Structa 0.2.1 Documentation

Release 0.2.1

Dave Jones

Oct 26, 2021

CONTENTS

CHAPTER

ONE

INSTALLATION

structa is distributed in several formats. The following sections detail installation on a variety of platforms.

1.1 Ubuntu Linux

For Ubuntu Linux, it is simplest to install from the author's PPA[1] as follows (this also ensures you are kept up to date as new releases are made):

```
$ sudo add-apt-repository ppa://waveform/ppa
$ sudo apt update
$ sudo apt install structa
```

If you wish to remove structa:

```
$ sudo apt remove structa
```

1.2 Microsoft Windows

Firstly, install a version of Python 3[2] (this must be Python 3.5 or later), or ensure you have an existing installation of Python 3.

Ideally, for the purposes of following the *Getting Started* (page 3) you should add your Python 3 install to the system PATH variable so that python can be easily run from any command line.

You can install structa with the "pip" tool like so:

```
C:\Users\me> pip install structa
```

Upgrading structa can be done via pip too:

```
C:\Users\me> pip install --upgrade structa
```

And removal can be performed via pip:

```
C:\Users\me> pip uninstall structa
```

[1] https://launchpad.net/~waveform/+archive/ppa
[2] https://www.python.org/downloads/windows/

1.3 Other Platforms

If your platform is *not* covered by one of the sections above, structa is available from PyPI and can therefore be installed with the Python setuptools "pip" tool:

```
$ pip install structa
```

On some platforms you may need to use a Python 3 specific alias of pip:

```
$ pip3 install structa
```

If you do not have either of these tools available, please install the Python setuptools[3] package first.

You can upgrade structa via pip:

```
$ pip install --upgrade structa
```

And removal can be performed as follows:

```
$ pip uninstall structa
```

[3] https://pypi.python.org/pypi/setuptools/

CHAPTER

TWO

GETTING STARTED

Warning: Big fat "unfinished" warning: structa is still very much incomplete at this time and there's plenty of rough edges (like not showing CSV column titles).

If you run into unfinished stuff, do check the issues[4] first as I may have a ticket for that already. If you run into genuinely "implemented but broken" stuff, please do file an issue; it's these things I'm most interested in at this stage.

Getting the most out of structa is part science, part art. The science part is understanding how structa works and what knobs it has to twiddle. The art bit is figuring out what to twiddle them to!

2.1 Pre-requisites

You'll need the following to start this tutorial:

- A structa installation; see *Installation* (page 1) for more information on this.
- A Python 3 installation; given that structa requires this to run at all, if you've got structa installed, you've got this too. However, it'll help enormously if Python is in your system's "PATH" so that you can run python scripts at the command line.
- Some basic command line knowledge. In particular, it'll help if you're familiar with shell redirection and piping[5] (note: while that link is on askubuntu.com[6] the contents are equally applicable to the vast majority of UNIX shells, and even to Windows' cmd!)

2.2 Basic Usage

We'll start with some basic data structures and see how structa handles them. The following Python script dumps a list of strings representing integers to stdout in JSON format:

Listing 1: str-nums.py

```
import sys
import json

json.dump([str(i) for i in range(1000)] * 3, sys.stdout)
```

This produces output that looks (partially) like this:

[4] https://github.com/waveform80/structa/issues
[5] https://askubuntu.com/a/172989
[6] https://askubuntu.com/

```
["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13",
"14", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24", "25",
"26", "27", "28", "29", "30", "31", "32", "33", "34", "35", "36", "37",
"38", "39", "40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
"50", "51", "52", "53", "54", "55", "56", "57", "58", "59", "60", "61",
"62", "63", "64", "65", "66", "67", "68", "69", "70", "71", "72", "73",
"74", "75", "76", "77", "78", "79", "80", "81", "82", "83", "84", "85",
"86", "87", "88", "89", "90", "91", "92", "93", "94", "95", "96", "97",
"98", "99", "100", "101", "102", "103", "104", "105", "106", "107", "108",
"109", "110", "111", "112", "113", "114", "115", "116", "117", "118",
"119", "120", "121", "122", "123", "124", "125", "126", "127", "128",
"129", "130",
// lots more output...
]
```

We can capture the output in a file and pass this to structa:

```
$ python3 str-nums.py > str-nums.json
$ structa str-nums.json
[ str of int range=0..999 pattern="d" ]
```

Alternatively, we can pipe the output straight to structa:

```
$ python3 str-nums.py | structa
[ str of int range=0..999 pattern="d" ]
```

The output shows that the data contains a list (indicated by the square-brackets surrounding the output) of strings of integers ("str of int"), which have values between 0 and 999 (inclusive). The "pattern" at the end indicates that the strings are in decimal ("d") form (structa would also recognize octal, "o", and hexadecimal "x" forms of integers).

