TEST-DRIVEN DATA ANALYSIS

PyData London 2018 • Tutorial • 27th April 2018

http://www.tdda.info/pdf/tdda-tutorial-pydata-london-2018.pdf

Nicholas J. Radcliffe
Stochastic Solutions Limited
& Department of Mathematics, University of Edinburgh

TDD ↦ TDDA

We need to extend TDD's idea of testing for

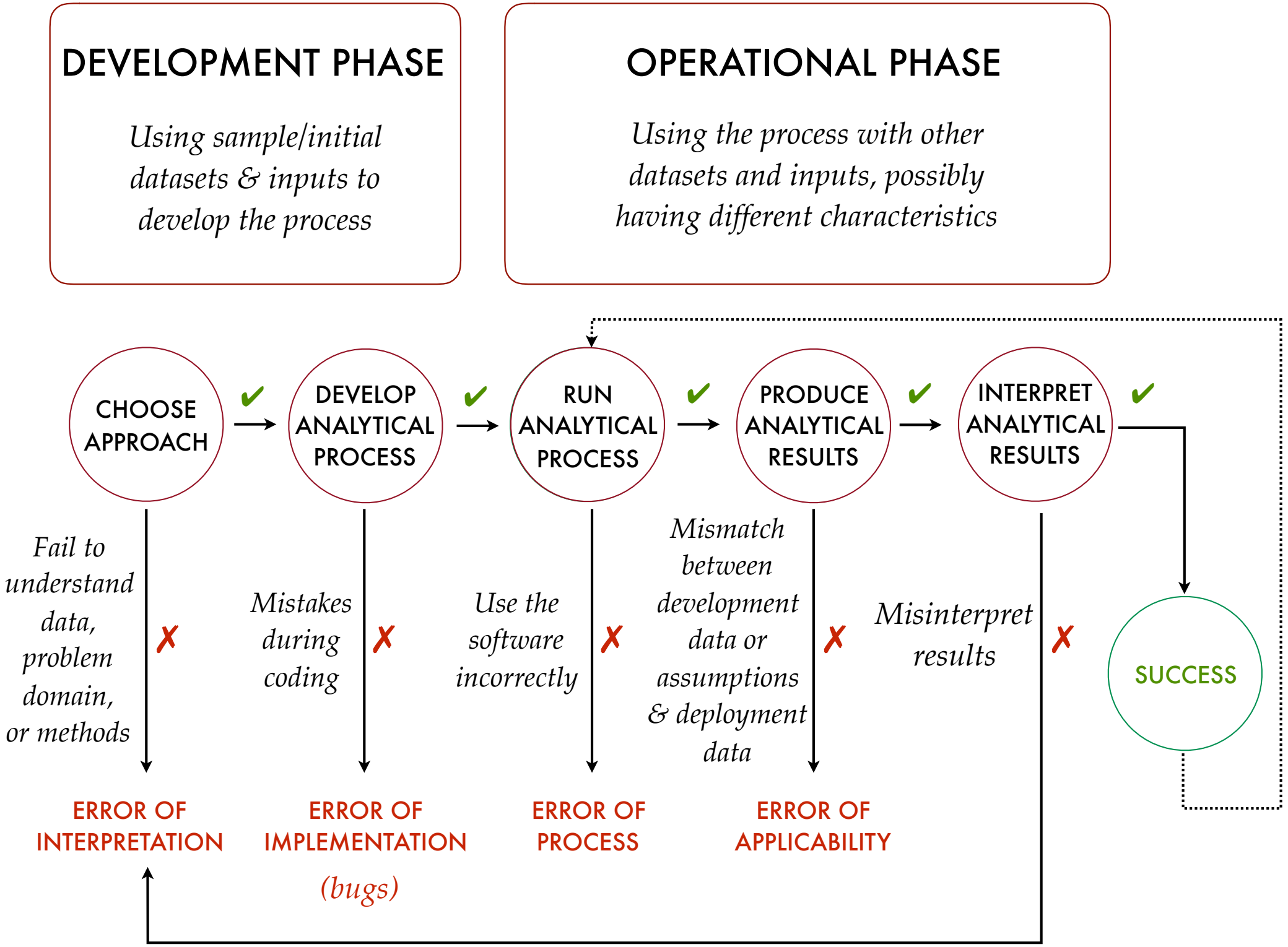
software correctness

with the idea of testing for

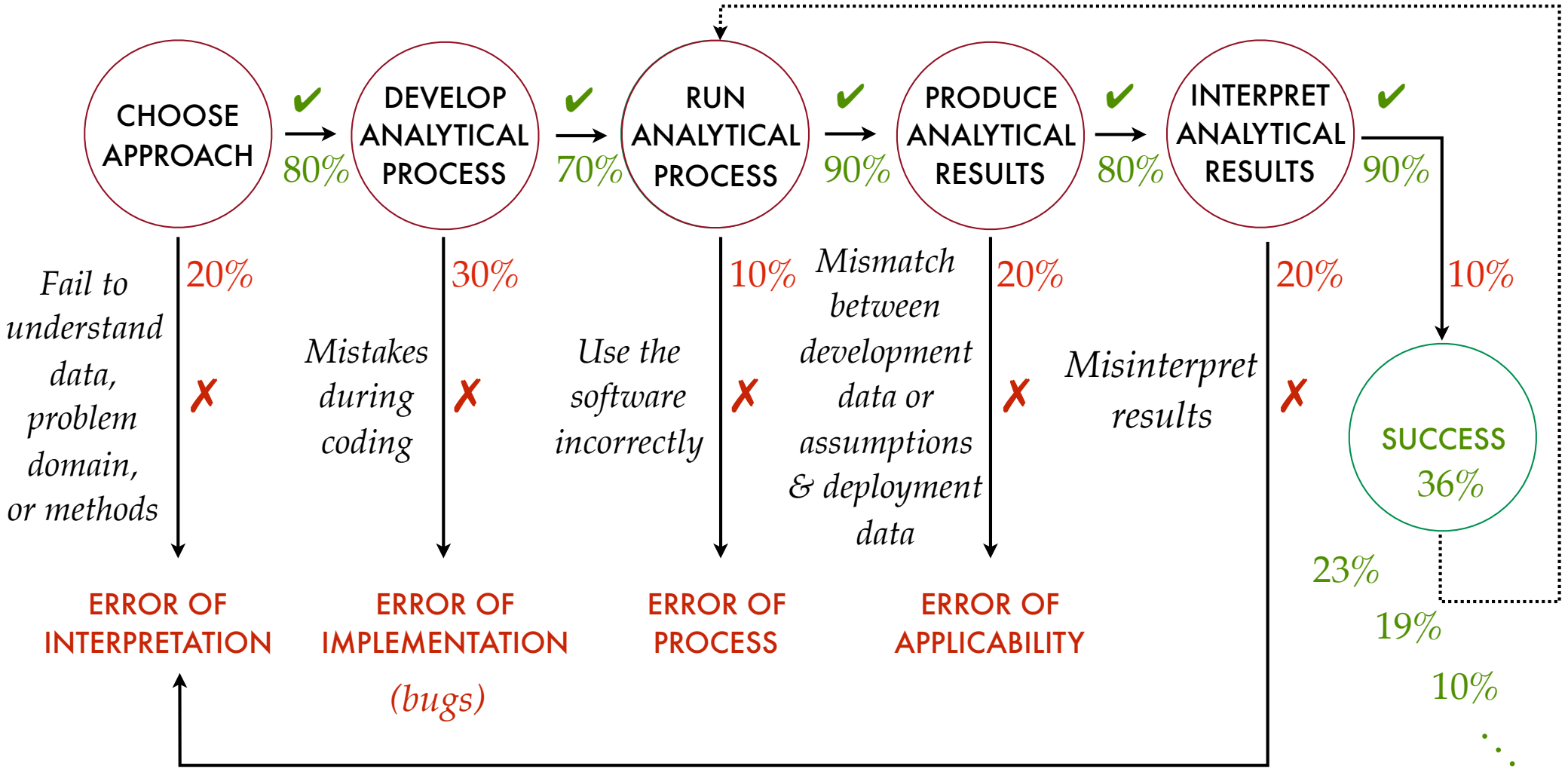
meaningfulness of analysis,

correctness and validity of input data,

& *correctness of interpretation*.


DEVELOPMENT PHASE
Using sample/initial datasets & inputs to develop the process
OPERATIONAL PHASE
Using the process with other datasets and inputs, possibly having different characteristics
CHOOSE APPROACH
DEVELOP ANALYTICAL PROCESS
RUN ANALYTICAL PROCESS
PRODUCE ANALYTICAL RESULTS
INTERPRET ANALYTICAL RESULTS
SUCCESS
Fail to understand data, problem domain, or methods
Mistakes during coding
Use the software incorrectly
Mismatch between development data or assumptions & deployment data
Misinterpret results
ERROR OF INTERPRETATION
ERROR OF IMPLEMENTATION
(bugs)
ERROR OF PROCESS
ERROR OF APPLICABILITY

If you buy into this model, it's sobering to attach probability estimates to each transition and calculate the probability of success after a few runs . . .



