettyprint
toptable
totable

eval

Once your data is in recs, you want to know you can get it back out. The goal isn't to keep your data as streams of schema-less JSON records forever (unless you're into NoSQL).

For tabular data with any arbitrary delimiter, the primary output command is tocsv. totable prints a pretty, ASCII table which is indispensable when reviewing results or copying and pasting into an email. The common options for commands are standardized, so I'll often make a larger pipeline output a table in development/debug mode and a CSV in real use just by conditionalizing the command name.

In the case of recs, the eval command refers to evaluating a snippet of Perl for each record. It loops over input records and pushes arbitrary lines of output as returned by the code snippet run on each record. This is like your plain old Perl oneliner, but with the convenience of records as input, and it's just as handy.

Even with just the from and to commands you can start to do something useful, like look at a CSV...

```
$ cat slc-sky.csv
mean_observed,month,sky
5.6,January,Clear
6.5,January,"Partly Cloudy"
18.9,January,Cloudy
5.2,February,Clear
6.9,February,"Partly Cloudy"
16.1,February,Cloudy
7,March,Clear
8.1,March,"Partly Cloudy"
15.9,March,Cloudy
6.7,April,Clear
9.3,April,"Partly Cloudy"
```

...such as this one, as an aligned table that’s easier to read...

```
$ recs fromcsv --header slc-sky.csv \
    | recs totable -k month,sky,@mean
```

...by using fromcsv and totable.

```
$ recs fromcsv --header slc-sky.csv \
    | recs totable -k month,sky,@mean
month       sky             mean_observed
---------   -------------   -------------
January     Clear           5.6
January     Partly Cloudy   6.5
January     Cloudy          18.9
February    Clear           5.2
February    Partly Cloudy   6.9
February    Cloudy          16.1
March       Clear           7
March       Partly Cloudy   8.1
March       Cloudy          15.9
```

This example is a trivial data set of the average number of days of three sky conditions observed over Salt Lake City. Even with a small dataset, don't underestimate the power of simply being able to see the distribution and kind of values you have. This is especially true as the dataset grows more fields and more records.

The most interesting commands, though, are the ones that transform your records rather than just input and output them.

Transformational commands

annotate
assert
chain
collate
decollate
delta
flatten
generate
grep
join
multiplex
normalizetime
sort
stream2table
substream
xform

These commands are the heart of recs and provide powerful building blocks for manipulating and analysing your record stream. grep, sort, and join are all pretty much what you'd expect, except that grep takes a Perl snippet to evaluate for truthiness against each record.

collate is how you summarize and group records together and generate aggregate values like counts, sums, arrayrefs of records, and more. If you can use SQL's GROUP BY clause for it, you can probably use recs collate.

xform is a general purpose record transformer that can also operate on sliding windows, if you need it to. xform is both high and low-level enough that you could implement collate and grep in it if you wanted.

assert lets you explicitly state your assumptions about your data along the way, so that something useful happens when those assumptions are broken. It's basically grep but it dies when a record doesn't match. Combined with bash's pipefail option, for example, you can bail out of the entire pipeline with an error status if an assertion is broken.

Snippets

`$r` is the current record

```
$r->{key}

ref($r) eq "App::RecordStream::Record"

$r->rename("key", "newkey")
$r->prune_to("foo", "bar")
```

Many of these commands take arbitrary snippets of Perl to execute per-record or per-group of records. There are a few conveniences and conventions that make snippets easy. $r is always the current record, and can be accessed just like a hashref. It's actually an instance of the App::RecordStream::Record class too, which provides some nice utility methods. There are a bunch of basic methods to set and get fields as well as helpers to rename fields and prune the record to the set of fields you specify. prune_to is particularly useful with large records.

