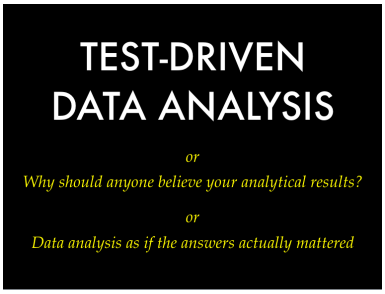
Test-Driven Data Analysis
or: *Why should anyone believe your analytical results?*
or: *Data Analysis as if the answers actually mattered*

Nick Radcliffe

Stochastic Solutions
& Department of Mathematics, University of Edinburgh

PyCon UK 2016
Saturday 17th September 2016



Good morning.

I'm Nick Radcliffe, and this morning my mission is to persuade you that if you analyse data, you should be doing *test-driven* data analysis, which is the title of my talk . . .
. . . though I have some alternative titles as well.

I've been feeling my way towards what I now call test-driven data analysis for about fifteen years, and, with your indulgence, I'm going to start by explaining the journey that brought me to TDDA, before describing what it is and why I think you should adopt it.

Image: Dave Briggs, Flickr, https://www.flickr.com/photos/theclosedcircle/3349126651/

As a teenager, I spent most of my time writing code—mostly machine code—first for the Zilog Z80, and later for the 6502. I spent my school holidays writing database and word processing software for the BBC Micros used by kids at Hertfordshire schools.

And I was good at it. My code was used by tens of thousands of school kids, it ran over a thousand times faster than the code it replaced, while using the same manual and producing identical results, and remarkably few bugs were ever reported against it. In truth, I thought I was pretty hot shit.

I went to university and did a Ph.D. in genetic algorithms and *deep learning* (as we didn't call it) and fell into data analysis. I formed a company, Quadstone, stopped coding, and spent 15 years as the CTO, managing software development and data analysis consulting services.

Around the year 2000, we started trying to solve a new kind of modelling problem—something we called *uplift modelling*—and I found myself writing increasingly detailed pseudocode, which—naturally—led to my starting to write Python.

Writing code is not like riding a bike

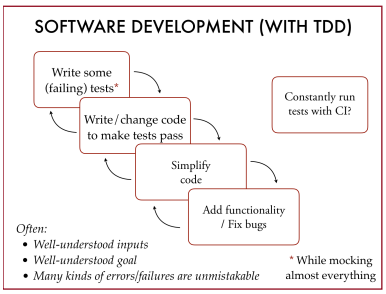
At this point that I made a frightening discovery:

Writing code is not like riding a bike

To my horror, I would write a page of code, run it, and it would turn out to have a dozen syntax errors and, more worryingly, half a dozen logic errors. I had become *terrible* at writing code.

Around this time, I had been reading Tim Bray's blog—*Ongoing*—and he had been proselytising on behalf of something called *test-driven development*, which as I understood it, was all about writing so many and such detailed tests that there was nowhere for bugs to hide, and to run these tests all the time, especially when you change code. So this is what I started doing and continue doing today: I test the bejesus out of my code to create a really inhospitable environment for bugs.

However, because all I did was read a couple of blog posts by Tim Bray, and because I was doing a mixture of writing a data analysis tool, and using it to perform actual data analysis, *my* form of test-driven development was, and is, a little different from orthodox TDD.



So my understanding of the orthodox TDD process is something like this:

- Write some tests (mostly unit tests), possibly mocking everything within an inch of its life.
- Write the code to make the tests pass.
- Stop when then tests pass (and maybe refactor)

Whereas what I did (and do) is much more like:

Don't mock: it's not kind

Why is this lying bastard lying to me?

- Write some code to generate some kind of results.
- Don't mock anything, ever
- Spend a long time carefully checking the results. (A good mindset for this is to imagine that you're Jeremy Paxman and that the numbers you're being asked to believe are being provided by a particularly oleaginous politician.)
- Once I really believe that the generated results are correct, write tests that assert that the reference inputs produce the expected "reference" results. We tend to call larger examples of this *reference* tests.
- Note particularly, that I nearly always write the tests *after* writing the code. Not long after: usually before *committing* any code. But *after*, not *before*.