TDDA: MAIN IDEAS

1. "Reference" Tests
    - *cf.* system/integration tests in TDD
    - With support for exclusions, regeneration, helpful reporting etc.
    - Re-run these tests *all the time*, *everywhere*
2. Constraint Discovery & Verification
    - a bit like unit tests for data
    - can cover inputs, outputs and intermediate results
    - automatically discovered
        - *more-or-less* including regular expressions for characterising strings (Rexpy)
    - Use as part of analysis to verify inputs, outputs and intermediates (as appropriate)

TDDA LIBRARY

1. From PyPI (recommended)

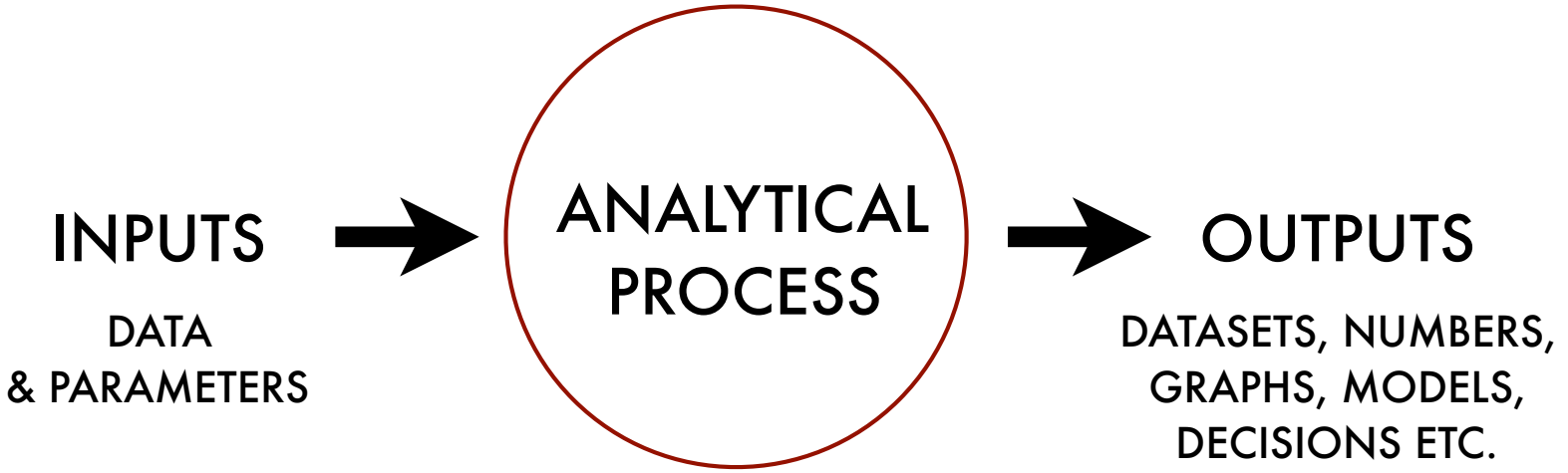
   ```
   pip install tdda
   ```

2. From Github (source)

   ```
   git clone https://github.com/tdda/tdda.git
   ```

- Runs on Python 2 & Python 3, Mac, Linux & Windows, under `unittest` and `pytest`
- MIT Licensed
- Documentation:
  - Sphinx source in `doc` subdirectory
  - Built copy at `http://tdda.readthedocs.io`
- Quick reference: `http://www.tdda.info/pdf/tdda-quickref.pdf`

REFERENCE TESTS



Develop a verification procedure (`diff`) *and periodically rerun:*
do the same inputs (still) produce the same outputs?

