Structa 0.3 Documentation

Release 0.3

Dave Jones

Oct 28, 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/structa
$ 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 20)) 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 20). 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 20) 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.

It is also worth nothing that, 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 20) and *structa --no-strip-whitespace* (page 20) options in any case.

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

The next major knob that can be twiddled in structa is the *structa --field-threshold* (page 19). 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 "flight_id" to "passengers", 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 19) might be too low.

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

The final major knob available for twiddling is the *`structa --merge-threshold`* (page 20) which dictates how similar record mappings have to be in order to be considered for merging. This only applies to mappings at the same "level" with similar (but not necessarily perfectly identical) structures.

To illustrate, consider the following example script:

Listing 6: merge-dicts.py

```
import sys
import json
import random

airports = {
    "MAN", "LON", "LHR", "ABZ", "AMS", "AUS", "BCN",
    "BER", "BHX", "BRU", "CHI", "ORK", "DAL", "EDI",
}

facilities = [
    "WiFi", "Shopping", "Conferences", "Chapel", "Parking",
    "Lounge", "Spotters Area", "Taxi Rank", "Train Station",
    "Tram Stop", "Bus Station", "Duty Free",
]

data = {
    airport: {
        "code": airport,
        "facilities": random.sample(
            facilities, random.randint(3, len(facilities))),
        "terminals": random.randint(1, 4),
        "movements": random.randint(10000, 300000),
```

(continues on next page)

(continued from previous page)

```
        "passengers": random.randint(1000000, 30000000),
        "cargo": random.randint(10000, 1000000),
    }
    for airport in airports
}

for entry in data.values():
    # Exclude reporting terminals if the airport only has one
    if entry['terminals'] == 1:
        del entry['terminals']
    # Exclude some other stats semi-randomly
    if random.random() > 0.7:
        del entry['movements']
    if random.random() > 0.9:
        del entry['cargo']

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

In keeping with the prior examples, this generates a list of airports with associated statistics. When we run the results through structa they seem to produce sensible output:

```
$ python3 merge-dicts.py | structa
{
    str range="ABZ".."ORK" pattern="Iii": {
        'cargo'?: int range=55.0K..949.1K,
        'code': str range="ABZ".."ORK" pattern="[A-EL-MO][A-EHMORU][IK-LNR-SUXZ]",
        'facilities': [ str range="Bus Station".."WiFi" ],
        'movements'?: int range=10.0K..295.7K,
        'passengers': int range=1.0M..24.9M,
        'terminals'?: int range=2..4
    }
}
```

However, there are several things to note about the data:

- The number of top-level entries (the airport codes) is less than the default field threshold (20). This means that the "outer" mapping will initially be treated as a record rather than a table (see the explanation of `--field-threshold` above).
- In some entries, statistics are missing. When "terminals" would be 1, it's excluded, and 30% and 10% of entries will be missing their "movements" and "cargo" stats respectively.
- The "code", "facilities", and "passengers" entries are *always* present out of a total of 6 fields that *could* be present. This means that at least 50% of all the fields are guaranteed to be present, which is the default level of `--merge-threshold`.

As noted above, structa's initial pass will treat the outer mapping as a record so each airport will be analyzed as a separate entity. After this phase a first merge pass will run, which will compare all the airport records. After concluding that all contain at least 50% of the same fields as the rest, and that all field values found are compatible, those rows will be merged. What happens if we raise the merge threshold to 100%, which would require that every single airport record shared exactly the same fields?

```
$ python3 docs/examples/merge-dicts.py | structa --merge-threshold 100%
{
    'ABZ': {
        'cargo': int range=192.6K,
        'code': str range="ABZ" pattern="ABZ",
        'facilities': [ str range="Bus Station".."WiFi" ],
        'passengers': int range=27.5M,
        'terminals': int range=4
    },
    'AMS': {
```

(continues on next page)

(continued from previous page)

```
        'cargo': int range=606.4K,
        'code': str range="AMS" pattern="AMS",
        'facilities': [ str range="Bus Station".."WiFi" ],
        'movements': int range=132.5K,
        'passengers': int range=4.8M,
        'terminals': int range=3
    },
    'AUS': {
        'cargo': int range=607.4K,
        'code': str range="AUS" pattern="AUS",
        'facilities': [ str range="Bus Station".."WiFi" ],
        'movements': int range=212.2K,
        'passengers': int range=13.7M
    },
    ...
```

A whole lot of output! When you get excessively large output consisting of largely (but not completely) similar records, it's a reasonable sign that *structa --merge-threshold* (page 20) is set too high.

That said, the merge threshold is fairly forgiving. The specific algorithm used is as follows:

- For two given mappings, find the length (number of fields) of the shortest mapping.
- Calculate the minimum required number of common fields as the merge threshold percentage of the shortest length. For example, if the shortest mapping contains 8 fields, and the merge threshold is 50%, then there must be at least 4 common fields.
- Note that in the case that one side is an empty mapping this will *always* permit the match as at least 0 common fields will be required percentage of the shortest length.

2.7 Other Switches

There are quite a few other switches in structa, but all are less important than the four covered in the prior sections. The rest largely have to do with specific formats (*structa --csv-format* (page 20) for CSV files, *structa --no-json-strict* (page 21) for JSON files), the character encoding of files (*structa --encoding* (page 19), *structa --encoding-strict* (page 19)), or tweaking the style of the output (*structa --show-count* (page 20), *structa --show-lengths* (page 20)).

2.7.1 Integer Handling

However, there are a couple that may be important for specific types of data. The first is *structa --max-numeric-len* (page 20) which dictates the maximum number of digits structa will consider as a number. This defaults to 30 which is more than sufficient to represent all 64-bit integer values (which only require 20 digits), with some lee-way for data that includes large integers (which Python handles happily).

However, the default is deliberately lower than 32 because at that point, data which includes hex-encoded hash values (MD5[7], SHA1[8], etc.) typically wind up mis-representing those hashes as literal integers (which, technically, they are, but that's not typically how users wish hash values to be interpreted).

[7] https://en.wikipedia.org/wiki/MD5
[8] https://en.wikipedia.org/wiki/SHA-1

2.7.2 Date Handling

The other important switches are those used in the detection of dates encoded as numbers: *structa --min-timestamp* (page 20) and *structa --max-timestamp* (page 20). When dates are encoded as (potentially fractional) day-offsets from the UNIX epoch (the 1st January, 1970), how does structa determine that it's looking at a set of dates rather than a set of numbers?

In a typical set of (arbitrary) numbers, it's quite normal to find "0" or "1" commonly represented, or for the set of numbers to span over a large range (consider file-sizes which might span over millions or billions of bytes). However, most date-based sets, *don't* tend to include values around the 1st or 2nd of January, 1970 (most data that's dealt with is, to some degree, fairly contemporary), and moreover tends to cluster around values that vary by no more than a few thousand (after all 3000 is enough to represent nearly a decade's worth of days).

Thus if we find that all numbers in a given set fall within some "reasonable" limits (structa defaults to 20 years prior, and 10 years after the current date) it's a *reasonable guess* that we're looking at dates encoded as numbers rather than an arbitrary set of numbers.

2.8 Conclusion

At this point, you should have a pretty good idea of the major controls that structa provides, what they do, and the circumstances under which you will need to fiddle with them. The *next tutorial* (page 11) goes through a variety of scenarios with some datasets that are closer to the sort of size and complexity one might encounter in the real world.

However, it won't be introducing any new functionality that we haven't covered above and at this point you may simply want to take structa for a spin with your own datasets.

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[9] 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[10] 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[11] (note: while that link is on askubuntu.com[12] the contents are equally applicable to the vast majority of UNIX shells, and even to Windows' cmd!)

[9] https://github.com/waveform80/structa/issues
[10] https://scipy.org/
[11] https://askubuntu.com/a/172989
[12] 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 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=".

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

Just because you *can* use something as a unique key, however, doesn't mean you *should* (floating point values being a classic example).

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)),
        '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 "corrupt" 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:
        reading = random.choice(list(data[location]['readings']))
        date = random.choice(list(data[location]['readings'][reading]))
        value = data[location]['readings'][reading].pop(date)
        # Change the date to the 31st of February...
        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 20)) which means not all data in a given sequence has to match the pattern under test.

3.5 Multiple Inputs

Time for another script (based on a copy of the prior `air-quality-bad.py` script), which produces each location as its own separate JSON file:

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

Listing 4: air-quality-multi.py