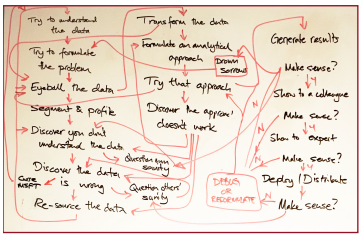
And on this last point, I've never really felt as if I have much choice. Even for quite small inputs, generating the results I want to test against by hand is typically hard enough that the only way I could do it is to write some more code. And while that isn't necessarily crazy, life is too short.

So that's the story of how I came to test-driven development, and to writing tests for analytical software.

Data Analysis

But what about actually *doing* analysis?

Doing analysis tends not to look *anything like* the somewhat-scientific approach of test-driven development. It just doesn't.



Doing data analysis invariably starts with scrabbling around trying and find some vaguely relevant data with a view to answering some question or performing some modelling that itself is not usually very well defined. You get some some messy collection of datasets, almost always without accurate data dictionaries, with poorly defined relationships between them, collected on various inconsistent bases, and usually either zero or half a dozen "unique identifiers" that might or might not act as reliable keys for joining the data to form a coherent whole. And as we try to understand the data and patterns within it, we'll find oddities, and start to question our own sanity, and realise that some of the values are in metric units and some in imperial units, and some of the values aren't real but have been inferred, and that some data is duplicated, while other is missing entirely. And slowly, if we're diligent and lucky, we'll slowly start to piece together an understanding of the data and the domain that don't seem inherently inconsistent, and we'll keep re-sourcing bits of data and asking questions until eventually we can try some kind of analytical approach to whatever problem we are trying to solve. And it just goes round and round and round until either we actually get to a result we believe; or maybe we just run out of time or energy and simply decide whether to go with what we've got or give the whole thing up.

Test-Driven Data Analysis

Shouldn't we do test-driven data analysis?

Now a few years ago, my good friend Patrick Surry said to me:

"We do test-driven developmnet.
So shouldn't we do test-driven data analysis?"

And we both immediately loved the idea. The only problem was, we had very little idea what it could mean. Obviously, today, we would channel our inner Teresa May and assert that "test-driven data analysis *means* test-driven data analysis", but at the time we felt we needed to move beyond tautologies.

Transfer the ideas of test-driven development from software development to data analysis (mutatis mutandis)*

Intuitively, we wanted to take the ideas from TDD in software development and apply them to the process of data analysis, making appropriate adjustments. But what does that actually mean?

Well, for me, the starting point is all about correctness. We do data analysis to understand things, to support better decision making, to optimise things. And it's often OK if our results aren't perfect—which is just as well—but there really isn't any point if we're not going to at least get the sign and sense of the analysis right. And, unfortunately, Sturgeon's Revelation (that's, of course, Theodore, not Nicola Sturgeon), that

STURGEON'S REVELATON

"Ninety per cent of everything is crap"

"ninety per cent of everything is crap"

proves to be as true for data analysis as for everything else.

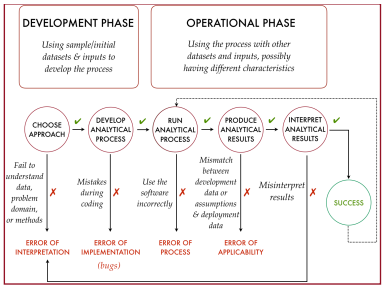
In fact, it's worse for data analysis than for software development, because not only do we need our analytical scripts to be "correct": good data analysis also needs to be concerned with

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.

- correctness and meaningfulness of our analytical processing
- *and* correctness and validity of inputs
- *and* correct interpretation of the outputs (or at least, the clearest possible presentation to minimise the chances of misinterpretation.)

Most of the data analysis processes I see these days look something like this.



We normally usually start with an exploration and development phase during which we explore the data and formulate our approach. During this phase, there are two main things that can go wrong. The first is that we can misunderstand something important, whether it be about the meaning or interpretation of the data, the meaningful manipulations that can be applied to the data, or the domain under study. We call these kinds of mistakes *errors of interpretation,* and they play a much more significant role in data analysis than in most other forms of software development.