2.3 Bad Data (`--bad-threshold`)

Let's see how structa handles bad data. We'll add a non-numeric string into our list of numbers:

Listing 2: bad-nums.py

```
import sys
import json

json.dump(['foo'] + [str(i) for i in range(1000)] * 3, sys.stdout)
```

What does structa do in the presence of this "corrupt" data?

```
$ python3 bad-nums.py | structa
[ str of int range=0..999 pattern="d" ]
```

Apparently nothing! It may seem odd that structa raised no errors, or even warnings when encountering subtly incorrect data. However, structa has a "bad threshold" setting (*`structa --bad-threshold`* (page 18)) which means not all data in a given sequence has to match the pattern under test.

This setting defaults to 1% (or 0.01) meaning that up to 1% of the values can fail to match and the pattern will still be considered valid. If we lower the bad threshold to zero, this is what happens:

```
$ python3 bad-nums.py | structa --bad-threshold 0
[ str range="0".."foo" ]
```

It's still recognized as a list of strings, but no longer as string representations of integers.

How about mixing types? The following script outputs our errant string, "foo", along with a list of numbers. However, note that this time the numbers are integers, not strings of integers. In other words we have a list of a string, and lots

of integers:

Listing 3: bad-types.py

```
import sys
import json

json.dump(['foo'] + list(range(1000)) * 3, sys.stdout)
```

```
$ python3 bad-types.py | structa
[ value ]
```

In this case, even with the default 1% bad threshold, structa doesn't exclude the bad data; the analysis simply returns it as a list of mixed "values".

This is because structa assumes that the *types* of data are at least consistent and correct, under the assumption that if whatever is generating your data hasn't even got the data types right, you've got bigger problems! The bad threshold mechanism only applies to bad data *within* a homogenous type (typically bad string representations of numeric or boolean types).

2.4 Missing Data (`--empty-threshold`)

Another type of "bad" data commonly encountered is empty strings which are typically used to represent *missing* data, and (predictably) structa has another knob that can be twiddled for this: *structa --empty-threshold* (page 18). The following script generates a list of strings of integers in which most of the strings (~70%) are blank:

Listing 4: mostly-blank.py

```
import sys
import json
import random

json.dump([
    '' if random.random() < 0.7 else str(random.randint(0, 100))
    for i in range(10000)
], sys.stdout)
```

Despite the vast majority of the data being blank, structa handles this as normal:

```
$ python3 mostly-blank.py | structa
[ str of int range=0..100 pattern="d" ]
```

This is because the default for *structa --empty-threshold* (page 18) is 99% or 0.99. If the proportion of blank strings in a field exceeds the empty threshold, the field will simply be marked as a string without any further processing. Hence, when we re-run this script with the setting turned down to 50%, the output changes:

```
$ python3 mostly-blank.py | structa --empty-threshold 50%
[ str range=""."99" ]
```

Note: For those slightly confused by the above output: structa hasn't lost the "100" value, but because it's now considered a string (not a string of integers), "100" sorts before "99" alphabetically.

2.5 Fields or Tables (`--field-threshold`)

The next major knob that can be twiddled in structa is the *structa --field-threshold* (page 17). This is used to distinguish between mappings that act as a "table" (mapping keys to records) and mappings that act as a record (mapping field-names, typically strings, to their values).

To illustrate the difference between these, consider the following script:

Listing 5: simple-fields.py

```
import sys
import json
import random

json.dump({
    str(flight_id): {
        "flight_id": flight_id,
        "passengers": random.randint(50, 200),
        "from": random.choice([
            "MAN", "LON", "LHR", "ABZ", "AMS", "AUS", "BCN",
            "BER", "BHX", "BRU", "CHI", "ORK", "DAL", "EDI",
        ]),
    }
    for flight_id in range(200)
}, sys.stdout)
```

The generates a JSON file containing a mapping of mappings which looks something like this snippet (but with a lot more output):

```
{
  "0": { "flight_id": 0, "passengers": 53, "from": "BHX" },
  "1": { "flight_id": 1, "passengers": 157, "from": "AMS" },
  "2": { "flight_id": 2, "passengers": 118, "from": "DAL" },
  "3": { "flight_id": 3, "passengers": 111, "from": "MAN" },
  "4": { "flight_id": 4, "passengers": 192, "from": "BRU" },
  "5": { "flight_id": 5, "passengers": 69, "from": "DAL" },
  "6": { "flight_id": 6, "passengers": 147, "from": "LON" },
  "7": { "flight_id": 7, "passengers": 187, "from": "LON" },
  "8": { "flight_id": 8, "passengers": 171, "from": "AMS" },
  "9": { "flight_id": 9, "passengers": 89, "from": "DAL" },
  "10": { "flight_id": 10, "passengers": 169, "from": "LHR" },
  // lots more output...
}
```

The outer mapping is what structa would consider a "table" since it maps keys (in this case a string representation of an integer) to records. The inner mappings are what structa would consider "records" since they map a relatively small number of field names to values.

Note: Record fields don't have to be simple scalar values (although they are here); they can be complex structures including lists or indeed further embedded records.