Snippets

```
      {{key}} eq $r->{key}
  {{sub/key}} eq $r->{sub}{key}
   {{key/#2}} eq $r->{key}[2]
    {{fuzzy}} eq $r->{fuzzy_wuzzy}
```

`-E` loads file as a snippet

`-M` works like perl's, including imports

The only syntax difference between snippets and vanilla Perl is the special double curly brace which does a fuzzy key lookup, using slashes for nested data structures. Keys that can't be found are autovivified, so you can use the double curlies to add new keys too. Fuzzy key matching can be triggered outside of a snippet by prefixing your key with an @ sign, which is often useful when specifying a keyspec for the dash k (-k) option.

Snippets are specified with the familiar -e option, although the -e is nearly always optional. Dash capital E (-E) loads a file as a snippet, for when you outgrow a single line. Dash capital M (-M) works just like perl's; it imports a module into your snippet.

For commands like grep, a snippet is the primary argument...

Slice!

```
recs grep '{{sky}} eq "Cloudy"
  and {{mean}} >= 10'

recs grep -v '{{sky}} =~ /Partly/'
recs grep    '{{sky}} !~ /Partly/'

recs sort -k sky,mean_observed=numeric
```

Grep will take any condition you want. It supports some of the most useful GNU grep options such as match inversion (dash v) and leading/trailing context (dash capital C, A, and B). Record context is accessed via the $A and $B arrayrefs in your snippet.

Sort will order by multiple keys, stringwise or numerically, forwards and reverse. It's not complicated, but still essential.

Dice!

```
recs xform -MMonth::Utils=name_to_num \
  '{{month_n}} = name_to_num({{month}})'

recs xform 'push_output(... ? $r : ())'
```

xform is the food processor for you data that lets you do whatever you want to your records. It takes a snippet which may or may not modify the record and outputs the new record. Think of it as map for your record streams.

xform also provides a couple snippet helper functions like push_output() and push_input(). For example, you can reimplement the grep command using push_output() to either push the current record or push nothing (which suppresses the normal xform behaviour of outputting the current record).

Julienne!

```
recs collate -k sky -a count

{"count":12,"sky":"Clear"}
{"count":12,"sky":"Partly Cloudy"}
{"count":12,"sky":"Cloudy"}
```

xform is great for handling individual records, but if you want to group and summarize your data, you want to use collate, the Japanese mandoline of recs. collate groups by a set of keys you provide and applies the aggregators you specify, if any, outputting a single record for each group.

Julienne!

```
recs collate -k sky

recs collate -k month -a sum,@mean
```

All the aggregate functions you might expect are there, such as a sum over a key.

Julienne!

```
{"month":"January","sum_@mean":31}
{"month":"February","sum_@mean":28.2}
{"month":"March","sum_@mean":31}
{"month":"April","sum_@mean":30}
{"month":"May","sum_@mean":30.9}
{"month":"June","sum_@mean":30}
{"month":"July","sum_@mean":31}
{"month":"August","sum_@mean":31}
```

and aggregators add new fields named after the aggregator's name and the aggregated fields.

If I wanted to verify that this data is sane, I could apply the assert command to this collation...

Julienne!

```
recs collate -k sky

recs collate -k month -a sum,@mean \
  | recs assert -v '{{sum}} <= 31'
```

...and check that the means for each month don't total more than 31, the maximum number of days that should be in any month. When I run this, assert finds a problem...

Julienne!

```
...
Assertion failed!
Expression: « {{sum}} <= 31 »
Filename: -
Line: 10
Record: $r = {
        'sum_@mean' => '31.1',
        'month' => 'October'
      };
```

and tells me that line 10 of the input had a record with a sum that failed my condition. In this case, it looks like there's probably some rounding errors in the data.

Julienne!

```
recs collate -k sky

recs collate -k month -a sum,@mean \
  | recs assert -v '{{sum}} <= 31'

recs collate -k month \
   -a maxmean=recordformax,@mean
```

There are also aggregators which make slightly more complicated things easy, like pulling out the record from a group with the maximum value for a particular key. This example uses the recordformax aggregator...

Julienne!

```
recs collate -k sky

recs collate -k month -a sum,@mean \
  | recs assert -v '{{sum}} <= 31'

recs collate -k month \
   -a maxmean=recordformax,@mean
```

specifying the field which fuzzily matches “mean”...