Once we've decided on an approach, we have to implement it—whether that's by writing Python code, or R code or simply setting up some analysis in a graphical user interface. Again, it's entirely possibly to make errors at this stage, and we call these *errors of implementation*. These are the errors that do have a direct analogue in software development: these are just bugs.

If we avoid errors in the first two phases, we then have to use the system we have developed to perform the analysis. If our tool is a spreadsheet, or even a Jupyter Notebook, it may be that by the time we have finished "implementing" the analysis has already been performed on the input data, but many cases, there's still something to do here. And at this point, we have further opportunities to make errors by using the analytical system incorrectly—mixing up our inputs, getting the format wrong, forgetting to disable some debugging shortcut, pasting the data into the wrong cells, picking up the wrong output file, failing actually to run the analysis and therefore looking at the last development results etc. We call these kinds of mistakes *errors of process*.

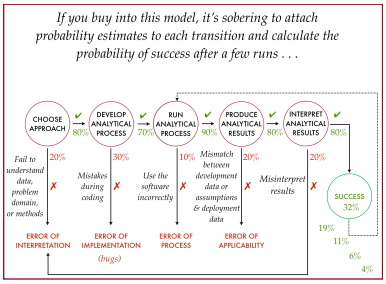
Another thing that can—and frequently does—go wrong here is that the data we actually use for the analysis (if it is not identical to the data used to develop the analysis, even if that was just a subset of the real data) may not correspond closely to data used to build the analytical process. It's normal (and proper) for decisions about parameters and approaches to be strongly influenced by the shape of the development data, and if the data used in production is materially different, it may be that choices we made are not appropriate. We call problems in this area *errors of*

applicability. The software is fine: the data just doesn't meet the analytical assumptions.

Finally, even if everything goes perfectly, we (or the end user) can still misinterpret the results—whether by holding the graph upside down, by not understanding probability, by misunderstanding the scale or whatever.

And a frightening number of these misinterpretation of the answer result in *reversing* the correct course of action—targeting the worst prospects, lending to the worst credit risks, flying the aeroplane when the system said "100%", meaning maximum danger, but we interpreted it as "100% AOK."

And then in most cases, these days, even if the analysis started of as a quick, one-off affair, it ends up being run over and over, on different input data. And of course, all the risks of getting something wrong in the operational deployment phase then arise again, possibly more so because the memory of coding the system is less fresh, or perhaps the process is run by someone else who is less familiar with it.



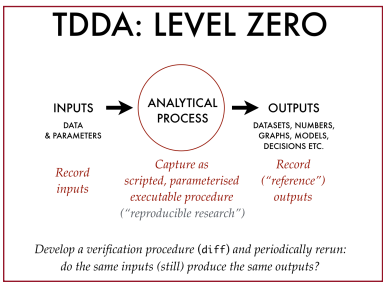
If you buy into the characterisation I've outlined here at all, it can be sobering to attach probability estimates to each phase of the analysis cycle and multiply them out to see how likely you are to get a good result. In this case, with what seem like quite optimistic probabilities to me, we end up with only a 32% chance of the getting the right answer.

And then to multiply through a few times by the over probability of getting the operational phase right, to simulate running the analysis repeatedly. With my numbers, successive runs drop to 19%, 11%, 6%, . . .

So what's to be done?

TDDA Level Zero

The first thing I would suggest, even if you're going to do nothing else, is to capture—to *measure*, if you will—what your analytical process does. So, however you do your analysis, whatever rule or heuristic or procedure you use for deciding when you're done, when you're happy to share or use or believe your results, keep that the same. But capture and record what you've done.



- Capture your inputs (or a useful subset, if the inputs are impractically large)
- Encapsulate your process as a runnable scripts—there's a whole field dedicated to this idea, going by the name *reproducible research*
- Capture your outputs
- And develop some kind of verification procedure that checks that the same inputs with the same analytical process produce the same answer as before.

Now you say—well, how could this fail? If I don't change anything, the results will be the same, right?

Wrong.

You could upgrade your Python, or your operating system, or some packages. Or Autoupdate might do that for you. You could run it on a different system. You could use random numbers in your code. You could have a hardware fault or a software fault. You could have a race condition in your code. Your disk could corrupt your data. Or your source code. Or a helpful colleague could improve the code without your realising. And those are just some of the things that could happen inadvertently.