REFERENCE TEST SUPPORT

1: UNSTRUCTURED (STRING) RESULTS

- Comparing actual string (in memory or in file) to reference (*expected*) string (in file)
- Exclude lines with substrings or regular expressions
- Preprocess output before comparison
- Write actual string produced to file when different
- Show specific diff command needed to examine differences
- Check multiple files in single test; report all failures
- Automatically re-write reference results after human verification.

REFERENCE TEST SUPPORT

UNSTRUCTURED (STRING) METHODS

Check a single (in-memory) string against a refererence file

```
self.assertStringCorrect(string, ref_path, . . .)
```

Check a single generated file against a reference file:

```
self.assertFileCorrect(actual_path, ref_path, . . .)
```

Check a multiple generated files against respective reference files:

```
self.assertFilesCorrect(actual_paths, ref_paths, . . .)
```

EXERCISE 1: STRING DATA REFERENCE TESTS

I. CHECK THE TESTS PASS

1. Copy examples somewhere:
   ```
   cd ~/tmp
   tdda examples
   cd referencetest-examples
   ```
2. Look at reference output:
   ```
   reference/string_result.html
   reference/file_result.html
   ```
3. Run tests (should **pass**).
   ```
   cd unittest; python test_using_referencetestcase.py;  cd ..
   ```
   or `cd pytest; pytest; cd ..`

NOTE

- Although tests pass, output is *not* identical

  — version number and copyright lines in reference files are different

```
self.assertFileCorrect(outpath, 'file_result.html',
                       ignore_patterns=['Copyright', 'Version'])
```

(This will be clearer after next part of exercise.)

EXERCISE 1 (CTD): STRING DATA REFERENCE TESTS

II. MODIFY THE GENERATOR, VERIFY RESULTS, RE-WRITE REFERENCE RESULTS

4. Modify `generators.py`

   e.g. Capitalise `<h1> ... </h1>` contents in the `generate_string` function

5. Repeat step 3 to run tests again. Two tests should **fail**.

   ```
   cd unittest; python test_using_referencetestcase.py;  cd ..
   ```
   or `cd pytest; pytest; cd ..`

6. Check modified results in (reported) temporary directory are as expected; run the suggested diff command or something similar (`opendiff`, `fc`, …). Again, note that in addition to the changes you introduced, the Copyright and Version lines are different

7. On the assumption that these now represent the verified,* new target results, re-write the reference output with

   ```
   cd unittest; python test_using_referencetestcase.py -W
   ```
   or `cd pytest; pytest --write-all -s`

8. Repeat step 5 to run tests again. All tests should **pass**.

*** WARNING**

If you habitually re-write results when tests fail without carefully verifying the new results, your tests will quickly become worthless.

With great power comes great responsibility: use TDDA Reference Tests wisely!

EXERCISE 1 (CTD): STRING DATA REFERENCE TESTS

III. MODIFY THE RESULTS VERSION NUMBER; CHECK STILL OK

9. Modify `generators.py` code to change version number in output.
10. Repeat step 3 to run tests again. All tests should still **pass** since version number is excluded by `ignore_substrings=['Copyright', 'Version']` parameter to `assertStringCorrect`.

REFERENCE TEST SUPPORT

2: STRUCTURED DATA METHODS (DATAFRAMES & CSV)

- Comparing generated DataFrame or CSV file to reference DataFrame or CSV file
- Show specific diff command needed to examine differences
- Check multiple CSV files in single test; report all failures
- Choose subset of columns (with list or function) to compare
- Choose whether to check (detailed) types
- Choose whether to check column order
- Choose whether to ignore actual data in particular columns
- Choose precision for floating-point comparisons
- Automatic re-writing of verified (changed) results.

REFERENCE TEST SUPPORT

STRUCTURED DATA METHODS (DATAFRAMES & CSV)

Check a single generated CSV file against a refererence CSV file

```
self.assertCSVFileCorrect(actual_path, ref_csv, . . .)
```

Check multiple generated files against respective reference CSV files:

```
self.assertCSVFilesCorrect(actual_paths, ref_csvs, . . .)
```

Check an (in-memory) DataFrame against a reference CSV file

```
self.assertDataFrameCorrect(df, ref_csv, . . .)
```

Check an (in-memory) DataFrame against another (in-memory) DataFrame

```
self.assertDataFramesEqual(df, ref_df, . . .)
```

EXERCISE 2: DATAFRAME/CSV REFERENCE TESTS

I. CHECK THE TESTS PASS

1. If you've done Exercise 1, you already have the examples in the same directory
2. Look at reference output:
   ```
   reference/dataframe_result.csv
   reference/dataframe_result2.csv
   ```
3. Run tests (should **pass**).
   ```
   cd unittest; python test_using_referencetestcase.py;  cd ..
   cd pytest; pytest; cd ..
   ```

NOTE

You can look at the data frame being generated with this 2-line program (save as `show.py`)

```
from dataframes import generate_dataframe

print(generate_dataframe())
```

EXERCISE 2: DATAFRAME/CSV REFERENCE TESTS

II. MODIFY THE DATA GENERATOR, VERIFY RESULTS, RE-WRITE REFERENCE RESULTS

4. Modify `dataframes.py`

   e.g. Change the default precision from `3` to `2` in the `generate_dataframe` function. This will cause the string column `s` to be different.