Julienne!

```
recs collate -k sky

recs collate -k month -a sum,@mean \
  | recs assert -v '{{sum}} <= 31'

recs collate -k month \
   -a maxmean=recordformax,@mean
```

...and stores the aggregated value, in this case the entire record, in a new field called maxmean. Specifying the name for our new aggregated value avoids a pretty ugly, but still functional, name.

Running this command produces 12 records, one for each month, that look like...

Julienne!

```
{
   "month" : "January",
   "maxmean" : {
      "month" : "January",
      "mean_observed" : "18.9",
      "sky" : "Cloudy"
   }
}
```

...this, if you prettify the single-line record with newlines and indents.

Julienne!

```
recs collate -k sky

recs collate -k month -a sum,@mean \
  | recs assert -v '{{sum}} <= 31'

recs collate -k month \
   -a maxmean=recordformax,@mean \
 | recs totable -k month,maxmean/sky
```

Instead of prettifying by hand, we could pipe the stream through the totable command with the fields of interest.

Julienne!

month	maxmean/sky	month	maxmean/sky
January	Cloudy	July	Clear
February	Cloudy	August	Clear
March	Cloudy	September	Clear
April	Cloudy	October	Clear
May	Cloudy	November	Cloudy
June	Clear	December	Cloudy

and see that Salt Lake is cloudy in the winter and sunny in the summer. No surprises there!

Aggregators

array	lastrec	sum
avg	linreg	uarray
countby	max	uconcat
concat	min	valuestokeys
correlation	mode	variance
count	percentile	
covariance	percentilemap	
dcount	recformax	
first	recformin	
firstrec	records	
last	stddev	

There are lots of aggregators that collate supports. You can get more info on each of these using the --show-aggregator option to the collate command. If you don't see the one you want, there's also easy ways to specify custom map-reduce and inject-into aggregators which pretty much let you build whatever you want using Perl snippets.

collate is a command that you'll use more as you become familiar with the best ways to apply it. Another advanced command is join...

Crossing the streams

```
month,high,low
January,37,21
February,43,26
March,53,33
April,61,39
May,71,47
June,82,56
July,91,63
August,89,62
```

...which lets us cross record streams. Suppose we have another data set for Salt Lake City that looks like this: average monthly high and low temperatures in degrees Fahrenheit. We want to combine it with our sky observations for each month. Despite what some people claim, joins are easy!

Crossing the streams

```
recs join month month \
  <(recs fromcsv --header slc-sky.csv) \
  <(recs fromcsv --header slc-temp.csv)
```

The join command takes four primary arguments, although you can omit the last one if you're piping records to stdin. The first two arguments are the names of the keys we should match records from each dataset on. In this case, we're using month names.

Crossing the streams

```
recs join month month \
  <(recs fromcsv --header slc-sky.csv) \
  <(recs fromcsv --header slc-temp.csv)
```

The next argument is the left-hand side of the join, and we're producing a record stream from a csv using bash's nice subshell redirection syntax.

Crossing the streams

```
recs join month month \
  <(recs fromcsv --header slc-sky.csv) \
  <(recs fromcsv --header slc-temp.csv)
```

We do the same thing for the temperature data, and then look at the results.

Crossing the streams

month	sky	mean_observed	high	low
January	Clear	5.6	37	21
January	Partly Cloudy	6.5	37	21
January	Cloudy	18.9	37	21
February	Clear	5.2	43	26
February	Partly Cloudy	6.9	43	26
February	Cloudy	16.1	43	26
March	Clear	7	53	33
March	Partly Cloudy	8.1	53	33
March	Cloudy	15.9	53	33
April	Clear	6.7	61	39

We now have each month's high and low merged into our sky records! The join command sensibly defaults to an inner join, but also supports left, right, and full outer joins.