Even more likely, you could use the code again, on new or updated data. And in doing so, you might fix a bug, or extend things to handle some new values, or an extra field, or a new data format, or a moving distribution. Probably in a way that couldn't *possibly* affect your previous results. And yet . . .

And I have to put in a small further rant against mocking here. Time and again I hear TDD folk say "it's not my responsibility to check that the database works" or that `numpy` or `SciKit Learn` work.

To which I say: true enough.

But it *is* your problem if you don't use database correctly, or if the interface to the data base changes. And it is your problem if a bug in the database causes your analysis to produce the wrong answer. So you have to test that the systems you depend on work correctly in the context of your code and your usage pattern. So maybe software engineers can get away with mocking, but as analyst, your output is analysis, and it's your responsibility to make sure it's correct, even with the imperfect tools the world gives you.

So consider upgrading your system by running this handy command.

```
pip uninstall mock
```

pip uninstall mock

So that's TDDA Level Zero. And of course, you can and should extend it, at least by adding extra tests cases when you use the process and run on new data, especially if it differs in some important way from the original data. Needless to say, if you want to add further, specific tests, that's a good idea too. But don't underestimate the benefits even of just using Level Zero.

We have a subclass of `unittest.TestCase` (`writableunittest.WritableTestCase`) that supports much of this, which we will be making available soon through the `tdda` account on Github.

```
0 :$ python3 test_graphs.py
.....
File check failed.
Compare with "diff /Users/njr/pycon/testoutput/tsr-288-DS-5.svg /Users/njr/pycon/
reference/tsr-288-DS-5.svg".
Note exclusions: Copyright
Compare preprocessed with "diff /tmp/actual-tsr-288-DS-5.svg /tmp/expected-tsr-288-
DS-5.svg".
F
======================================================================
FAIL: testTSRGraphs (__main__.TestGraphs)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_graphs.py", line 111, in testTSRGraphs
    maxPermutationCases=2)
  File "/Users/njr/python/tdda/writabletestcase.py", line 170, in check_file
    self.assertEqual(nFailures, 0)
AssertionError: 1 != 0

----------------------------------------------------------------------
Ran 9 tests in 11.320s

FAILED (failures=1)
```

The output here shows an example of using it to test some graph drawing code. In this case, you can see a single test fails. But rather than just reporting the failure, it tells you exactly what command you can use to see the differences between the files. It also lets you specify lines or part so lines that ought to be ignores, and reports what those are. Here, Copyright lines are ignored. The library has also written cleaned up copies of both the actual and expected results so that you can diff those

without the noise.

```
1 :$ python3 test_graphs.py -W
Written /Users/njr/pycon/reference/pndate-graph.svg.
.Written /Users/njr/pycon/reference/test-pnhbar-graph.svg.
.Written /Users/njr/pycon/reference/pnd-295-SGLD-28.svg.
Written /Users/njr/pycon/reference/pnd-295-SI-1136.svg.
Written /Users/njr/pycon/reference/pnd-295-DS-235.svg.
Written /Users/njr/pycon/reference/pnd-428-BH-44.svg.
Written /Users/njr/pycon/reference/pnd-295-SI-1139.svg.
Written /Users/njr/pycon/reference/pnd-295-SGLD-8.svg.
Written /Users/njr/pycon/reference/pnd-295-BD-97.svg.
Written /Users/njr/pycon/reference/pnd-295-BD-94.svg.
Written /Users/njr/pycon/reference/pnd-428-BH-4.svg.
...
Written /Users/njr/pycon/reference/tsr-273-DS-4.svg.
Written /Users/njr/pycon/reference/tsr-278-DS-4.svg.
Written /Users/njr/pycon/reference/tsr-282-DS-3.svg.
Written /Users/njr/pycon/reference/tsr-287-DS-3.svg.
Written /Users/njr/pycon/reference/tsr-287-FW-3.svg.
Written /Users/njr/pycon/reference/tsr-288-DS-8.svg.
.
.
----------------------------------------------------------------------
Ran 9 tests in 11.039s

OK
```

The library then has a **-W** flag that you can use if and when you convince yourself that the change is not a semantic one to regenerate all the reference results against which comparisons are made

```
0 :$ python3 test_graphs.py
.........
----------------------------------------------------------------------
Ran 9 tests in 11.658s

OK
0 :$
```

And if you run again after that, the tests pass.