5. Repeat step 3 to run tests again. Three tests should **fail**.

   ```
   cd unittest; python test_using_referencetestcase.py;  cd ..
   ```
   or `cd pytest; pytest; cd ..`

6. Look at the way differences are reported, and check that the only material change is to column `s`, as expected.

7. On the assumption that this new output now represents the new, verified target result,* re-write the reference output with

   ```
   cd unittest; python test_using_referencetestcase.py -W
   ```
   or `cd pytest; pytest --write-all -s`

8. Repeat step 5 to run tests again. All tests should now **pass**.

*** WARNING**

If you habitually re-write results when tests fail without carefully verifying the new results, your tests will quickly become worthless.

With great power comes great responsibility: use TDDA Reference Tests wisely!

NEW COOLNESS!

TAGGING TESTS TO RUN A SUBSET

With **`unittest`** (not yet **`pytest`**), you can "tag" single individual tests or whole test classes to allow only those ones to be run with when running with `-1` or `--tagged`.

```
from tdda.referencetest import ReferenceTestCase, tag
class TestDemo(ReferenceTestCase):
    def testOne(self):
        self.assertEqual(1, 1)

    @tag
    def testTwo(self):
        self.assertEqual(2, 2)

    @tag
    def testThree(self):
        self.assertEqual(3, 3)

    def testFour(self):
        self.assertEqual(4, 4)

if __name__ == '__main__':
     ReferenceTestCase.main()
```

```
$ python3 tests.py -1
..
------------------------------
Ran 2 tests in 0.000s

OK
```

See what classes have tagged tests with `-0` or `--istagged`

```
$ python3 tests.py -0
__main__.TestDemo

------------------------------
OK
```

CONSTRAINT GENERATION & VERIFICATION

CONSTRAINTS

- Very commonly, data analysis uses data tables (e.g. DataFrames) as inputs, outputs and intermediate results
- There are many things we know (or at least expect) to be true about these data tables
- *Could* write down all these expectations as constraints and check that they are actually satisfied during analysis . . . *but life's too short!* (Also: humans are rather error-prone)

THE BIG IDEA

- Get the computer to discover constraints satisfied by example datasets automatically.
- Verify against these constraints, modifying as required
- (Humans much happier to make tweaks than start from scratch)

EXAMPLE CONSTRAINTS

SINGLE FIELD CONSTRAINTS	DATASET CONSTRAINTS
Age ≤ 150	The dataset must contain field **CID**
type(Age) = int	Number of records must be 118
CID ≠ NULL	One field should be tagged O
CID unique	**Date** should be sorted ascending
len(CardNumber) = 16	**MULTI-FIELD CONSTRAINTS**
Base in {"C", "G", "A", T"}	StartDate ≤ EndDate
Vote ≠ "Trump"	AlmostEqual(F, m * a, 6)
StartDate < tomorrow()	sum(Favourite*) = 1
v < 2.97e10	minVal ≤ medianVal ≤ maxVal
Height ~ N(1.8, 0.2)	V ≤ H * w * d

CONSTRAINTS SUPPORTED BY TDDA LIBRARY

KIND	DESCRIPTION	BOOLEAN	NUMERIC	DATE	STRING
`min`	*Minimum allowed value; on verification interpreted with proportionate tolerance* `epsilon`.	✔	✔	✔	✗
`max`	*Maximum allowed value; on verification interpreted with proportionate tolerance* `epsilon`.	✔	✔	✔	✗
`sign`	`"positive"`, `"non-negative"`, `"zero"`, `"non-positive"` *or* `"negative"`.	✔	✔	✗	✗
`max_nulls`	`0` *if nulls not allowed. In principle, can be higher values (in particular,* `1`*), but* `discover` *function does not use these at present.*	✔	✔	✔	✔
`no_duplicates`	`true` *if duplicates are not allowed.*	✔	✔	✔	✔
`min_length`	*smallest allowed string length*	✗	✗	✗	✔
`max_length`	*largest allowed string length*	✗	✗	✗	✔
`rex`	*list of regular expressions; strings must match at least one. (Available from version 0.4.0 on.)*	✗	✗	✗	✔

CONSTRAINT GENERATION & VERIFICATION

1. Copy examples somewhere:
   ```
   cd ~/tmp
   tdda examples
   cd constraints-examples
   ```
   (if not done already)
2. Generate constraints from first 92 elements of periodic table (`testdata/elements92.csv`)
   ```
   cd constraints-examples
   python elements_discover_92.py
   ```
   or `tdda discover testdata/elements92.csv elements92.tdda`
3. Examine output (`elements92.tdda`)
4. Perform verification of same data (as DataFrame). Should **pass**.
   ```
   python elements_verify_92.py
   ```
   or `tdda verify testdata/elements92.csv elements92.tdda`