If structa finds mappings with more keys than the threshold, those mappings will be treated as tables. However, if mappings are found with fewer (or equal) keys to the threshold, they will be analyzed as records. It's a rather arbitrary value that (unfortunately) usually requires some fore-knowledge of the data being analyzed. However, it's usually quite easy to spot when the threshold is wrong, as we'll see.

First, let's take a look at what happens when the threshold is set correctly. When passed to structa, with the default field threshold of 20, we see the following output:

```
$ python3 simple-fields.py | structa
{
    str of int range=0..199 pattern="d": {
        'flight_id': int range=0..199,
        'from': str range="ABZ".."ORK" pattern="Iii",
        'passengers': int range=50..200
    }
}
```

This indicates that structa has recognized the data as consisting of a mapping (indicated by the surrounding braces), which is keyed by a decimal string representation of an integer (in the range 0 to 199), and the values of which are another mapping with the keys "flight_id", "from", and "passengers".

The reason the inner mappings were treated as a set of records was because all those mappings had less than 20 entries. The outer mapping had more than 20 entries (200 in this case) and thus was treated as a table.

What happens if we force the field threshold down so low that the inner mappings are also treated as a table?

```
$ python3 simple-fields.py | structa --field-threshold 2
{
    str of int range=0..199 pattern="d": { str range="flight_id".."passengers":␣
↪value }
}
```

The inner mappings are now defined simply as mappings of strings (in the range "bar" to "id", sorted alphabetically) which map to "value" (an arbitrary mix of types). Anytime you see a mapping of `{ str: value }` in structa's output, it's a *fairly* good clue that *`structa --field-threshold`* (page 17) might be too low.

2.6 Merging structures (`--merge-threshold`)

2.7 Whitespace

CHAPTER

THREE

REAL WORLD DATA

Warning: Big fat "unfinished" warning: structa is still very much incomplete at this time and there's plenty of rough edges (like not showing CSV column titles).

If you run into unfinished stuff, do check the issues[7] first as I may have a ticket for that already. If you run into genuinely "implemented but broken" stuff, please do file an issue; it's these things I'm most interested in at this stage.

3.1 Pre-requisites

You'll need the following to start this tutorial:

- A structa installation; see *Installation* (page 1) for more information on this.
- A Python 3 installation; given that structa requires this to run at all, if you've got structa installed, you've got this too. However, it'll help enormously if Python is in your system's "PATH" so that you can run python scripts at the command line.
- The scipy[8] library must be installed for the scripts we're going to be using to generate data. On Debian/Ubuntu systems you can run the following:

```
$ sudo apt install python3-scipy
```

  On Windows, or if you're running in a virtual environment, you should run the following:

```
$ pip install scipy
```

- Some basic command line knowledge. In particular, it'll help if you're familiar with shell redirection and piping[9] (note: while that link is on askubuntu.com[10] the contents are equally applicable to the vast majority of UNIX shells, and even to Windows' cmd!)

[7] https://github.com/waveform80/structa/issues
[8] https://scipy.org/
[9] https://askubuntu.com/a/172989
[10] https://askubuntu.com/

3.2 "Real World" Data

For this tutorial, we'll use a custom made data-set which will allow us to tweak things and see what's going on under structa's hood a bit more easily.

The following script generates a fairly sizeable JSON file (~11MB) apparently recording various air quality readings from places which bear absolutely no resemblance whatsoever to my adoptive home city (ahem):

Listing 1: air-quality.py

```
import sys
import json
import random
import datetime as dt
from scipy.stats import skewnorm

readings = {
    # stat:  (min, max),
    'O3':    (0, 50),
    'NO':    (0, 200),
    'NO2':   (0, 100),
    'PM10':  (0, 100),
    'PM2.5': (0, 100),
}

locations = {
    # location: {stat: (skew, scale), ...}
    'Mancford Peccadillo': {
        'O3':    (0,  1),
        'NO':    (5,  1),
        'NO2':   (0,  1),
        'PM10':  (10, 3),
        'PM2.5': (10, 1),
    },
    'Mancford Shartson': {
        'O3':    (-10, 1),
        'NO':    (10,  1),
        'NO2':   (0,   1),
    },
    'Salport': {
        'NO':    (10,  1),
        'NO2':   (-10, 1/2),
        'PM10':  (5,   1/2),
        'PM2.5': (5,   1/2),
    },
    'Prestchester': {
        'O3':    (1,  1),
        'NO':    (5,  1/2),
        'NO2':   (0,  1),
        'PM10':  (5,  1/2),
        'PM2.5': (10, 1/2),
    },
    'Blackshire': {
        'O3':    (-10, 1),
        'NO':    (50,  1/2),
        'NO2':   (10,  1/2),
        'PM10':  (10,  1/2),
        'PM2.5': (10,  1/2),
    },
    'St. Wigpools': {
        'O3':    (0,  1),
        'NO':    (10, 1),
```

(continues on next page)