TDDA Level One

TDDA LEVEL ONE:

CONSTRAINTS

The second area we've been giving a lot of thought to is constraints and declarations. By this, I mean checks that we can put place in that will quickly detect that something has gone badly wrong even if they can't really tell you why.

The concept is that we develop sets of constraints that should be true of the data at various points—particularly to the input dataset(s) and the results, but also potentially to any—or all—intermediate datasets. And these constraints can be almost anything. They can apply to individual fields:

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"	minVal ≤ medianVal ≤ maxVal
StartDate < tomorrow()	sum(Favourite*) = 1
v < 2.97e10	AlmostEqual(F, m * a, 6)
Height ~ N(1.8, 0.2)	V ≤ H * w * d

- Age should be between 0 and 150
- Age should be an integer
- CustomerID shouldn't be NULL
- Credit Card Number should have 16 digits
- CustomerID should be a valid UUID4
- Velocity should not exceed Speed of Light

and they can apply to whole datasets

- The dataset must contain a field CID
- The dataset must contain exactly 118 records
- Exactly one field should have an "O" tag

or to sets of fields in the dataset

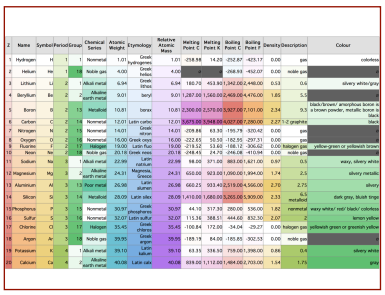
- StartDate ≤ EndDate
- $F = ma$ (to 6 decimal places)

Anyway, constraints are great, and can certainly pick up all sorts of problems, but in truth, who's going to write them? No one is going to write them.

So the second bit of technology we've been developing for TDDA is an automatic constraint discovery (or induction) system. The idea is very simple: you give the TDDA Discovery Process a dataset and it mindlessly spits out facts that are true about that dataset in the form of constraints that you *might* wish to adopt. To be clear, there is nothing smart about the system at all: it doesn't currently try to work out how likely a constraint is to be useful, or to continue holding. It just gives you a starting point on the basis that

(1) something's better than nothing
(2) most people are not going to bother sitting down trying in a set of constraints they expect to be true of the data, but they might be willing to look over a list of suggestions and cull the ones that are obviously ridiculous.

And as a bonus, the constraints the system spits out—and fails to spit out—are often quite revealing.



I'll show you an example.

This is the top of the Periodic Table, a dataset with 118 rows that I pieced together from Wikipedia a few years ago. It's not a great example for TDDA, in that we don't update the Periodic Table materially too often, but it should be familiar to some of you.

```
[2]> discover -l

declare (field-exists "Z")
declare (field-exists "Name")
declare (field-exists "Symbol")
declare (field-exists "Period")
declare (field-exists "Group")
declare (field-exists "ChemicalSeries")
declare (field-exists "AtomicWeight")
declare (field-exists "Etymology")
declare (field-exists "RelativeAtomicMass")
declare (field-exists "MeltingPointC")
declare (field-exists "MeltingPointKelvin")
declare (field-exists "BoilingPointC")
declare (field-exists "BoilingPointF")
declare (field-exists "Density")
declare (field-exists "Description")
declare (field-exists "Colour")
declare (= (length (field-names)) 16)
```

Our analysis software—Miró—has a command called **discover**, and when you use this it simply runs over a dataset discovering facts about them and expressing them as constraints using a Lisp-like expression language.