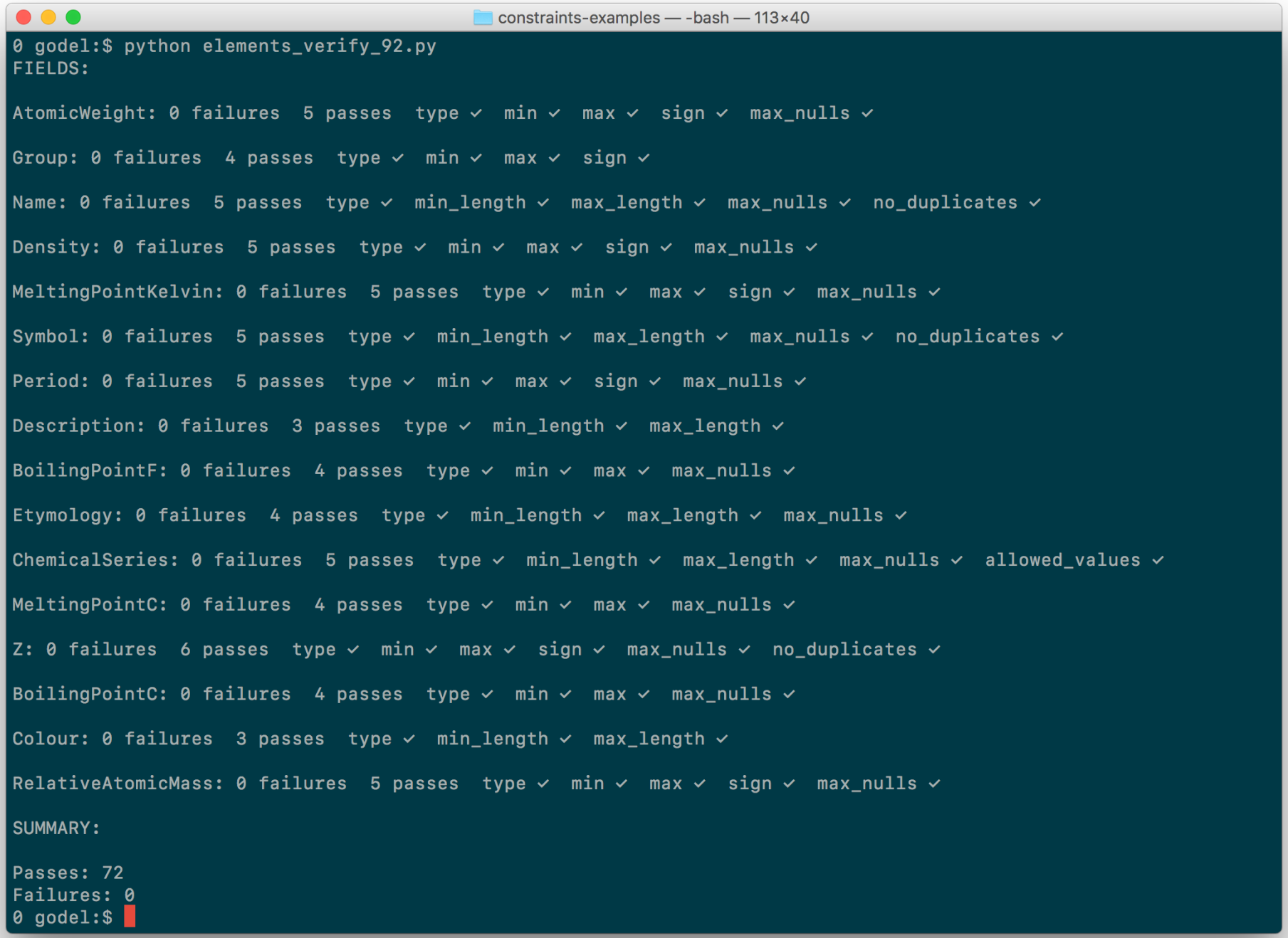
Obviously, verifying a dataset against the constraints generated from that dataset should always work!

```
pip install feather-format
```

EXAMPLE: elements92.tdda


```
{
   "fields": {
      "Z": {"type": "int", "min": 1, "max": 92, "sign": "positive", "max_nulls": 0, "no_duplicates": true},
      "Name": {"type": "string", "min_length": 3, "max_length": 12, "max_nulls": 0, "no_duplicates": true},
      "Symbol": {"type": "string", "min_length": 1,"max_length": 2, "max_nulls": 0, "no_duplicates": true},
      "Period": {"type": "int", "min": 1, "max": 7, "sign": "positive", "max_nulls": 0},
      "Group": {"type": "int", "min": 1, "max": 18,"sign": "positive"},
      "ChemicalSeries": {"type": "string", "min_length": 7, "max_length": 20, "max_nulls": 0,
                         "allowed_values": ["Actinoid", "Alkali metal", "Alkaline earth metal",
                                            "Halogen", "Lanthanoid", "Metalloid", "Noble gas",
                                            "Nonmetal", "Poor metal", "Transition metal"]},
      "AtomicWeight": {"type": "real", "min": 1.007946, "max": 238.028914, "sign": "positive", "max_nulls": 0},
      "Etymology": {"type": "string", "min_length": 4, "max_length": 39, "max_nulls": 0},
      "RelativeAtomicMass": {"type": "real", "min": 1.007946, "max": 238.028914, "sign": "positive",
                             "max_nulls": 0},
      "MeltingPointC": {"type": "real", "min": -258.975, "max": 3675.0, "max_nulls": 1},
      "MeltingPointKelvin": {"type": "real", "min": 14.2, "max": 3948.0, "sign": "positive", "max_nulls": 1},
      "BoilingPointC": {"type": "real", "min": -268.93, "max": 5596.0, "max_nulls": 0},
      "BoilingPointF": {"type": "real", "min": -452.07, "max": 10105.0, "max_nulls": 0},
      "Density": {"type": "real", "min": 8.9e-05, "max": 22.610001, "sign": "positive", "max_nulls": 0},
      "Description": {"type": "string", "min_length": 1, "max_length": 83},
      "Colour": {"type": "string", "min_length": 4, "max_length": 80}
   }
}
```

EXAMPLE SUCCESSFUL VERIFICATION



CONSTRAINT GENERATION & VERIFICATION

5. Now run verification of larger dataset (first 118 elements of periodic table) against the same constraints. Should **fail** (because, for example, atomic number now goes to 118).

   ```
   python elements_verify_118_against_92.py
   ```
   or `tdda verify testdata/elements118.csv elements92.tdda`

6. Repeat verification of larger dataset (118 elements) against constraints generated against that same (118) data. Should **pass**.

   ```
   python elements_verify_118.py
   ```
   or `tdda verify testdata/elements118.csv elements118.tdda`

7. Finally, verify the constraints from 118 data against the 92 data. Should **pass**.

   ```
   tdda verify testdata/elements92.csv elements118.tdda
   ```

Note: fewer constraints are discovered for elements118 than for elements92 (67 against 72). This is because there are nulls in some fields in the 118 data (the melting points, density etc.) but not in the 92 data.