(continued from previous page)

```
        'NO2':   (5,  3/4),
        'PM10':  (5,  1/2),
        'PM2.5': (5,  1/2),
    },
}

def skewfunc(min, max, a=0, scale=1):
    s = skewnorm(a)
    real_min = s.ppf(0.0001)
    real_max = s.ppf(0.9999)
    real_range = real_max - real_min
    res_range = max - min
    def skewrand():
        return min + res_range * scale * (s.rvs() - real_min) / real_range
    return skewrand

generators = {
    location: {
        reading: skewfunc(read_min, read_max, skew, scale)
        for reading, params in loc_readings.items()
        for read_min, read_max in (readings[reading],)
        for skew, scale in (params,)
    }
    for location, loc_readings in locations.items()
}

timestamps = [
    dt.datetime(2020, 1, 1) + dt.timedelta(hours=n)
    for n in range(10000)
]

data = {
    location: {
        'euid': 'GB{:04d}A'.format(random.randint(200, 2000)),
        'ukid': 'UKA{:05d}'.format(random.randint(100, 800)),
        'lat': random.random() + 53.0,
        'long': random.random() - 3.0,
        'alt': random.randint(5, 100),
        'readings': {
            reading: {
                timestamp.isoformat(): loc_gen()
                for timestamp in timestamps
            }
            for reading, loc_gen in loc_gens.items()
        }
    }
    for location, loc_gens in generators.items()
}

json.dump(data, sys.stdout)
```

If you run the script it will output JSON on stdout, which you can redirect to a file (or straight to structa, but given the script takes a while to run you may wish to capture the output to a file for experimentation purposes). Passing the output to structa should produce output something like this:

```
$ python3 air-quality.py > air-quality.json
$ structa air-quality.json
{
    str range="Blackshire".."St. Wigpools": {
        'alt': int range=31..85,
        'euid': str range="GB1012A".."GB1958A" pattern="GB1[0-139][13-58][2-37-9]A
↪",
```

(continues on next page)

(continued from previous page)

```
        'lat': float range=53.29812..53.6833,
        'long': float range=-2.901626..-2.362118,
        'readings': {
            str range="NO".."PM2.5": { str of timestamp range=2020-01-01 00:00:00..
↪2021-02-20 15:00:00 pattern="%Y-%m-%dT%H:%M:%S": float range=-5.634479..335.6384␣
↪}
        },
        'ukid': str range="UKA00129".."UKA00713" pattern="UKA00[1-24-57][1-38][0-
↪13579]"
    }
}
```

Note: It should be notable that the output of structa looks rather similar to the end of the `air-quality.py` script, where the "data" variable that is ultimately dumped is constructed. This neatly illustrates the purpose of structa: to summarize repeating structures in a mass of hierarchical data.

Looking at this output we can see that the data consists of a mapping (or Javascript "object") at the top level, keyed by strings in the range "Blackshire" to "St. Wigpools" (when sorted).

Under these keys are more mappings which have six keys (which structa has displayed in alphabetical order for ease of reading):

- "alt" which maps to an integer in some range (in the example above 31 to 85, but this will likely be different for you)
- "euid" which maps to a string which always started with "GB" and is followed by several numerals
- "lat" which maps to a floating point value around 53
- "long" which maps to another floating point roughly around -2
- "ukid" which maps to a string always starting with UKA00 followed by several numerals
- And finally, "readings" which maps to another dictionary of strings …
- Which maps to *another* dictionary which is keyed by timestamps in string format, which map to floating point values

If you have a terminal capable of ANSI codes, you may note that types are displayed in a different color (to distinguish them from literals like the "ukid" and "euid" keys), as are patterns within fixed length strings, and various keywords like "range=".

You may also notice that several of the types (definitely the outer "str", but possibly other types within the top-level dictionary) are underlined. This indicates that these values are *unique* throughout the entire dataset (suitable as top-level keys if entered into a database).

3.3 Optional Keys

Let's explore how structa handles various "problems" in the data. Firstly, we'll make a copy of our script and add a chunk of code to remove approximately half of the altitude readings:

```
$ cp air-quality.py air-quality-opt.py
$ editor air-quality-opt.py
```

Listing 2: air-quality-opt.py

```
data = {
    location: {
        'euid': 'GB{:04d}A'.format(random.randint(200, 2000)),
```

(continues on next page)

(continued from previous page)

```
        'ukid': 'UKA{:05d}'.format(random.randint(100, 800)),
        'lat': random.random() + 53.0,
        'long': random.random() - 3.0,
        'alt': random.randint(5, 100),
        'readings': {
            reading: {
                timestamp.isoformat(): loc_gen()
                for timestamp in timestamps
            }
            for reading, loc_gen in loc_gens.items()
        }
    }
    for location, loc_gens in generators.items()
}

for location in data:
    if random.random() < 0.5:
        del data[location]['alt']

json.dump(data, sys.stdout)
```

What does structa make of this?

```
$ python3 air-quality-opt.py > air-quality-opt.json
$ structa air-quality-opt.json
{
    str range="Blackshire".."St. Wigpools": {
        'alt'?: int range=31..85,
        'euid': str range="GB1012A".."GB1958A" pattern="GB1[0-139][13-58][2-37-9]A
↪",
        'lat': float range=53.29812..53.6833,
        'long': float range=-2.901626..-2.362118,
        'readings': {
            str range="NO".."PM2.5": { str of timestamp range=2020-01-01 00:00:00..
↪2021-02-20 15:00:00 pattern="%Y-%m-%dT%H:%M:%S": float range=-5.634479..335.6384␣
↪}
        },
        'ukid': str range="UKA00129".."UKA00713" pattern="UKA00[1-24-57][1-38][0-
↪13579]"
    }
}
```

Note that a question-mark has now been appended to the "alt" key in the second-level dictionary (if your terminal supports color codes, this should appear in red). This indicates that the "alt" key is optional and not present in every single dictionary at that level.