```
for location in data:
    filename = location.lower().replace(' ', '-').replace('.', '')
    filename = 'air-quality-{filename}.json'.format(filename=filename)
    with open(filename, 'w') as out:
        json.dump({location: data[location]}, out)
```

We can pass all the files as inputs to structa simultaneously, which will cause it to assume that they should all be processed as if they have comparable structures:

```
$ python3 air-quality-multi.py
$ ls *.json
air-quality-blackshire.json           air-quality-prestchester.json
air-quality-mancford-peccadillo.json  air-quality-salport.json
air-quality-mancford-shartson.json    air-quality-st-wigpools.json
$ structa air-quality-*.json
{
    str range="Blackshire".."St. Wigpools": {
        'alt': int range=15..92,
        'euid': str range="GB0213A".."GB1029A" pattern="GB[01][028-9][1-26-7][2-
↪379]A",
        'lat': float range=53.49709..53.98315,
        'long': float range=-2.924566..-2.021445,
        '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=-2.982586..327.4161␣
↪}
        },
        'ukid': str range="UKA00148".."UKA00786" pattern="UKA00[135-7][13-47-8][06-
↪9]"
    }
}
```

In this case, structa has merged the top-level mapping in each file into one large top-level mapping. It would do the same if a top-level list were found in each file too.

3.6 Conclusion

This concludes the structa tutorial series. You should now have some experience of using structa with more complex datasets, how to tune its various settings for different scenarios, and what to look out for in the results to get the most out of its analysis.

In other words, if you wish to use structa from the command line, you should be all set. If you want help dealing with some specific scenarios, the sections in *Recipes* (page 23) may be of interest. Alternatively, if you wish to use structa in your own Python scripts, the *API Reference* (page 25) may prove useful.

Finally, if you wish to hack on structa yourself, please see the *Development* (page 43) chapter for more information.

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 20))

--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 20) 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 20)

--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 20) 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 21) if you trust the source of your data

--json-strict, --no-json-strict
Controls whether the JSON decoder permits control characters within strings, which isn't technically valid JSON. The default is to be strict and disallow such characters

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[13]:

```
$ 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 19):

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:

[13] 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 19) 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 20) 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* (page 25) in the *structa.analyzer* (page 25) module, but it is important to understand the various classes in the *structa.types* (page 33) module to interpret the output of the analyzer.

6.1 Modules

6.1.1 structa.analyzer

The *structa.analyzer* (page 25) module contains the *Analyzer* (page 25) class which is the primary entry point for using structa's as an API. It can be constructed without any arguments, and the *analyze()* (page 26) method can be immediately used to determine the structure of some data. The *merge()* (page 26) method can be used to further refine the returned structure, and *measure()* (page 26) 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()* (page 26) (and by *merge()* (page 26)) will be an instance of one of the classes in the *structa.types* (page 33) module, all of which have sensible `str`[14] and `repr()`[15] output.

A more complete example, using *Source* (page 32) 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)
```

[14] https://docs.python.org/3.9/library/stdtypes.html#str
[15] https://docs.python.org/3.9/library/functions.html#repr

class structa.analyzer.**Analyzer**(*, *bad_threshold=Fraction(1, 50)*, *empty_threshold=Fraction(49, 50)*, *field_threshold=20*, *merge_threshold=Fraction(1, 2)*, *max_numeric_len=30*, *strip_whitespace=False*, *min_timestamp=None*, *max_timestamp=None*, *progress=None*)

This class is the core of structa. The various keyword-arguments to the constructor correspond to the command line options (see *Command Line Reference* (page 19)).

The `analyze()` (page 26) method is the primary method for analysis, which simply accepts the data to be analyzed. The `measure()` (page 26) method can be used to perform some pre-processing for the purposes of progress reporting (useful with very large datasets), while `merge()` (page 26) can be used for additional post-processing to improve the analysis output.

Parameters

- **bad_threshold** (`numbers.Rational`[16]) – The proportion of data within a field (across repetitive structures) which is permitted to be invalid without affecting the type match. Primarily useful with string representations. Valid values are between 0 and 1.
- **empty_threshold** (`numbers.Rational`[17]) – The proportion of strings within a field (across repetitive structures) which can be blank without affecting the type match. Empty strings falling within this threshold will be discounted by the analysis. Valid values are between 0 and 1.
- **field_threshold** (`int`[18]) – The minimum number of fields in a mapping before it will be treated as a "table" (a mapping of keys to records) rather than a record (a mapping of fields to values). Valid values are any positive integer.
- **merge_threshold** (`numbers.Rational`[19]) – The proportion of fields within repetitive mappings that must match for the mappings to be considered "mergeable" by the `merge()` (page 26) method. Note that the proportion is calculated with the length of the *shorter* mapping in the comparision. Valid values are between 0 and 1.
- **strip_whitespace** (`bool`[20]) – If `True`[21], whitespace is stripped from all strings prior to any further analysis.
- **min_timestamp** (`datetime.datetime`[22] `or None`[23]) – The minimum timestamp to use when determining whether floating point values potentially represent epoch-based datetime values.
- **max_timestamp** (`datetime.datetime`[24] `or None`[25]) – The maximum timestamp to use when determining whether floating point values potentially represent epoch-based datetime values.
- **progress** (`object`[26] `or None`[27]) – If specificed, must be an object with `update` and `reset` methods that will be called to provide progress feedback. See `progress` (page 27) for further details.

analyze(*data*)

Given some value *data* (typically an iterable or a mapping), return a `Type` descendent describing its structure.

measure(*data*)

Given some value *data* (typically an iterable or mapping), measure the number of items within it, for the purposes of accurately reporting progress during the running of the `analyze()` (page 26) and `merge()` (page 26) methods.

If this is not called prior to these methods, they will still run successfully, but progress tracking (via the `progress` (page 27) object) will be inaccurate as the total number of steps to process will never be calculated.

As measurement is itself a potentially lengthy process, progress will be reported as a function of the top-level items within *data* during the run of this method.

merge(*struct*)

Given some *struct* (as returned by `analyze()` (page 26)), merge common sub-structures within it, returning the new top level structure (another `Type` (page 34) instance).

property progress

The object passed as the *progress* parameter on construction.

If this is not `None`[28], it must be an object which implements the following methods:

- `reset(*, total: int=None)`
- `update(n: int=None)`

The "reset" method of the object will be called with either the keyword argument "total", indicating the new number of steps that have yet to complete, or with no arguments indicating the progress display should be cleared as a task is complete.

The "update" method of the object will be called with either the number of steps to increment by (as the positional "n" argument), or with no arguments indicating that the display should simply be refreshed (e.g. to recalculate the time remaining, or update a time elapsed display).

It is no coincidence that this is a sub-set of the public API of the tqdm[29] progress bar project (as that's what structa uses in its CLI implementation).

6.1.2 structa.chars

The *structa.chars* (page 27) module provides classes and constants for defining and manipulating character classes (in the sense of regular expressions[30]). The primary class of interest is *CharClass* (page 27), but most uses can likely be covered by the set of constants defined in the module.

class structa.chars.**CharClass**(*chars*)

A descendent of `frozenset`[31] intended to represent a character class in a regular expression. Can be instantiated from any iterable of single characters (including a `str`[32]).

All operations of `frozenset`[33] are supported, but return instances of *CharClass* (page 27) instead (and thus, are only valid for operations which result in sets containing individual character values). For example:

```
>>> abc = CharClass('abc')
>>> abc
CharClass('abc')
>>> ghi = CharClass('ghi')
>>> abc == ghi
False
>>> abc < ghi
False
>>> abc | ghi
CharClass('abcghi')
>>> abc < abc | ghi
True
```

difference(**others*)

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

[16] https://docs.python.org/3.9/library/numbers.html#numbers.Rational
[17] https://docs.python.org/3.9/library/numbers.html#numbers.Rational
[18] https://docs.python.org/3.9/library/functions.html#int
[19] https://docs.python.org/3.9/library/numbers.html#numbers.Rational
[20] https://docs.python.org/3.9/library/functions.html#bool
[21] https://docs.python.org/3.9/library/constants.html#True
[22] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[23] https://docs.python.org/3.9/library/constants.html#None
[24] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[25] https://docs.python.org/3.9/library/constants.html#None
[26] https://docs.python.org/3.9/library/functions.html#object
[27] https://docs.python.org/3.9/library/constants.html#None
[28] https://docs.python.org/3.9/library/constants.html#None
[29] https://pypi.org/project/tqdm/
[30] https://en.wikipedia.org/wiki/Regular_expression

intersection(**others*)
Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

symmetric_difference(**others*)
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

union(**others*)
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

class structa.chars.**AnyChar**
A singleton class (all instances are the same) which represents any possible character. This is comparable with, and compatible in operations with, instances of *CharClass* (page 27). For instance:

```
>>> abc = CharClass('abc')
>>> any_ = AnyChar()
>>> any_
AnyChar()
>>> abc < any_
True
>>> abc > any_
False
>>> abc | any_
AnyChar()
```

structa.chars.**char_range**(*start*, *stop*)
Returns a *CharClass* (page 27) containing all the characters from *start* to *stop* inclusive (in unicode codepoint order). For example:

```
>>> char_range('a', 'c')
CharClass('abc')
>>> char_range('0', '9')
CharClass('0123456789')
```

Parameters

- **start** (*str*[34]) – The inclusive start point of the range
- **stop** (*str*[35]) – The inclusive stop point of the range

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.

[31] https://docs.python.org/3.9/library/stdtypes.html#frozenset
[32] https://docs.python.org/3.9/library/stdtypes.html#str
[33] https://docs.python.org/3.9/library/stdtypes.html#frozenset
[34] https://docs.python.org/3.9/library/stdtypes.html#str
[35] https://docs.python.org/3.9/library/stdtypes.html#str

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* (page 28)).

6.1.3 structa.collections

class structa.collections.**FrozenCounter**(*it*)
An immutable variant of the collections.Counter[36] class from the Python standard library.

This implements all readable properties and behaviours of the collections.Counter[37] class, but excludes all methods and behaviours which permit modification of the counter. The resulting instances are hashable and can be used as keys in mappings.

elements()
See collections.Counter.elements()[38].

classmethod from_counter(*counter*)
Construct a *FrozenCounter* (page 29) from a collections.Counter[39] instance. This is generally much faster than attempting to construct from the elements of an existing counter.

The *counter* parameter must either be a collections.Counter[40] instance, or a *FrozenCounter* (page 29) instance (in which case it is returned verbatim).

most_common(*n=None*)
See collections.Counter.most_common()[41].

6.1.4 structa.conversions

structa.conversions.**try_conversion**(*sample*, *conversion*, *threshold=0*)
Given a Counter[42] *sample* of strings, call the specified *conversion* on each string returning the set of converted values.

conversion must be a callable that accepts a single string parameter and returns the converted value. If the *conversion* fails it must raise a ValueError[43] exception.

If *threshold* is specified (defaults to 0), it defines the number of "bad" conversions (which result in ValueError[44] being raised) that will be ignored. If *threshold* is exceeded, then ValueError[45] will be raised (or rather passed through from the underlying *conversion*). Likewise, if *threshold* is not exceeded, but zero conversions are successful then ValueError[46] will also be raised.

structa.conversions.**parse_bool**(*s*, *false='0'*, *true='1'*)
Convert the string *s* (stripped and lower-cased) to a bool, if it matches either the *false* string (defaults to '0') or *true* (defaults to '1'). If it matches neither, raises a ValueError[47].

structa.conversions.**parse_duration**(*s*)
Convert the string *s* to a relativedelta. The string must consist of white-space and/or comma separated values which are a number followed by a suffix indicating duration. For example:

[36] https://docs.python.org/3.9/library/collections.html#collections.Counter
[37] https://docs.python.org/3.9/library/collections.html#collections.Counter
[38] https://docs.python.org/3.9/library/collections.html#collections.Counter.elements
[39] https://docs.python.org/3.9/library/collections.html#collections.Counter
[40] https://docs.python.org/3.9/library/collections.html#collections.Counter
[41] https://docs.python.org/3.9/library/collections.html#collections.Counter.most_common
[42] https://docs.python.org/3.9/library/collections.html#collections.Counter
[43] https://docs.python.org/3.9/library/exceptions.html#ValueError
[44] https://docs.python.org/3.9/library/exceptions.html#ValueError
[45] https://docs.python.org/3.9/library/exceptions.html#ValueError
[46] https://docs.python.org/3.9/library/exceptions.html#ValueError
[47] https://docs.python.org/3.9/library/exceptions.html#ValueError

```
>>> parse_duration('1s')
relativedelta(seconds=+1)
>>> parse_duration('5 minutes, 30 seconds')
relativedelta(minutes=+5, seconds=+30)
>>> parse_duration('1 year')
relativedelta(years=+1)
```

Note that some suffixes like "m" can be ambiguous; using common abbreviations should avoid ambiguity:

```
>>> parse_duration('1 m')
relativedelta(months=+1)
>>> parse_duration('1 min')
relativedelta(minutes=+1)
>>> parse_duration('1 mon')
relativedelta(months=+1)
```

The set of possible durations, and their recognized suffixes is as follows:

- *Microseconds*: microseconds, microsecond, microsec, micros, micro, mseconds, msecond, msecs, msec, ms
- *Seconds*: seconds, second, secs, sec, s
- *Minutes*: minutes, minute, mins, min, mi
- *Hours*: hours, hour, hrs, hr, h
- *Days*: days, day, d
- *Weeks*: weeks, week, wks, wk, w
- *Months*: months, month, mons, mon, mths, mth, m
- *Years*: years, year, yrs, yr, y

If conversion fails, `ValueError`[48] is raised.

`structa.conversions.`**`parse_duration_or_timestamp`**(*s*)
Convert the string *s* to a `datetime`[49] or a `relativedelta`. Duration conversion is attempted to and, if this fails, date-time conversion is attempted. A `ValueError`[50] is raised if both conversions fail.

6.1.5 structa.errors

The *structa.errors* (page 30) module defines all the custom exception and warning classes used in structa.

exception `structa.errors.`**`ValidationWarning`**
Warning raised when a value fails to validate against the computed pattern or schema.

6.1.6 structa.format

The *structa.format* (page 30) module contains various simple routines for "nicely" formatting certain structures for output.

`structa.format.`**`format_chars`**(*chars*, *range_sep='-'*, *list_sep=''*)
Given a set of *chars*, returns a compressed string representation of all values in the set. For example:

```
>>> char_ranges({'a', 'b'})
'ab'
>>> char_ranges({'a', 'b', 'c'})
```

(continues on next page)

[48] https://docs.python.org/3.9/library/exceptions.html#ValueError
[49] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[50] https://docs.python.org/3.9/library/exceptions.html#ValueError

(continued from previous page)

```
'a-c'
>>> char_ranges({'a', 'b', 'c', 'd', 'h'})
'a-dh'
>>> char_ranges({'a', 'b', 'c', 'd', 'h', 'i'})
'a-dh-i'
```

range_sep and *list_sep* can be optionally specified to customize the strings used to separate ranges and lists of ranges respectively.

structa.format.**format_int**(*i*)

Reduce *i* by some appropriate power of 1000 and suffix it with an appropriate Greek qualifier (K for kilo, M for mega, etc). For example:

```
>>> format_int(0)
'0'
>>> format_int(10)
'10'
>>> format_int(1000)
'1.0K'
>>> format_int(1600)
'1.6K'
>>> format_int(2**32)
'4.3G'
```

structa.format.**format_repr**(*self*, ***override*)

Returns a `repr()`[51] style string for *self* in the form `class(name=value, name=value, ...)`.

Note: At present, this function does *not* handle recursive structures unlike `reprlib.recursive_repr()`[52].

structa.format.**format_sample**(*value*)

Format a scalar value for output. The *value* can be a `str`[53], `int`[54], `float`[55], `bool`[56], `datetime`[57], or `None`[58].

The result is a `str`[59] containing a "nicely" formatted representation of the value. For example:

```
>>> format_sample(1.0)
'1'
>>> format_sample(1.5)
'1.5'
>>> format_sample(200000000000)
'200.0G'
>>> format_sample(200000000000.0)
'2e+11'
>>> format_sample(None)
'null'
>>> format_sample(False)
'false'
>>> format_sample('foo')
'"foo"'
>>> format_sample(datetime.now())
'2021-08-16 14:05:04'
```

[51] https://docs.python.org/3.9/library/functions.html#repr
[52] https://docs.python.org/3.9/library/reprlib.html#reprlib.recursive_repr

6.1.7 structa.source

class structa.source.**Source**(*source*, *, *encoding='auto'*, *encoding_strict=True*, *format='auto'*, *csv_delimiter='auto'*, *csv_quotechar='auto'*, *yaml_safe=True*, *json_strict=True*, *sample_limit=1048576*)

A generalized data source capable of automatically recognizing certain popular data formats, and guessing character encodings. Constructed with a mandatory file-like object as the *source*, and a multitude of keyword-only options, the decoded content can be access from *data* (page 32)

The *source* must have a `read()`[60] method which, given a number of bytes to return, returns a `bytes`[61] string up to that length, but has no requirements beyond this. Note that this means files over sockets or pipes are acceptable inputs.

Parameters

- **source** (*file*) – The file-like object to decode (must have a `read` method).
- **encoding** (*str*[62]) – The character encoding used in the source, or "auto" (the default) if it should be guessed from a sample of the data.
- **encoding_strict** (*bool*[63]) – If `True`[64] (the default), raise an exception if character decoding errors occur. Otherwise, replace invalid characters silently.
- **format** (*str*[65]) – If "auto" (the default), guess the format of the data source. Otherwise can be explicitly set to "csv", "yaml", or "json" to force parsing of that format.
- **csv_delimiter** (*str*[66]) – If "auto" (the default), attempt to guess the field delimiter when the "csv" format is being decoded using the `csv.Sniffer`[67] class. Comma, semi-colon, space, and tab characters will be attempted. Otherwise must be set to the single character `str`[68] used as the field delimiter (e.g. ",").
- **csv_quotechar** (*str*[69]) – If "auto" (the default), attempt to guess the string delimiter when the "csv" format is being decoded using the `csv.Sniffer`[70] class. Otherwise must be set to the single character `str`[71] used as the string delimiter (e.g. '"').
- **yaml_safe** (*bool*[72]) – If `True`[73] (the default) the "safe" YAML parser from ruamel.yaml[74] will be used.
- **json_strict** (*bool*[75]) – If `True`[76] (the default), control characters will not be permitted inside decoded strings.
- **sample_limit** (*int*[77]) – The number of bytes to sample from the beginning of the stream when attempting to determine character encoding. Defaults to 1MB.

property csv_dialect

The `csv.Dialect`[78] used when *format* (page 32) is "csv", or `None`[79] otherwise.

property data

The decoded data. Typically a `list`[80] or `dict`[81] of values, but can be any value representable in the source format.

property encoding

The character encoding detected or specified for the source, e.g. "utf-8".

property format

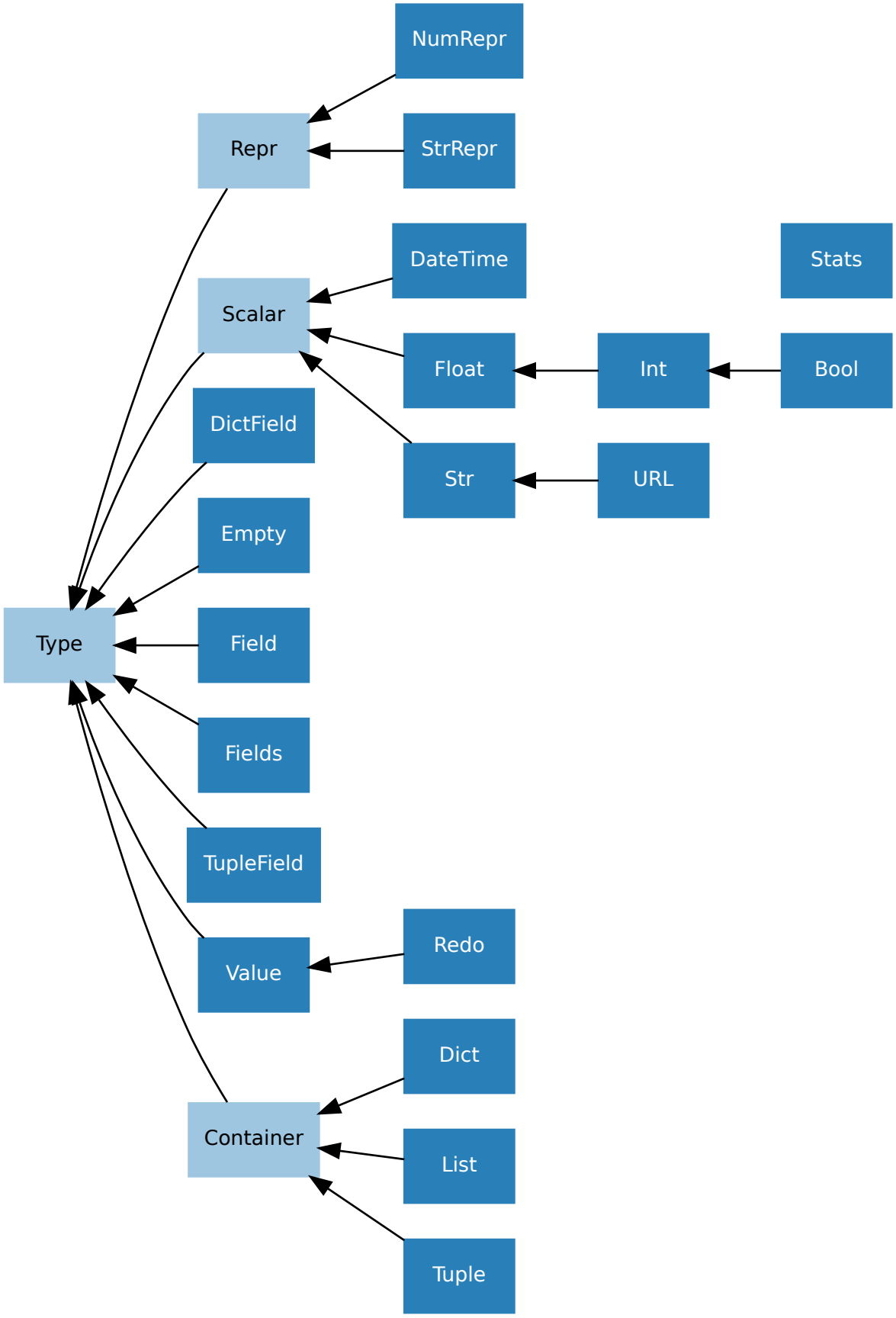
The data format detected or specified for the source, e.g. "csv", "yaml", or "json".

[53] https://docs.python.org/3.9/library/stdtypes.html#str
[54] https://docs.python.org/3.9/library/functions.html#int
[55] https://docs.python.org/3.9/library/functions.html#float
[56] https://docs.python.org/3.9/library/functions.html#bool
[57] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[58] https://docs.python.org/3.9/library/constants.html#None
[59] https://docs.python.org/3.9/library/stdtypes.html#str

6.1.8 structa.types

The *structa.types* (page 33) module defines the class hierarchy used to represent the structural types of analyzed data. The root of the hierarchy is the *Type* (page 34) class. The rest of the hierarchy is illustrated in the chart below:

[60] https://docs.python.org/3.9/library/io.html#io.RawIOBase.read
[61] https://docs.python.org/3.9/library/stdtypes.html#bytes
[62] https://docs.python.org/3.9/library/stdtypes.html#str
[63] https://docs.python.org/3.9/library/functions.html#bool
[64] https://docs.python.org/3.9/library/constants.html#True
[65] https://docs.python.org/3.9/library/stdtypes.html#str
[66] https://docs.python.org/3.9/library/stdtypes.html#str
[67] https://docs.python.org/3.9/library/csv.html#csv.Sniffer
[68] https://docs.python.org/3.9/library/stdtypes.html#str
[69] https://docs.python.org/3.9/library/stdtypes.html#str
[70] https://docs.python.org/3.9/library/csv.html#csv.Sniffer
[71] https://docs.python.org/3.9/library/stdtypes.html#str
[72] https://docs.python.org/3.9/library/functions.html#bool
[73] https://docs.python.org/3.9/library/constants.html#True
[74] https://pypi.org/project/ruamel.yaml/
[75] https://docs.python.org/3.9/library/functions.html#bool
[76] https://docs.python.org/3.9/library/constants.html#True
[77] https://docs.python.org/3.9/library/functions.html#int
[78] https://docs.python.org/3.9/library/csv.html#csv.Dialect
[79] https://docs.python.org/3.9/library/constants.html#None
[80] https://docs.python.org/3.9/library/stdtypes.html#list
[81] https://docs.python.org/3.9/library/stdtypes.html#dict



class structa.types.**Type**

The abstract base class of all types recognized by structa.

This class ensures that instances are hashable (can be used as keys in dictionaries), have a reasonable `repr()`[82] value for ease of use at the REPL[83], can be passed to the *`xml()`* (page 41) function.

However, the most important thing implemented by this base class is the equality test which can be used to test whether a given type is "compatible" with another type. The base test implemented at this level is that one type is compatible with another if one is a sub-class of the other.

Hence, *`Str`* (page 38) is compatible with *`Str`* (page 38) as they are the same class (and hence one is, redundantly, a sub-class of the other). And *`Int`* (page 37) is compatible with *`Float`* (page 36) as it is a sub-class of the latter. However *`Int`* (page 37) is not compatbile with *`Str`* (page 38) as both descend from *`Scalar`* (page 36) and are siblings rather than parent-child.

class structa.types.**Container**(*sample*, *content=None*)

Abstract base of all types that can contain other types. Constructed with a *sample* of values, and an optional definition of *content*.

This is the base class of *`List`* (page 36), *`Tuple`* (page 35), and *`Dict`* (page 35). Note that it is *not* the base class of *`Str`* (page 38) as, although that is a compound type, it cannot *contain other types*; structa treats *`Str`* (page 38) as a scalar type.

`Container` (page 35) extends *`Type`* (page 34) by permitting instances to be added to (compatible, by equality) instances, combining their *`content`* (page 35) appropriately.

content: list[84]**[*Type* (page 34)]**

A list of *`Type`* (page 34) descendents representing the content of this instance.

lengths: *Stats* (page 40)

The *`Stats`* (page 40) of the lengths of the *`sample`* (page 35) values.

sample: [list[85]**] | [tuple**[86]**] | [dict**[87]**]**

The sample of values that this instance represents.

with_content(*content*)

Return a new copy of this container with the *`content`* (page 35) replaced with *content*.

class structa.types.**Dict**(*sample*, *content=None*, *, *similarity_threshold=0.5*)

Represents mappings (or dictionaries).

This concrete refinement of *`Container`* (page 35) uses *`DictField`* (page 36) instances in its *`content`* (page 35) list.

In the case that a mapping is analyzed as a "record" mapping (of fields to values), the *`content`* (page 35) list will contain one or more *`DictField`* (page 36) instances, for which the *`key`* (page 36) attribute(s) will be *`Field`* (page 39) instances.

However, if the mapping is analyzed as a "table" mapping (of keys to records), the *`content`* (page 35) list will contain a single *`DictField`* (page 36) instance mapping the key's type to the value structure.

validate(*value*)

Validate that *value* (which must be a `dict`[88]) matches the analyzed mapping structure.

Raises **TypeError**[89] – if *value* is not a `dict`[90]

class structa.types.**Tuple**(*sample*, *content=None*)

Represents sequences of heterogeneous types (typically tuples).

This concrete refinement of *`Container`* (page 35) uses *`TupleField`* (page 36) instances in its *`content`* (page 35) list.

[82] https://docs.python.org/3.9/library/functions.html#repr
[83] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop
[84] https://docs.python.org/3.9/library/stdtypes.html#list
[85] https://docs.python.org/3.9/library/stdtypes.html#list
[86] https://docs.python.org/3.9/library/stdtypes.html#tuple
[87] https://docs.python.org/3.9/library/stdtypes.html#dict
[88] https://docs.python.org/3.9/library/stdtypes.html#dict
[89] https://docs.python.org/3.9/library/exceptions.html#TypeError
[90] https://docs.python.org/3.9/library/stdtypes.html#dict

Tuples are typically the result of an analysis of some homogeneous outer sequence (usually a *List* (page 36) though sometimes a *Dict* (page 35)) that contains heterogeneous sequences (the *Tuple* (page 35) instance).

validate(*value*)
Validate that *value* (which must be a tuple[91]) matches the analyzed mapping structure.

Raises

- **TypeError**[92] – if *value* is not a tuple[93]
- **ValueError**[94] – if *value* is not within the length limits of the sampled values

class structa.types.**List**(*sample*, *content=None*)
Represents sequences of homogeneous types. This only ever has a single *Type* (page 34) descendent in its *content* (page 35) list.

validate(*value*)
Validate that *value* (which must be a list[95]) matches the analyzed mapping structure.

Raises **TypeError**[96] – if *value* is not a list[97]

class structa.types.**DictField**(*key*, *value=None*)
Represents a single mapping within a *Dict* (page 35), from the *key* (page 36) to its corresponding *value* (page 36). For example, a *Field* (page 39) of a record mapping to some other type, or a generic *Str* (page 38) mapping to an *Int* (page 37) value.

key: *Type* (page 34)
The *Type* (page 34) descendent representing a single key in the mapping. This is usually a *Scalar* (page 36) descendent, or a *Field* (page 39).

value: *Type* (page 34)
The *Type* (page 34) descendent representing a value in the mapping.

class structa.types.**TupleField**(*index*, *value=None*)
Represents a single field within a *Tuple* (page 35), with the *index* (page 36) (an integer number) and its corresponding *value* (page 36).

index: int[98]
The index of the field within the tuple.

value: *Type* (page 34)
The *Type* (page 34) descendent representing a value in the tuple.

class structa.types.**Scalar**(*sample*)
Abstract base of all types that cannot contain other types. Constructed with a *sample* of values.

This is the base class of *Float* (page 36) (from which *Int* (page 37) and then *Bool* (page 37) descend), *Str* (page 38), and *DateTime* (page 37).

values: *Stats* (page 40)
The *Stats* (page 40) of the *sample* (page 36) values.

property sample
A sequence of the sample values that the instance was constructed from (this will not be the original sequence, but one derived from that).

class structa.types.**Float**(*sample*)
Represents scalar floating-point values in datasets. Constructed with a *sample* of values.

[91] https://docs.python.org/3.9/library/stdtypes.html#tuple
[92] https://docs.python.org/3.9/library/exceptions.html#TypeError
[93] https://docs.python.org/3.9/library/stdtypes.html#tuple
[94] https://docs.python.org/3.9/library/exceptions.html#ValueError
[95] https://docs.python.org/3.9/library/stdtypes.html#list
[96] https://docs.python.org/3.9/library/exceptions.html#TypeError
[97] https://docs.python.org/3.9/library/stdtypes.html#list
[98] https://docs.python.org/3.9/library/functions.html#int

classmethod from_strings(*sample*, *pattern*, *bad_threshold=0*)
Class method for constructing an instance wrapped in a *StrRepr* (page 39) to indicate a string representation of a set of floating-point values. Constructed with an *sample* of strings, a *pattern* (which currently must simply be "f"), and a *bad_threshold* of values which are permitted to fail conversion.

validate(*value*)
Validate that *value* (which must be a float[99]) lies within the range of sampled values.

Raises

- **TypeError**[100] – if *value* is not a float[101]
- **ValueError**[102] – if *value* is outside the range of sampled values

class structa.types.**Int**(*sample*)
Represents scalar integer values in datasets. Constructed with a *sample* of values.

classmethod from_strings(*sample*, *pattern*, *bad_threshold=0*)
Class method for constructing an instance wrapped in a *StrRepr* (page 39) to indicate a string representation of a set of integer values. Constructed with an *sample* of strings, a *pattern* (which may be "d", "o", or "x" to represent the base used in the string representation), and a *bad_threshold* of values which are permitted to fail conversion.

validate(*value*)
Validate that *value* (which must be an int[103]) lies within the range of sampled values.

Raises

- **TypeError**[104] – if *value* is not a int[105]
- **ValueError**[106] – if *value* is outside the range of sampled values

class structa.types.**Bool**(*sample*)
Represents scalar boolean values in datasets. Constructed with a *sample* of values.