FAILURE (ANOMALY) DETECTION

This afternoon's workshop has much more on this!

8. "Detect" the failing records

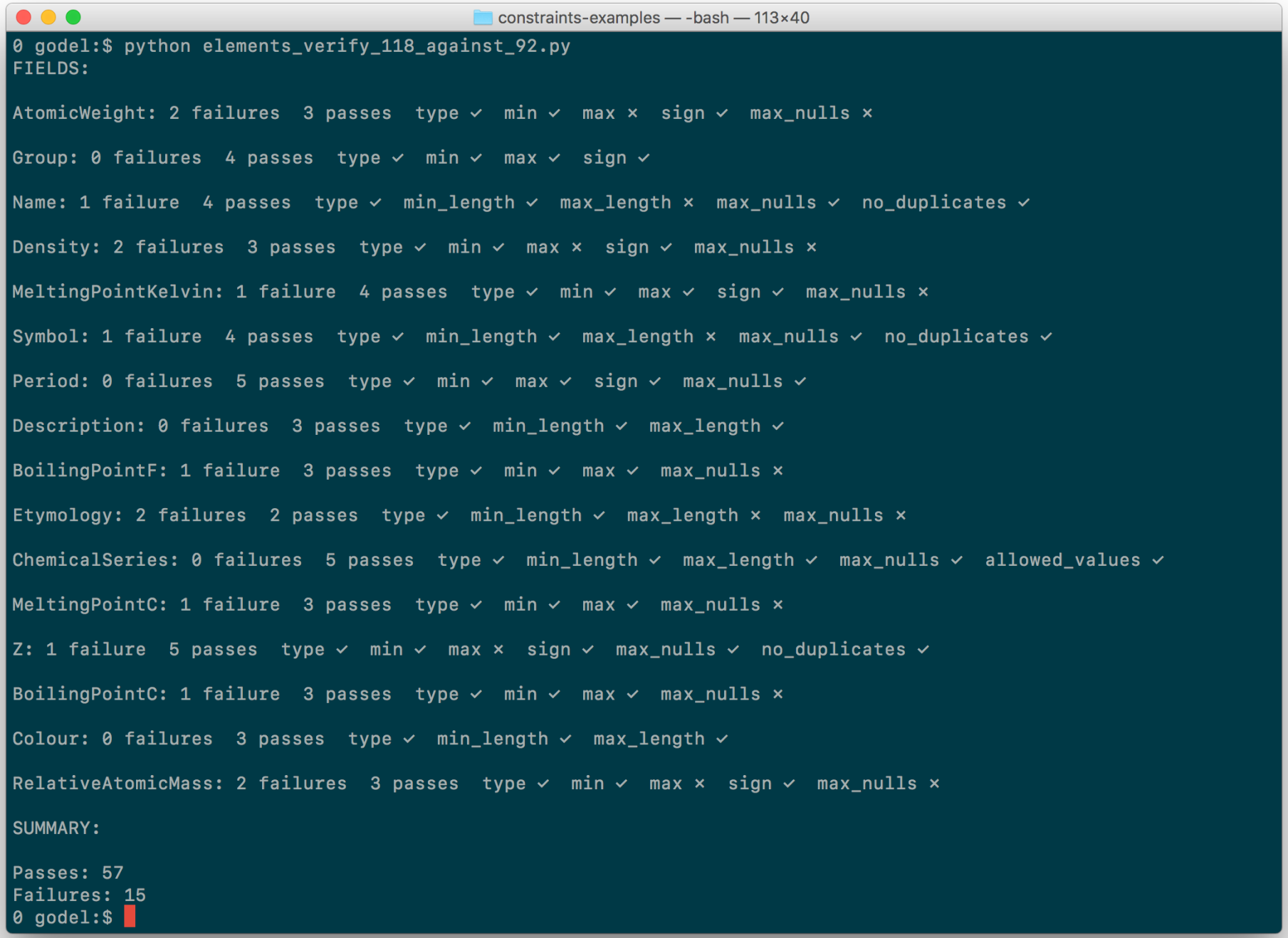
```
tdda detect testdata/elements118.csv \
            elements92.tdda          \
            bads.csv                 \
            --per-constraint         \
            --output-fields
```

- This writes out the failing records to `bads.csv`. (Can use `.feather` instead)

- The `--per-constraint` flag says write out a column for every constraint that ever fails, as well as an `nfailures` column.

- `--output-fields` says write all the original fields as well as the results fields (otherwise, it just writes an index/row number).

EXAMPLE UNSUCCESSFUL VERIFICATION



ABSENT CONSTRAINTS

Gregory (Scotland Yard detective): *"Is there any other point to which you would wish to draw my attention?"*
Holmes: *"To the curious incident of the dog in the night-time."*
Gregory: *"The dog did nothing in the night-time."*
Holmes: *"That was the curious incident."*

— *Silver Blaze*, in *Memoirs of Sherlock Holmes*
Arthur Conan Doyle, 1892.

CONSTRAINTS API

DISCOVERY

```
from tdda.constraints.pdconstraints import discover_constraints

constraints = discover_constraints(df)

with open('constraints.tdda', 'w') as f:
    f.write(constraints.to_json())
```

VERIFICATION

```
from tdda.constraints.pdconstraints import verify_df

verification = verify_df(df, 'constraints.tdda') # (printable object)

constraints_df = verification.to_frame())        # (Pandas DataFrame)
```