3.4 "Bad" Data

Next, we'll make another script (a copy of `air-quality-opt.py`), which adds some more code to "corrupts" some of the timestamps:

```
$ cp air-quality-opt.py air-quality-bad.py
$ editor air-quality-bad.py
```

Listing 3: air-quality-bad.py

```
for location in data:
    if random.random() < 0.5:
```

(continues on next page)

(continued from previous page)

```
        reading = random.choice(list(data[location]['readings']))
        date = random.choice(list(data[location]['readings'][reading]))
        value = data[location]['readings'][reading].pop(date)
        data[location]['readings'][reading]['2020-02-31T12:34:56'] = value

json.dump(data, sys.stdout)
```

What does structa make of this?

```
$ python3 air-quality.py > air-quality-bad.json
$ structa air-quality-bad.json
{
    str range="Blackshire".."St. Wigpools": {
        'alt'?: int range=31..85,
        'euid': str range="GB1012A".."GB1958A" pattern="GB1[0-139][13-58][2-37-9]A
↪",
        'lat': float range=53.29812..53.6833,
        'long': float range=-2.901626..-2.362118,
        'readings': {
            str range="NO".."PM2.5": { str of timestamp range=2020-01-01 00:00:00..
↪2021-02-20 15:00:00 pattern="%Y-%m-%dT%H:%M:%S": float range=-5.634479..335.6384␣
↪}
        },
        'ukid': str range="UKA00129".."UKA00713" pattern="UKA00[1-24-57][1-38][0-
↪13579]"
    }
}
```

Apparently nothing! It may seem odd that structa raised no errors, or even warnings when encountering subtly incorrect data. One might (incorrectly) assume that structa just thinks anything that vaguely looks like a timestamp in a string is such.

For the avoidance of doubt, this is not the case: structa *does* attempt to convert timestamps correctly and does *not* think February 31st is a valid date (unlike certain databases!). However, structa does have a "bad threshold" setting (*`structa --bad-threshold`* (page 18)) which means not all data in a given sequence has to match the pattern under test.

3.5 Whitespace

By default, structa strips whitespace from strings prior to analysis. This is probably not necessary for the vast majority of modern datasets, but it's a reasonably safe default, and can be controlled with the *`structa --strip-whitespace`* (page 18) and *`structa --no-strip-whitespace`* (page 18) options in any case.

One other option that is affected by whitespace stripping is the "empty" threshold. This is the proportion of string values that are permitted to be empty (and thus ignored) when analysing a field of data. By default, this is 99% meaning the vast majority of a given field can be blank, and structa will still analyze the remaining strings to determine whether they represent integers, datetimes, etc.

If the proportion of blank strings in a field exceeds the empty threshold, the field will simply be marked as a string without any further processing.

For example:

Listing 4: examples/mostly-blank.py

```
import sys
import json
import random
```

(continues on next page)

(continued from previous page)

```
json.dump([
    '' if random.random() < 0.7 else str(random.randint(0, 100))
    for i in range(10000)
], sys.stdout)
```

This script outputs (as JSON) a list of strings of integers, roughly 70% of which will be blank. By default, structa is happy with this:

```
$ python3 mostly-blank.py | structa
[ str of int range=0..100 pattern="d" ]
```

However, if we force the empty threshold down below 70%:

CHAPTER

FOUR

COMMAND LINE REFERENCE

4.1 Synopsis

```
structa [-h] [--version] [-f {auto,csv,json,yaml}] [-e ENCODING]
        [--encoding-strict] [--no-encoding-strict]
        [-F INT] [-M NUM] [-B NUM] [-E NUM] [--str-limit NUM]
        [--hide-count] [--show-count] [--hide-lengths] [--show-lengths]
        [--hide-pattern] [--show-pattern]
        [--hide-range] [--show-range {hidden,limits,median,quartiles,graph}]
        [--hide-samples] [--show-samples]
        [--min-timestamp WHEN] [--max-timestamp WHEN]
        [--max-numeric-len LEN] [--sample-bytes SIZE]
        [--strip-whitespace] [--no-strip-whitespace]
        [--csv-format FIELD[QUOTE]] [--yaml-safe] [--no-yaml-safe]
        [file [file ...]]
```

4.2 Positional Arguments

file
: The data-file(s) to analyze; if this is - or unspecified then stdin will be read for the data; if multiple files are specified all will be read and analyzed as an array of similar structures