classmethod from_strings(*iterable*, *pattern*, *bad_threshold=0*)
Class method for constructing an instance wrapped in a *StrRepr* (page 39) to indicate a string representation of a set of booleans. Constructed with an *sample* of strings, a *pattern* (which is a string of the form "falseltrue", i.e. the expected string representations of the False[107] and True[108] values separated by a bar), and a *bad_threshold* of values which are permitted to fail conversion.

validate(*value*)
Validate that *value* is an int[109] (with the value 0 or 1), or a bool[110]. Raises TypeError[111] or ValueError[112] in the event that *value* fails to validate.

Raises

- **TypeError**[113] – if *value* is not a bool[114] or int[115]
- **ValueError**[116] – if *value* is an int[117] that is not 0 or 1

[99] https://docs.python.org/3.9/library/functions.html#float
[100] https://docs.python.org/3.9/library/exceptions.html#TypeError
[101] https://docs.python.org/3.9/library/functions.html#float
[102] https://docs.python.org/3.9/library/exceptions.html#ValueError
[103] https://docs.python.org/3.9/library/functions.html#int
[104] https://docs.python.org/3.9/library/exceptions.html#TypeError
[105] https://docs.python.org/3.9/library/functions.html#int
[106] https://docs.python.org/3.9/library/exceptions.html#ValueError
[107] https://docs.python.org/3.9/library/constants.html#False
[108] https://docs.python.org/3.9/library/constants.html#True
[109] https://docs.python.org/3.9/library/functions.html#int
[110] https://docs.python.org/3.9/library/functions.html#bool
[111] https://docs.python.org/3.9/library/exceptions.html#TypeError
[112] https://docs.python.org/3.9/library/exceptions.html#ValueError
[113] https://docs.python.org/3.9/library/exceptions.html#TypeError
[114] https://docs.python.org/3.9/library/functions.html#bool
[115] https://docs.python.org/3.9/library/functions.html#int
[116] https://docs.python.org/3.9/library/exceptions.html#ValueError
[117] https://docs.python.org/3.9/library/functions.html#int

class structa.types.**DateTime**(*sample*)

Represents scalar timestamps (a date, and a time) in datasets. Constructed with a *sample* of values.

classmethod from_numbers(*pattern*)

Class method for constructing an instance wrapped in a *NumRepr* (page 39) to indicate a numeric representation of a set of timestamps (e.g. day offset from the UNIX epoch).

Constructed with an *sample* of number, a *pattern* (which can be a *StrRepr* (page 39) instance if the numbers are themselves represented as strings, otherwise must be the *Int* (page 37) or *Float* (page 36) instance representing the numbers), and a *bad_threshold* of values which are permitted to fail conversion.

classmethod from_strings(*iterable*, *pattern*, *bad_threshold=0*)

Class method for constructing an instance wrapped in a *StrRepr* (page 39) to indicate a string representation of a set of timestamps.

Constructed with an *sample* of strings, a *pattern* (which must be compatible with datetime.datetime.strptime()[118]), and a *bad_threshold* of values which are permitted to fail conversion.

validate(*value*)

Validate that *value* (which must be a datetime[119]) lies within the range of sampled values.

Raises

- **TypeError**[120] – if *value* is not a datetime.datetime[121]
- **ValueError**[122] – if *value* is outside the range of sampled values

class structa.types.**Str**(*sample*, *pattern=None*)

Represents string values in datasets. Constructed with a *sample* of values, and an optional *pattern* (a sequence of *CharClass* (page 27) instances indicating which characters are valid at which position in fixed-length strings).

lengths: *Stats* (page 40)

The *Stats* (page 40) of the lengths of the sample values.

pattern: [*structa.chars.CharClass* (page 27)]

None[123] if the string is variable length or has no discernable pattern to its values. Otherwise a sequence of *CharClass* (page 27) instances indicating the valid characters at each position of the string.

validate(*value*)

Validate that *value* (which must be a str[124]) lies within the range of sampled values and, if *pattern* (page 38) is not None[125], that it matches the pattern stored there.

Raises

- **TypeError**[126] – if *value* is not a str[127]
- **ValueError**[128] – if *value* is outside the range of sampled values or deviates from the given *pattern* (page 38)

class structa.types.**Repr**(*content*, *pattern=None*)

Abstract base class for representations (string, numeric) of other types. Parent of *StrRepr* (page 39) and *NumRepr* (page 39).

content: *Type* (page 34)

The *Type* (page 34) that this instance is a representation of. For example, a string representation of

[118] https://docs.python.org/3.9/library/datetime.html#datetime.datetime.strptime
[119] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[120] https://docs.python.org/3.9/library/exceptions.html#TypeError
[121] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[122] https://docs.python.org/3.9/library/exceptions.html#ValueError
[123] https://docs.python.org/3.9/library/constants.html#None
[124] https://docs.python.org/3.9/library/stdtypes.html#str
[125] https://docs.python.org/3.9/library/constants.html#None
[126] https://docs.python.org/3.9/library/exceptions.html#TypeError
[127] https://docs.python.org/3.9/library/stdtypes.html#str
[128] https://docs.python.org/3.9/library/exceptions.html#ValueError

integer numbers would be represented by a *StrRepr* (page 39) instance with *content* (page 38) being a *Int* (page 37) instance.

pattern: str[129] **| *Type* (page 34) | None**[130]
Particulars of the representation. For example, in the case of string representations of integers, this is a string indicating the base ("o", "d", "x"). In the case of a numeric representation of a datetime, this is the *Type* (page 34) (*Int* (page 37) or *Float* (page 36)) of the values.

class structa.types.**StrRepr**(*content*, *pattern=None*)
A string representation of an inner type. Typically used to wrap *Int* (page 37), *Float* (page 36), *Bool* (page 37), or *DateTime* (page 37). Descends from *Repr* (page 38).

class structa.types.**NumRepr**(*content*, *pattern=None*)
A numeric representation of an inner type. Typically used to wrap *DateTime* (page 37). Descends from *Repr* (page 38).

class structa.types.**URL**(*sample*, *pattern=None*)
A specialization of *Str* (page 38) for representing URLs. Currently does little more than trivial validation of the scheme.

validate(*value*)
Validate that *value* starts with "http://" or "https://"

Raises **ValueError**[131] – if *value* does not start with a valid scheme

class structa.types.**Field**(*value*, *count*, *optional=False*)
Represents a single key in a *DictField* (page 36) mapping. This is used by the analyzer when it decides a mapping represents a "record" (a mapping of fields to values) rather than a "table" (a mapping of keys to records).

Constructed with the *value* of the key, the *count* of mappings that the key appears in, and a flag indicating if the key is *optional* (defaults to `False`[132] for mandatory).

value: str[133] **| int**[134] **| float**[135] **| tuple**[136] **| ...**
The value of the key.

count: int[137]
The number of mappings that the key belongs to.

optional: bool[138]
If `True`[139], the key may be ommitted from certain mappings in the data. If `False`[140] (the default), the key always appears in the owning mapping.

validate(*value*)
Validates that *value* matches the expected key value.

Raises **ValueError**[141] – if *value* does not match the expected value

class structa.types.**Value**(*sample*)
A descendent of *Type* (page 34) that represents any arbitrary type at all. This is used when the analyzer comes across a container of a multitude of (incompatible) types, e.g. a list of both strings and integers.

It compares equal to all other types, and when added to other types, the result is a new *Value* (page 39) instance.

[129] https://docs.python.org/3.9/library/stdtypes.html#str
[130] https://docs.python.org/3.9/library/constants.html#None
[131] https://docs.python.org/3.9/library/exceptions.html#ValueError
[132] https://docs.python.org/3.9/library/constants.html#False
[133] https://docs.python.org/3.9/library/stdtypes.html#str
[134] https://docs.python.org/3.9/library/functions.html#int
[135] https://docs.python.org/3.9/library/functions.html#float
[136] https://docs.python.org/3.9/library/stdtypes.html#tuple
[137] https://docs.python.org/3.9/library/functions.html#int
[138] https://docs.python.org/3.9/library/functions.html#bool
[139] https://docs.python.org/3.9/library/constants.html#True
[140] https://docs.python.org/3.9/library/constants.html#False
[141] https://docs.python.org/3.9/library/exceptions.html#ValueError

validate(*value*)
Trivial validation; always passes, never raises an exception.

class structa.types.**Empty**
A descendent of *Type* (page 34) that represents a container with no content. For example, if the analyzer comes across a field which always contains an empty list, it would be represented as a *List* (page 36) instance where `List.content` was a sequence containing an *Empty* (page 40) instance.

It compares equal to all other types, and when added to other types, the result is the other type. This allows the merge phase to combine empty lists with a list of integers found at the same level, for example.

validate(*value*)
Trivial validation; always passes.

Note: This counter-intuitive behaviour is because the *Empty* (page 40) value indicates a lack of type-information rather than a definitely empty container (after all, there's usually little sense in having a container field which will always be empty in most hierarchical structures).

The way this differs from *Value* (page 39) is in the additive action.

---

class structa.types.**Stats**(*sample*, *card*, *min*, *q1*, *q2*, *q3*, *max*)
Stores cardinality, minimum, maximum, and (high) median of a *sample* of numeric values (or lengths of strings or containers), along with the specified sample of values.

Typically instances of this class are constructed via the *from_sample()* (page 40) or *from_lengths()* (page 40) class methods rather than directly. However, instances can also be added to other instances to generate statistics for the combined *sample* set. Instances may also be compared for equality.

card: int[142]
The number of items in the *sample* (page 40) that the statistics were calculated from.

q1: int[143] **| float**[144] **| str**[145] **| datetime.datetime**[146] **| ...**
The first (lower) quartile of the *sample* (page 40).

q2: int[147] **| float**[148] **| str**[149] **| datetime.datetime**[150] **| ...**
The second quartile (aka the *median* (page 40)) of the *sample* (page 40).

q3: int[151] **| float**[152] **| str**[153] **| datetime.datetime**[154] **| ...**
The third (upper) quartile of the *sample* (page 40).

max: int[155] **| float**[156] **| str**[157] **| datetime.datetime**[158] **| ...**
The largest value in the *sample* (page 40).

min: int[159] **| float**[160] **| str**[161] **| datetime.datetime**[162] **| ...**
The smallest value in the *sample* (page 40).

sample: *structa.collections.FrozenCounter* (page 29)
The sample data that the statistics were calculated from. This is always an instance of *FrozenCounter* (page 29).

classmethod from_lengths(*sample*)
Given an iterable of *sample* values, which must be of a homogeneous compound type (e.g. `str`[163], `tuple`[164]), construct an instance after calculating the `len()`[165] of each item of the *sample*, and then the minimum, maximum, and quartile values of the lengths.

classmethod from_sample(*sample*)
Given an iterable of *sample* values, which must be of a homogeneous comparable type (e.g. `int`[166], `str`[167], `float`[168]), construct an instance after calculating the minimum, maximum, and quartile values of the *sample*.

property median
An alias for the second quartile, *q2* (page 40).

6.1.9 structa.xml

The *structa.xml* (page 41) module provides methods for generating and manipulating XML, primarily in the form of xml.etree.ElementTree[169] objects. The main class of interest is *ElementFactory* (page 41), which can be used to generate entire element-tree documents in a functional manner.

The *xml()* (page 41) function can be used in a similar manner to str[170] or repr()[171] to generate XML representations of supported objects (most classes within *structa.types* (page 33) support this). Finally, *get_transform()* (page 42) can be used to obtain XSLT trees defined by structa (largely for display purposes).

class structa.xml.**ElementFactory**(*namespace=None*)

A class inspired by Genshi for easy creation of ElementTree Elements.

The ElementFactory class was inspired by the Genshi builder unit in that it permits simple creation of Elements by calling methods on the tag object named after the element you wish to create. Positional arguments become content within the element, and keyword arguments become attributes.

If you need an attribute or element tag that conflicts with a Python keyword, simply append an underscore to the name (which will be automatically stripped off).

Content can be just about anything, including booleans, integers, longs, dates, times, etc. This class simply applies their default string conversion to them (except basestring derived types like string and unicode which are simply used verbatim).

For example:

```
>>> tostring(tag.a('A link'))
'<a>A link</a>'
>>> tostring(tag.a('A link', class_='menuitem'))
'<a class="menuitem">A link</a>'
>>> tostring(tag.p('A ', tag.a('link', class_='menuitem')))
'<p>A <a class="menuitem">link</a></p>'
```

structa.xml.**xml**(*obj*)

In a similar manner to str[172], this function calls the __xml__ method (if any) on *obj*, returning the result which is expected to be an Element[173] instance representing the object.

[142] https://docs.python.org/3.9/library/functions.html#int
[143] https://docs.python.org/3.9/library/functions.html#int
[144] https://docs.python.org/3.9/library/functions.html#float
[145] https://docs.python.org/3.9/library/stdtypes.html#str
[146] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[147] https://docs.python.org/3.9/library/functions.html#int
[148] https://docs.python.org/3.9/library/functions.html#float
[149] https://docs.python.org/3.9/library/stdtypes.html#str
[150] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[151] https://docs.python.org/3.9/library/functions.html#int
[152] https://docs.python.org/3.9/library/functions.html#float
[153] https://docs.python.org/3.9/library/stdtypes.html#str
[154] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[155] https://docs.python.org/3.9/library/functions.html#int
[156] https://docs.python.org/3.9/library/functions.html#float
[157] https://docs.python.org/3.9/library/stdtypes.html#str
[158] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[159] https://docs.python.org/3.9/library/functions.html#int
[160] https://docs.python.org/3.9/library/functions.html#float
[161] https://docs.python.org/3.9/library/stdtypes.html#str
[162] https://docs.python.org/3.9/library/datetime.html#datetime.datetime
[163] https://docs.python.org/3.9/library/stdtypes.html#str
[164] https://docs.python.org/3.9/library/stdtypes.html#tuple
[165] https://docs.python.org/3.9/library/functions.html#len
[166] https://docs.python.org/3.9/library/functions.html#int
[167] https://docs.python.org/3.9/library/stdtypes.html#str
[168] https://docs.python.org/3.9/library/functions.html#float
[169] https://docs.python.org/3.9/library/xml.etree.elementtree.html#module-xml.etree.ElementTree
[170] https://docs.python.org/3.9/library/stdtypes.html#str
[171] https://docs.python.org/3.9/library/functions.html#repr

structa.xml.**get_transform**(*name*)
Return the XSLT transform defined by *name* in the structa.ui module.

structa.xml.**merge_siblings**(*elem*)
Consolidate the content of adjacent sibling child elements with the same tag. For example:

```
>>> x = XML('<doc><a>a</a><a>b</a><a>c</a><b>d</b><a>e</a></doc>')
>>> tostring(merge_siblings(x))
b'<doc><a>abc</a><b>d</b><a>e</a></doc>'
```

Note that the function only deals with *direct* child elements of *elem*; it does nothing to descendents of those children, even if they have the same tag as their parent:

```
>>> x = XML('<doc><a>a<a>b</a></a><a>c</a><b>d</b><a>e</a></doc>')
>>> tostring(merge_siblings(x))
b'<doc><a>a<a>b</a>c</a><b>d</b><a>e</a></doc>'
```

[172] https://docs.python.org/3.9/library/stdtypes.html#str
[173] https://docs.python.org/3.9/library/xml.etree.elementtree.html#xml.etree.ElementTree.Element

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 43) 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[174] 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
```

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

CHAPTER

EIGHT

CHANGELOG

8.1 Release 0.3 (2021-10-27)

- Fixed dictionary merging of scalar and field keys (#19[175])
- Wrote full documentation including tutorials and API reference
- Lots of other minor fixes …

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

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

8.3 Release 0.2 (2021-08-17)

- Better tuple analysis (#4[176]) which was a pre-requisite for…
- Added CSV support (#5[177])
- Added some pretty progress output (#6[178])
- Prettier output (#8[179])
- Added documentation (#9[180])
- Added YAML support (#10[181])
- Better elimination of common sub-trees (#12[182])
- Multi-file input support (#15[183])

[175] https://github.com/waveform80/structa/issues/19
[176] https://github.com/waveform80/structa/issues/4
[177] https://github.com/waveform80/structa/issues/5
[178] https://github.com/waveform80/structa/issues/6
[179] https://github.com/waveform80/structa/issues/8
[180] https://github.com/waveform80/structa/issues/9
[181] https://github.com/waveform80/structa/issues/10
[182] https://github.com/waveform80/structa/issues/12
[183] https://github.com/waveform80/structa/issues/15

8.4 Release 0.1 (2018-12-17)

- Initial commit of something that works … ish

CHAPTER

NINE

LICENSE

This file is part of structa.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA or see <https://www.gnu.org/licenses/>.

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

S

T

U

V

W

X