Missing something? Write your own!

```
use JSON::MaybeXS;
while (<>) {
    my $r = decode_json($_);
    ...
    say encode_json($r);
}
```

By now you've seen a handful of commands that recs has to offer. One of recs best features though is it's flexibility: it provides a sensible, convenient core and lots of ways to extend it as long as you follow a minimal set of conventions. For example, if you can't find the command you want, it's easy to extend with your own commands. At the simplest, you can write a program where you fill in this ellipsis, and that's all! Decode a line of JSON, do some stuff, encode it, repeat.

Missing something? Write your own!

```
import sys, json
for line in sys.stdin:
    r = json.loads(line)
    ...
    json.dump(r, sys.stdout, \
        separators=(',', ':'))
    print
```

...or maybe it's easier to do what you want in Python! This is certainly true for some of the custom commands I wrote for bioinformatics work, where libraries like BioPython rule the day. recs is written in Perl, but there's no reason your commands have to be! There's even a subset of the standard collate command, called recs-fast-collate, which is written in C for speed that you can drop in to most pipelines if you're getting bogged down.

One of the important goals for recs is that it relies only on a minimal set of conventions, letting you use whatever data formats and languages make the most sense for you. The Unix philosophy of interoperation between small programs is firmly embraced by recs.

Missing something? Write your own!

```
package App::RecordStream::Operation::mogrify;
use base "App::RecordStream::Operation";

sub init { ... }
sub usage { ... }

sub accept_record {
    my ($self, $r) = @_;
    ...
    $self->push_record($r);
}
```

But maybe you want as much infrastructure support as a core recs command. In this case, you can write your own operation subclass that defines a few required methods. The recs command dispatcher will automatically pick up this class if it's in your @INC somewhere, just look at the output of `recs --list` to see if it's found. In this example, an operation named mogrify accepts records and outputs records.

Missing something? Write your own!

```
package App::RecordStream::Operation::mogrify;
use base "App::RecordStream::Operation";

sub init { ... }
sub usage { ... }

sub accept_line {
    my ($self, $r) = @_;
    ...
    $self->push_record($r);
}
```

You can also define an accept_line method if you want to handle textual input lines, or...

Missing something? Write your own!

```
package App::RecordStream::Operation::mogrify;
use base "App::RecordStream::Operation";

sub init { ... }
sub usage { ... }

sub accept_record {
    my ($self, $r) = @_;
    ...
    $self->push_line($r);
}
```

...or use the push_line method if you want to accept records but output in text lines. This is what the input/output edge commands do.

Getting recs

```
$ curl -fL# https://recs.pl > recs
$ chmod +x recs
$ ./recs --version
recs/4.0.14 (fatpacked)

$ cpanm --interactive App::RecordStream
```

github.com/benbernard/RecordStream

By this point hopefully you're excited to try out recs and give it a spin, and lucky for you recs is pretty darn easy to start using anywhere you find yourself needing it. While you have the option of a full CPAN-based install and choosing to install dependencies for optional features, the core of recs is pure-Perl and provided as a single standalone script.

In my experience, this portability is key for making recs a part of my daily data-diving life. Since it's easy to download and run no matter the Perl version, I no longer have to worry about copying data around from remote servers just so I can have the convenience of making sense of it using recs.

Getting help

- `recs help`
- `recs examples` & `recs story`
- `recs` *`command`* `--help-all`
- metacpan.org/release/App-RecordStream

- Open an issue on Github or rt.cpan.org
- #spug on irc.perl.org

The documentation for recs is pretty good, at least for the commands. One thing worth noting is that --help by default is an abbreviated form. There is often more help under --help-all or other sub-categories of help, so read the full option list printed by --help! Details on some of the trickier bits of recs — its tiny DSLs, for example — are behind some of these options.

If you still have problems, Ben Bernard, one of the original authors, and I have helped a few folks out via questions raised on Github issues or rt.cpan.org, so feel free to give that a shot if you're really stuck. Finally, there's at least two folks I know of, myself included, using recs on the #spug channel. It's a very quiet channel, but I idle there nonetheless.

Thanks!

Questions?

Thanks again so much for attending, and I hope after this you're as excited as I am about recs! If you have any questions, I'd love to answer them.