4.3 Optional Arguments

-h, --help
: show this help message and exit

--version
: show program's version number and exit

-f {auto,csv,json,yaml}, **--format** {auto,csv,json,yaml}
: The format of the data file; if this is unspecified, it will be guessed based on the first bytes of the file; valid choices are auto (the default), csv, or json

-e ENCODING, **--encoding** ENCODING
: The string encoding of the file, e.g. utf-8 (default: auto). If "auto" then the file will be sampled to determine the encoding (see *--sample-bytes* (page 18))

--encoding-strict, --no-encoding-strict
: Controls whether character encoding is strictly enforced and will result in an error if invalid characters are found during analysis. If disabled, a replacement character will be inserted for invalid sequences. The default is strict decoding

-F INT, **--field-threshold** INT
: If the number of distinct keys in a map, or columns in a tuple is less than this then they will be considered distinct fields instead of being lumped under a generic type like *str* (default: 20)

-M NUM, **--merge-threshold** NUM
: The proportion of mapping fields which must match other mappings for them to be considered potential merge candidates (default: 50%)

-B NUM, **--bad-threshold** NUM
: The proportion of string values which are allowed to mismatch a pattern without preventing the pattern from being reported; the proportion of "bad" data permitted in a field (default: 1%)

-E NUM, **--empty-threshold** NUM
: The proportion of string values permitted to be empty without preventing the pattern from being reported; the proportion of "empty" data permitted in a field (default: 99%)

--str-limit NUM
: The length beyond which only the lengths of strs will be reported; below this the actual value of the string will be displayed (default: 20)

--hide-count, --show-count
: If set, show the count of items in containers, the count of unique scalar values, and the count of all sample values (if *--show-samples* (page 18) is set). If disabled, counts will be hidden

--hide-lengths, --show-lengths
: If set, display the range of lengths of string fields in the same format as specified by *--show-range* (page 18)

--hide-pattern, --show-pattern
: If set, show the pattern determined for fixed length string fields. If disabled, pattern information will be hidden

--hide-range, --show-range {hidden,limits,median,quartiles,graph}
: Show the range of numeric (and temporal) fields in a variety of forms. The default is 'limits' which simply displays the minimum and maximum; 'median' includes the median between these; 'quartiles' shows all three quartiles between the minimum and maximum; 'graph' displays a crude chart showing the positions of the quartiles relative to the limits. Use *--hide-range* (page 18) to hide all range info

--hide-samples, --show-samples
: If set, show samples of non-unique scalar values including the most and least common values. If disabled, samples will be hidden

--min-timestamp WHEN
: The minimum timestamp to use when guessing whether floating point fields represent UNIX timestamps (default: 20 years). Can be specified as an absolute timestamp (in ISO-8601 format) or a duration to be subtracted from the current timestamp

--max-timestamp WHEN
: The maximum timestamp to use when guessing whether floating point fields represent UNIX timestamps (default: 10 years). Can be specified as an absolute timestamp (in ISO-8601 format) or a duration to be added to the current timestamp

--max-numeric-len LEN
: The maximum number of characters that a number, integer or floating-point, may use in its representation within the file. Defaults to 30

--sample-bytes SIZE
: The number of bytes to sample from the file for the purposes of encoding and format detection. Defaults to 1m. Typical suffixes of k, m, g, etc. may be specified

--strip-whitespace, --no-strip-whitespace
: Controls whether leading and trailing found in strings in the will be left alone and thus included or excluded in any data-type analysis. The default is to strip whitespace

--csv-format FIELD[QUOTE]
: The characters used to delimit fields and strings in a CSV file. Can be specified as a single character which will be used as the field delimiter, or two characters in which case the second will be used as the string quotation

character. Can also be "auto" which indicates the delimiters should be detected. Bear in mind that some characters may require quoting for the shell, e.g. ';"'

--yaml-safe, --no-yaml-safe
Controls whether the "safe" or "unsafe" YAML loader is used to parse YAML files. The default is the "safe" parser. Only use *--no-yaml-safe* (page 19) if you trust the source of your data

CHAPTER

FIVE

RECIPES

The following sections cover analyzing various common data scenarios with structa, and how structa's various options should be set to handle them.