OUTPUT of to_frame()

	field	failures	passes	type	min	min_length	max	\
0	AtomicWeight	2	3	True	True	NaN	False	
2	Name	1	4	True	NaN	True	NaN	
3	Density	2	3	True	True	NaN	False	
4	MeltingPointKelvin	1	4	True	True	NaN	True	
5	Symbol	1	4	True	NaN	True	NaN	
7	BoilingPointF	1	3	True	True	NaN	True	
8	Etymology	2	2	True	NaN	True	NaN	
9	RelativeAtomicMass	2	3	True	True	NaN	False	
11	MeltingPointC	1	3	True	True	NaN	True	
12	Z	1	5	True	True	NaN	False	
13	BoilingPointC	1	3	True	True	NaN	True	

	max_length	sign	max_nulls	no_duplicates	allowed_values
0	NaN	True	False	NaN	NaN
2	False	NaN	True	True	NaN
3	NaN	True	False	NaN	NaN
4	NaN	True	False	NaN	NaN
5	False	NaN	True	True	NaN
7	NaN	NaN	False	NaN	NaN
8	False	NaN	False	NaN	NaN
9	NaN	True	False	NaN	NaN
11	NaN	NaN	False	NaN	NaN
12	NaN	True	True	True	NaN
13	NaN	NaN	False	NaN	NaN

CONSTRAINTS

True *Satisfied*

FALSE *Not satisfied*

NaN *No constraint*

TDDA FUTURES

1. Rexpy: Automatic discovery of regular expressions for characterising string fields.
    - Now in of TDDA library
    - Use `discover --rex` to generate Regular Expression constraints
2. Discover & verify working in RDBMS (postgres; mysql; SQLServer; ORACLE)
3. Join key discovery and verification between datasets (working in proprietary version; might come over)
4. Characterizing distributions
5. "Nearly" constraint discovery (working in proprietary version; might come over)
6. Possibly JSON etc. (cf. JSON Schemas)
7. Lots of other ideas.

njr@StochasticSolutions.com

http://tdda.info

https://github.com/tdda

#tdda*

@tdda0 @njr0

* *tweet (DM) us email address for invitation Or email me.*

Correct interpretation: Zero

Error of interpretation: Letter "Oh"

www.tdda.info/pdf/tdda-tutorial-pydata-london-2017.pdf

Rexpy

Regular Expressions by Example

Source: git clone http://github.com/tdda

Package: pip install tdda

Try online: http://rexpy.herokuapp.com

Regular Expressions

212-988-0321

476 123 8829

1 701 734 9288

(617) 222 0529

optional 1

```
^1?[ \(]?\d{3}\)?[ \-]\d{3}[ \-]\d{4}$
```

start of line — *optional space or open bracket* — *digits (3)* — *optional close bracket* — *space or hyphen* — *digits (3)* — *space or hyphen* — *digits (3)* — *end of line*

Regular Expressions

212-988-0321

```
^\d{3}\-\d{3}\-\d{4}$
```

as before

```
^\d+\-\d+\-\d+$
```

less specific:
+ = 1 or more times

```
^[1-2]+\-[8-9]+\-[0-3]+$
```

specific range of digits

```
^212\-988\-0321$
```

totally specific

```
^.*$
```

matches anything
. = any char
** = 0 or more times*

Regular Expressions

MN 55402

OH 45202

^[A-Z]{2} \d{5}$

Regular Expressions

MN 55402

OH 45202-7735

^[A–Z]{2} \d{5}(\–\d{4})?$

unescaped parentheses (no backslash) "tag" sub-expressions

optional

You have a problem.

You think

“I know, I’ll use regular expressions.”

Now you have two problems

Pros & Cons

Powerful

*Ugly

Hard to write

Harder to read

Harder still to debug

Hard to quote/escape†

---

*Butt . . .

† `r'...'` is your friend

Verbal Expressions

```
verbal_expression = VerEx()
tester = (verbal_expression.
                start_of_line().
                find('http').
                maybe('s').
                find('://').
                maybe('www.').
                anything_but(' ').
                end_of_line()
)
```

Why not let
the computer do
the work?