So here, it first declares that all the fields it fields are in the dataset, and that there are 16 of them.

```
if (field-exists "Z")
   declare (= (type Z) "int")
   declare (>= (min Z) 1)
   declare (<= (max Z) 118)
   declare (= (countnull Z) 0)
   declare (non-nulls-unique Z)
fi

if (field-exists "Name")
   declare (= (type Name) "string")
   declare (>= (min (length Name)) 3)
   declare (<= (min (length Name)) 13)
   declare (= (countnull Name) 0)
   declare (non-nulls-unique Name)
fi

if (field-exists "Symbol")
   declare (= (type Symbol) "string")
   declare (>= (min (length Symbol)) 1)
   declare (<= (min (length Symbol)) 3)
   declare (= (countnull Symbol) 0)
   declare (non-nulls-unique Symbol)
fi
```

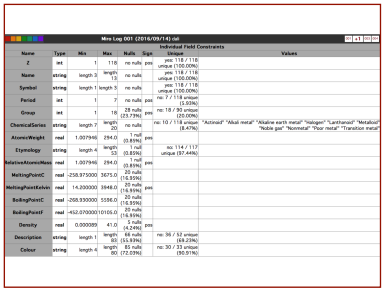
Next, it goes through and makes specific declarations about each field. So here, the atomic number, Z, is an integer between 1 and 118, it has no nulls and each value is unique. And those might or might no be constraints you want. If a new periodic does come along, one possibilities certainly that it will add element 119—probably ununnonium—but equally, you might want to know about that if it does happen, so maybe you'd like a warning or an error.

```
if (field-exists "Period")
   declare (= (type Period) "int")
   declare (>= (min Period) 1)
   declare (<= (max Period) 7)
   declare (= (countnull Period) 0)
fi

if (field-exists "Group")
   declare (= (type Group) "int")
   declare (>= (min Group) 1)
   declare (<= (max Group) 18)
fi

if (field-exists "ChemicalSeries")
   declare (= (type ChemicalSeries) "string")
   declare (>= (min (length ChemicalSeries)) 7)
   declare (<= (min (length ChemicalSeries)) 20)
   declare (= (countnull ChemicalSeries) 0)
   declare (= (countzero (or (isnull ChemicalSeries) (in ChemicalSeries (list
"Actinoid" "Alkali metal" "Alkaline earth metal" "Halogen" "Lanthanoid"
"Metalloid" "Noble gas" "Nonmetal" "Poor metal" "Transition metal")))) 0)
fi
```

It's a pretty mindless procedure, but it does try to spot things like when a string field has a small cardinality and suggest limiting values to those it sees, like for the Chemical Series here.



It also spits out a nice tabular summary and can save the results both as this series of executable lisp-like statements (that our software understands), but also as a JSON file that almost anything could consume. It generates multi-field and dataset constraints too, though I haven't shown those here.

Right now, this functionality is only in our software, but over the next few months we plan to release a version at least for Pandas, and possibly beyond that. Watch the Github account for **tdda**.

CONSTRAINT DISCOVERY

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.

The other important thing to note is that constraints the system *doesn't* spit out can be just as important and useful as ones it does. Oh: my customer ID isn't unique in the customer table. And it does contain missing values. That's not good . . .

Conclusion

So thank you for listening.

To recap, test-driven data analysis is more of an idea than a reality at the moment, with its guiding principle being to take the ideas from test-driven development and to adapt them to help increase the likelihood of getting our data analysis correct.

Thus far, we have two concrete suggestions for how to do this. The *sine qua non*—Level Zero—is the reference test, by which we mean using the ideas from reproducible research to encapsulate an analytical process in an executable form, to capture one or more collections of inputs and outputs and to develop a way of testing that the inputs, when passed through the analytical process, produce the intended outputs. In terms of tooling, we can begin to offer support for this with extensions to Python's `unittest` module that help with useful semantic comparisons and reference output re-writing. There is a version on Github now, but the only documentation is in the form of some posts on the TDDA blog about Apple Health Data. I'll update **writableunittest** in the next week or so and document it then tweet about it from **@tdda0** probably blog about it too.

The next idea—Level One—is to use constraints to assert properties of input, output and perhaps intermediate datasets that should always be true. In terms of tooling, we have developed for our software, Miró, a discovery process that can suggest constraints that are true within an example dataset, and the ability to record these in executable, testable form. We fully intend to extend these to at least Pandas and perhaps Postgres and make these available on a permissive license over the next few months.

I hope you find these ideas useful. You can read more and pick stuff up from all the places listed below.

In the unlikely event that I've comfortably met the constraint of the time allocated to me, I'll be happy to take any questions, now or throughout the rest of the conference.

Thank you.