5.1 Analyzing from a URL

While structa itself can't read URLs directly, the fact you can pipe data to it makes it ideal for use with something like curl[11]:

```
$ curl -s https://piwheels.org/packages.json | structa
[
    (
        str,
        int range=0..32.8K,
        int range=0..1.7M
    )
]
```

5.2 Dealing with large records

In the *Getting Started* (page 3) we saw the following script, which generates a mapping of mappings, for the purposes of learning about *`structa --field-threshold`* (page 17):

Listing 1: simple-fields.py

```
import sys
import json
import random

json.dump({
    str(flight_id): {
        "flight_id": flight_id,
        "passengers": random.randint(50, 200),
        "from": random.choice([
            "MAN", "LON", "LHR", "ABZ", "AMS", "AUS", "BCN",
            "BER", "BHX", "BRU", "CHI", "ORK", "DAL", "EDI",
        ]),
    }
    for flight_id in range(200)
}, sys.stdout)
```

We saw what happens when the threshold is too low:

[11] https://curl.se/

```
$ python3 simple-fields.py | structa --field-threshold 2
{
    str of int range=0..199 pattern="d": { str range="flight_id".."passengers":␣
↪value }
}
```

What happens if the threshold is set too high, resulting in the outer mapping being treated as a (very large!) record?

```
$ python3 simple-fields.py | structa --field-threshold 300
{
    str of int range=0..199 pattern="d": {
        'flight_id': int range=0..199,
        'from': str range="ABZ".."ORK" pattern="[A-EL-MO][A-EHMORU][IK-LNR-SUXZ]",
        'passengers': int range=50..199
    }
}
```

Curiously it seems to have worked happily anyway, although the pattern of the "from" field is now considerably more complex. The reasons for this are relatively complicated, but has to do with a later pass of structa's algorithm merging common sub-structures of records. The merging process unfortunately handles certain things (like the merging of string field patterns) rather crudely.

Hence, while it's generally safe to bump *structa --field-threshold* (page 17) up quite high whenever you need to, be aware that it will:

- significantly slow down analysis of large files (because the merging process is quite slow)
- complicate the pattern analysis of repeated string fields and a few other things (e.g. string representations of date-times)

In other words, whenever you find yourself in a situation where you need to bump up the field threshold, a reasonable procedure to follow is:

1. Bump the threshold very high (e.g. 1000) and run the analysis with *structa --show-count* (page 18) enabled.
2. Run the analysis again with the field threshold set below the count of the outer container(s), but above the count of the inner record mappings

The first run will probably be quite slow, but the second run will be much faster and will produce better output.

CHAPTER

SIX

API REFERENCE

In addition to being a utility, structa can also be used as an API from Python (either in a script, or just at the console).

The primary class of interest will generally be `Analyzer` in the *`structa.analyzer`* (page 23) module, but it is important to understand the various classes in the *`structa.types`* (page 24) module to interpret the output of the analyzer.

6.1 Modules

6.1.1 structa.analyzer

The *`structa.analyzer`* (page 23) module contains the `Analyzer` class which is the primary entry point for using structa's as an API. It can be constructed without any arguments, and the `analyze()` method can be immediately used to determine the structure of some data. The `merge()` method can be used to further refine the returned structure, and `measure()` can be used before-hand if you wish to use the *progress* callback to track the progress of long analysis runs.

A typical example of basic usage would be:

```
from structa.analyzer import Analyzer

data = {
    str(i): i
    for i in range(1000)
}
an = Analyzer()
structure = an.analyze(data)
print(structure)
```

The structure returned by `analyze()` (and by `merge()`) will be an instance of one of the classes in the *`structa.types`* (page 24) module, all of which have sensible `str`[12] and `repr()`[13] output.

A more complete example, using `Source` to figure out the source format and encoding:

```
from structa.analyzer import Analyzer
from structa.source import Source
from urllib.request import urlopen

with urlopen('https://usn.ubuntu.com/usn-db/database-all.json') as f:
    src = Source(data)
    an = Analyzer()
    an.measure(src.data)
    structure = an.analyze(src.data)
    structure = an.merge(structure)
    print(structure)
```

[12] https://docs.python.org/3.8/library/stdtypes.html#str
[13] https://docs.python.org/3.8/library/functions.html#repr

6.1.2 structa.chars

The *structa.chars* (page 24) module provides classes and constants for defining and manipulating character classes (in the sense of regular expressions[14]). The primary class of interest is `CharClass`, but most uses can likely be covered by the set of constants defined in the module.

Constants

`structa.chars.`**`oct_digit`**
Represents any valid digit in base 8 (octal).

`structa.chars.`**`dec_digit`**
Represents any valid digit in base 10 (decimal).

`structa.chars.`**`hex_digit`**
Represents any valid digit in base 16 (hexidecimal).

`structa.chars.`**`ident_first`**
Represents any character which is valid as the first character of a Python identifier.

`structa.chars.`**`ident_char`**
Represents any character which is valid within a Python identifier.

`structa.chars.`**`any_char`**
Represents any valid character (an instance of `AnyChar`).

6.1.3 structa.collections

6.1.4 structa.conversions

6.1.5 structa.errors

The *structa.errors* (page 24) module defines all the custom exception and warning classes used in structa.

6.1.6 structa.format

The *structa.format* (page 24) module contains various simple routines for "nicely" formatting certain structures for output.

6.1.7 structa.source

6.1.8 structa.types

The *structa.types* (page 24) module defines the class hierarchy used to represent the structural types of analyzed data. The root of the hierarchy is the `Type` class. The rest of the hierarchy is illustrated in the chart below:

[14] https://en.wikipedia.org/wiki/Regular_expression

6.1.9 structa.xml

The *structa.xml* (page 25) module provides methods for generating and manipulating XML, primarily in the form of `xml.etree.ElementTree`[15] objects. The main class of interest is `ElementFactory`, which can be used to generate entire element-tree documents in a functional manner.