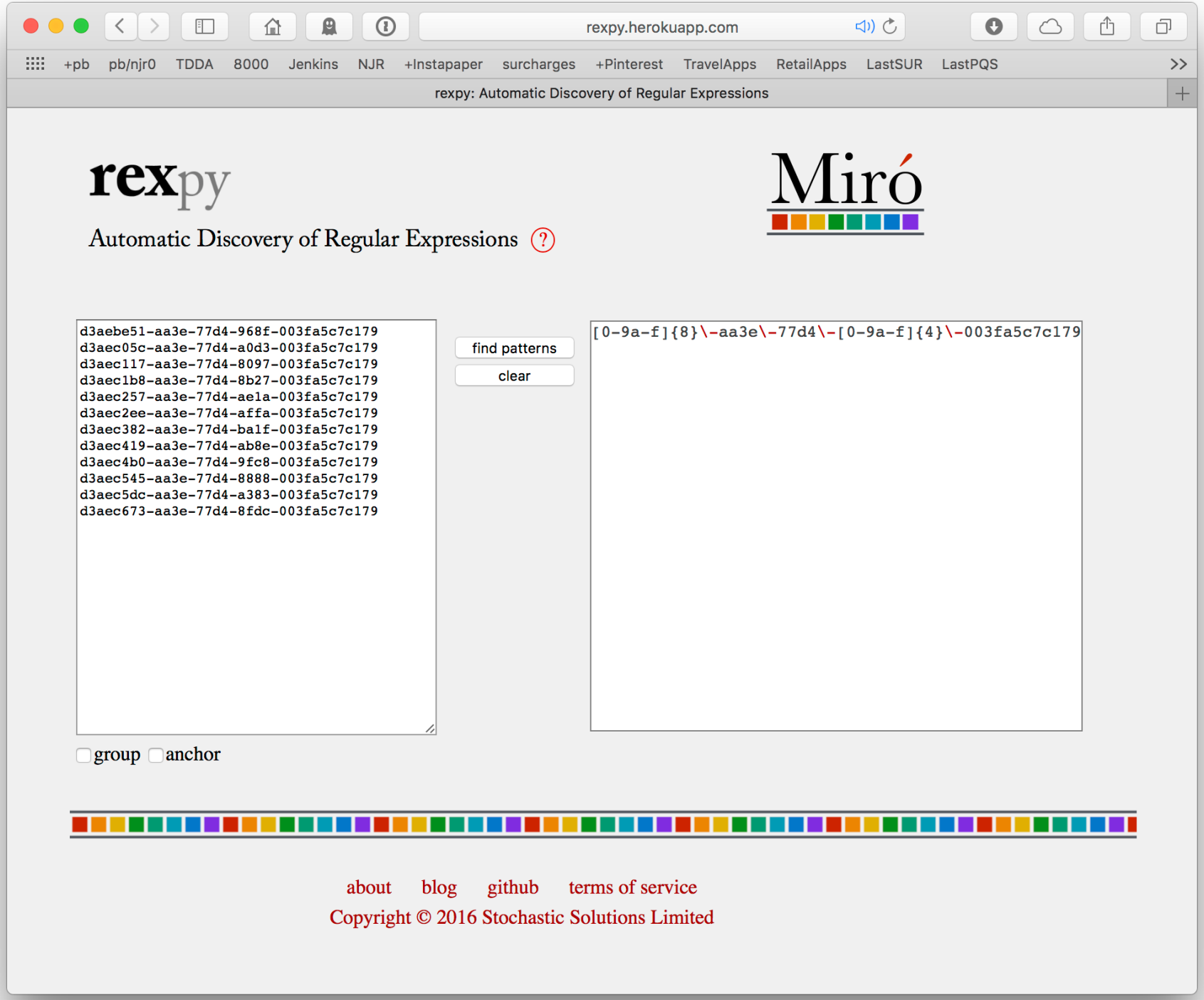
Rexpy

is our early attempt to let the computer
find useful regular expressions
from examples

```
$ python

>>> tels = ['212-988-0321', '987-654-3210', '476 123 8829', '123 456 7890',
>>>         '701 734 9288', '177 441 7712', '617 222 0529', '222 111 9276']
>>> regexps = rexpy.extract(tels)
>>> for r in regexps:
...     print r
^\d{3}\-\d{3}\-\d{4}$
^\d{3}\ \d{3}\ \d{4}$
```

Rexpy currently never groups white space with punctuation; but it will soon.


rexpy.herokuapp.com
+pb pb/njr0 TDDA 8000 Jenkins NJR +Instapaper surcharges +Pinterest TravelApps RetailApps LastSUR LastPQS
rexpy: Automatic Discovery of Regular Expressions
rexpy
Automatic Discovery of Regular Expressions
Miró
d3aebe51-aa3e-77d4-968f-003fa5c7c179
d3aec05c-aa3e-77d4-a0d3-003fa5c7c179
d3aec117-aa3e-77d4-8097-003fa5c7c179
d3aec1b8-aa3e-77d4-8b27-003fa5c7c179
d3aec257-aa3e-77d4-ae1a-003fa5c7c179
d3aec2ee-aa3e-77d4-affa-003fa5c7c179
d3aec382-aa3e-77d4-ba1f-003fa5c7c179
d3aec419-aa3e-77d4-ab8e-003fa5c7c179
d3aec4b0-aa3e-77d4-9fc8-003fa5c7c179
d3aec545-aa3e-77d4-8888-003fa5c7c179
d3aec5dc-aa3e-77d4-a383-003fa5c7c179
d3aec673-aa3e-77d4-8fdc-003fa5c7c179
find patterns
clear
[0-9a-f]{8}\-aa3e\-77d4\-[0-9a-f]{4}\-003fa5c7c179
group
anchor
about
blog
github
terms of service
Copyright © 2016 Stochastic Solutions Limited

Command Line

```
$ rexpy --help

Usage:

    rexpy [FLAGS] [input file [output file]]

or

    python -m tdda.rexpy.rexpy [FLAGS] [input file [output file]]

If input file is provided, it should contain one string per line; otherwise lines will be read from
standard input.

If output file is provided, regular expressions found will be written to that (one per line);
otherwise they will be printed.

FLAGS are optional flags. Currently::

  -h, --header      Discard first line, as a header.

  -?, --help        Print this usage information and exit (without error)

  -g, --group       Generate capture groups for each variable fragment of each regular expression
                    generated, i.e. surround variable components with parentheses

                        e.g.     '^([A-Z]+)\-([0-9]+)$'
                        becomes  '^[A-Z]+\-[0-9]+$'

  -u, --underscore  Allow underscore to be treated as a letter. Mostly useful for matching
                    identifiers. Also allow -_.

  -d, --dot         Allow dot to be treated as a letter. Mostly useful for matching identifiers.

                    Also -. --period.

  -m, --minus       Allow minus to be treated as a letter. Mostly useful for matching
                    identifiers. Also --hyphen or --dash.

  -v, --version     Print the version number.
```

API: Pure Python

Get examples: `python -m tdda.rexpy.examples`

ids.py:

```
from tdda import rexpy

corpus = ['123-AA-971', '12-DQ-802', '198-AA-045', '1-BA-834']
results = rexpy.extract(corpus)
print('Number of regular expressions found: %d' % len(results))
for rex in results:
        print('    ' + rex)
```

RESULTS

```
$ python ids.py
Number of regular expressions found: 1
   ^\d{1,3}\-[A-Z]{2}\-\d{3}$
```

API: Pandas

pandas_ids.py:

```
import pandas as pd

from tdda import rexpy

df = pd.DataFrame({'a3': ["one", "two", pd.np.NaN],
                   'a45': ['three', 'four', 'five']})
re3 = rexpy.pdextract(df['a3'])
re45 = rexpy.pdextract(df['a45'])
re345 = rexpy.pdextract([df['a3'], df['a45']])
print('  re3: %s' % re3)
print(' re45: %s' % re45)
print('re345: %s' % re345)
```

RESULTS

```
$ python pandas_ids.py
  re3: [u'^[a-z]{3}$']
 re45: [u'^[a-z]{4,5}$']
re345: [u'^[a-z]{3,5}$']
```

njr@StochasticSolutions.com

http://tdda.info

https://github.com/tdda

#tdda*

@tdda0 @njr0

* *tweet (DM) us email address for invitation Or email me.*

Correct interpretation: Zero

Error of interpretation: Letter "Oh"

www.tdda.info/pdf/tdda-tutorial-pydata-london-2018.pdf