The `xml()` function can be used in a similar manner to `str`[16] or `repr()`[17] to generate XML representations of supported objects (most classes within *structa.types* (page 24) support this). Finally, `get_transform()` can be used to obtain XSLT trees defined by structa (largely for display purposes).

[15] https://docs.python.org/3.8/library/xml.etree.elementtree.html#module-xml.etree.ElementTree
[16] https://docs.python.org/3.8/library/stdtypes.html#str
[17] https://docs.python.org/3.8/library/functions.html#repr

CHAPTER

SEVEN

DEVELOPMENT

The main GitHub repository for the project can be found at:

https://github.com/waveform80/structa

The project is currently in its early stages, but is quite useable and the documentation, while incomplete, should be useful to both users and developers wishing to hack on the project itself. The test suite is also nearing full coverage.

7.1 Development installation

If you wish to develop structa, obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development. The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt install build-essential git virtualenvwrapper
```

After installing `virtualenvwrapper` you'll need to restart your shell before commands like **`mkvirtualenv`** will operate correctly. Once you've restarted your shell, continue:

```
$ cd
$ mkvirtualenv -p /usr/bin/python3 structa
$ workon structa
(structa) $ git clone https://github.com/waveform80/structa.git
(structa) $ cd structa
(structa) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ workon structa
(structa) $ cd ~/structa
(structa) $ git pull
(structa) $ make develop
```

To remove your installation, destroy the sandbox and the clone:

```
(structa) $ deactivate
$ rmvirtualenv structa
$ rm -rf ~/structa
```

7.2 Building the docs

If you wish to build the docs, you'll need a few more dependencies. Inkscape is used for conversion of SVGs to other formats, Graphviz is used for rendering certain charts, and TeX Live is required for building PDF output. The following command should install all required dependencies:

```
$ sudo apt install texlive-latex-recommended texlive-latex-extra \
    texlive-fonts-recommended texlive-xetex graphviz inkscape \
    python3-sphinx python3-sphinx-rtd-theme latexmk xindy
```

Once these are installed, you can use the "doc" target to build the documentation in all supported formats (HTML, ePub, and PDF):

```
$ workon structa
(structa) $ cd ~/structa
(structa) $ make doc
```

However, the easiest way to develop the documentation is with the "preview" target which will build the HTML version of the docs, and start a web-server to preview the output. The web-server will then watch for source changes (in both the documentation source, and the application's source) and rebuild the HTML automatically as required:

```
$ workon structa
(structa) $ cd ~/structa
(structa) $ make preview
```

The HTML output is written to `build/html` while the PDF output goes to `build/latex`.

7.3 Test suite

If you wish to run the structa test suite, follow the instructions in *Development installation* (page 27) above and then make the "test" target within the sandbox:

```
$ workon structa
(structa) $ cd ~/structa
(structa) $ make test
```

The test suite is also setup for usage with the **tox** utility, in which case it will attempt to execute the test suite with all supported versions of Python. If you are developing under Ubuntu you may wish to look into the Dead Snakes PPA[18] in order to install old/new versions of Python; the tox setup *should* work with the version of tox shipped with Ubuntu Focal, but more features (like parallel test execution) are available with later versions.

For example, to execute the test suite under tox, skipping interpreter versions which are not installed:

```
$ tox -s
```

To execute the test suite under all installed interpreter versions in parallel, using as many parallel tasks as there are CPUs, then displaying a combined report of coverage from all environments:

```
$ tox -p auto -s
$ coverage combine .coverage.py*
$ coverage report
```

[18] https://launchpad.net/~deadsnakes/%2Barchive/ubuntu/ppa

CHAPTER

EIGHT

CHANGELOG

8.1 Release 0.2.1 (2021-08-17 ... later)

- It'd help if you included the XSL for the UI …

8.2 Release 0.2 (2021-08-17)

- Better tuple analysis (#4[19]) which was a pre-requisite for…
- Added CSV support (#5[20])
- Added some pretty progress output (#6[21])
- Prettier output (#8[22])
- Added documentation (#9[23])
- Added YAML support (#10[24])
- Better elimination of common sub-trees (#12[25])
- Multi-file input support (#15[26])

8.3 Release 0.1 (2018-12-17)

- Initial commit of something that works … ish

[19] https://github.com/waveform80/structa/issues/4
[20] https://github.com/waveform80/structa/issues/5
[21] https://github.com/waveform80/structa/issues/6
[22] https://github.com/waveform80/structa/issues/8
[23] https://github.com/waveform80/structa/issues/9
[24] https://github.com/waveform80/structa/issues/10
[25] https://github.com/waveform80/structa/issues/12
[26] https://github.com/waveform80/structa/issues/15

CHAPTER

NINE

LICENSE

PYTHON MODULE INDEX

S

INDEX

Symbols

A

D

F

H

I