No More Loopy Code

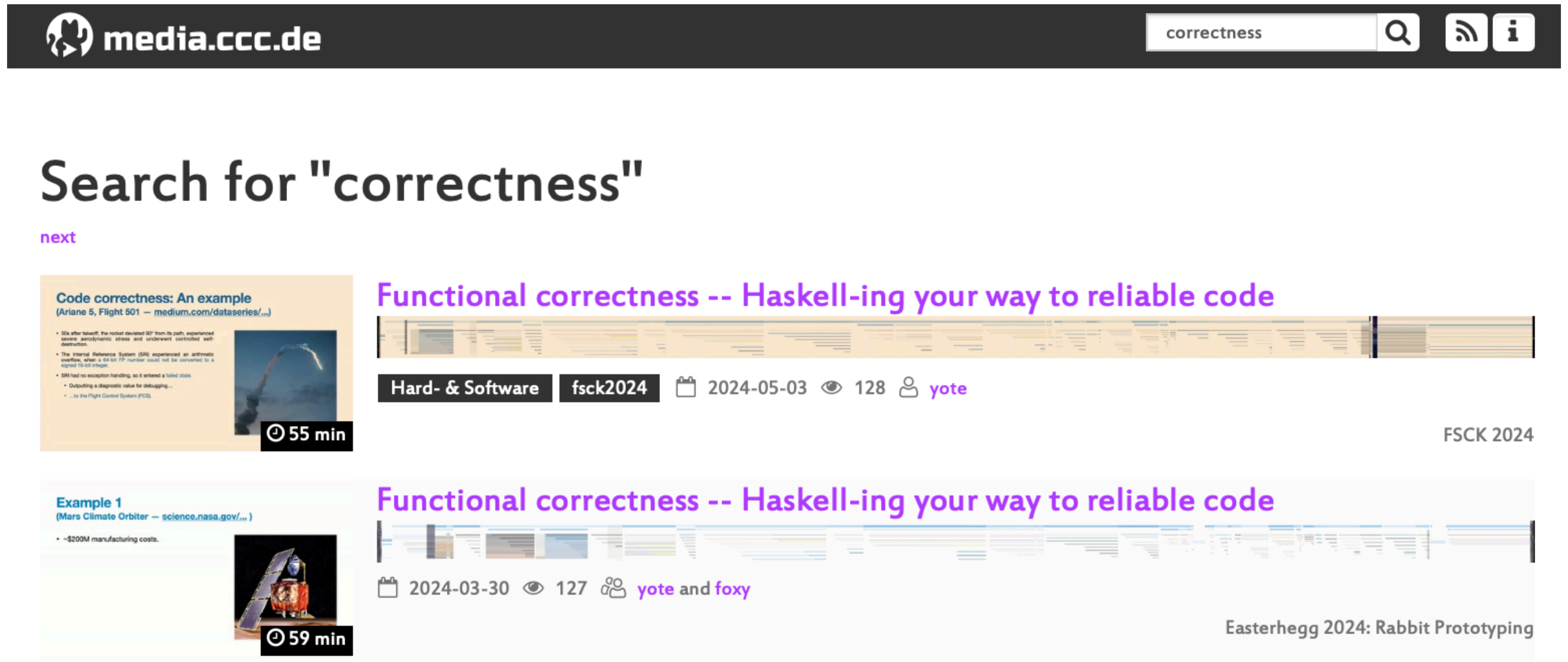
Data Science Goes Functional

yote

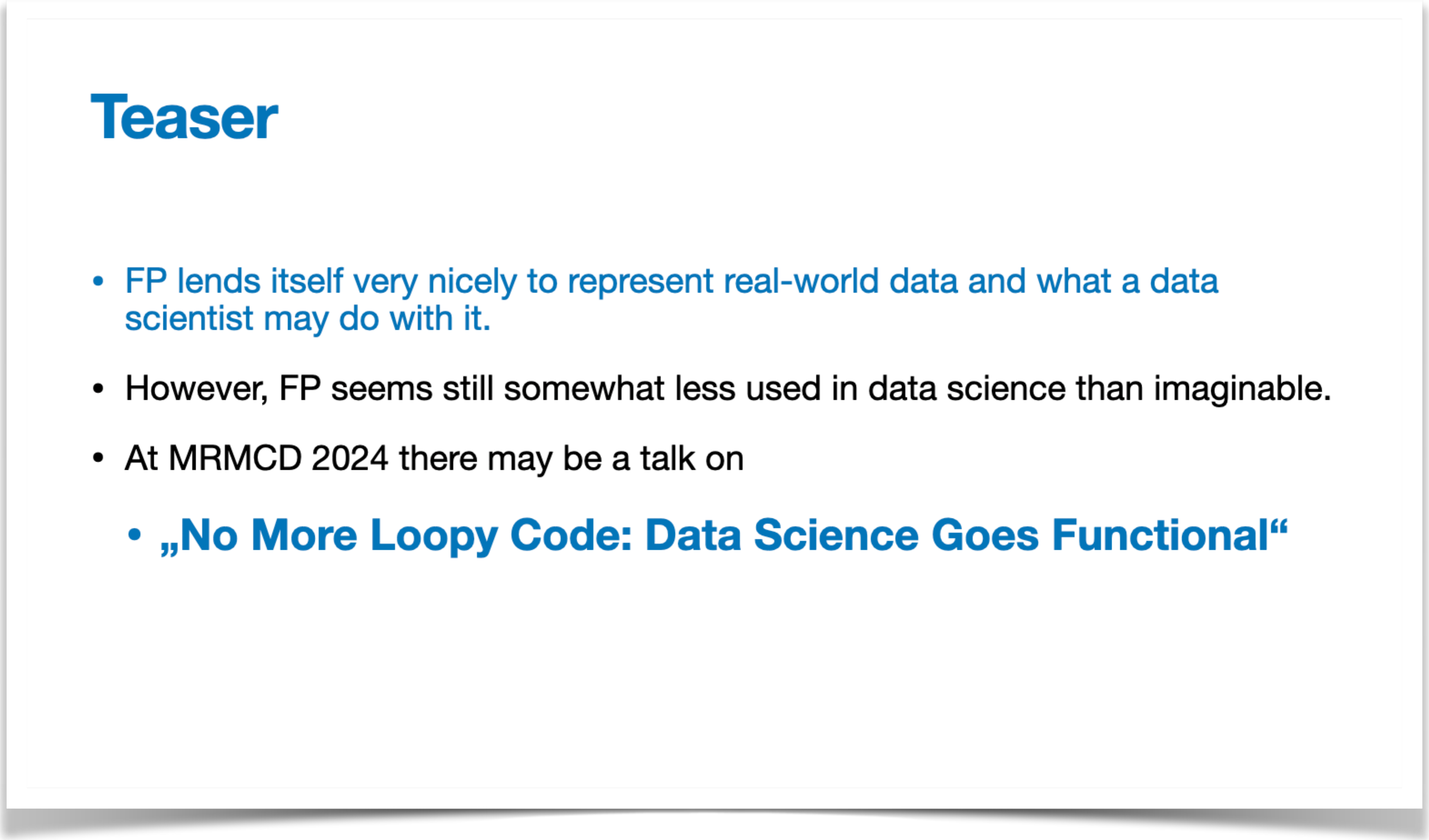
whoami: yote

- Biologist turned biomathematician
  - Great interest in provably correct code and functional programming
  - Does „something with data“ in industry and for nonprofits.
  - Lots of coding, but no „proper“ SE practice

How did we get here?



Hackover still missing :(


Teaser

- FP lends itself very nicely to represent real-world data and what a data scientist may do with it.
- However, FP seems still somewhat less used in data science than imaginable.
- At MRMCD 2024 there may be a talk on
  - „No More Loopy Code: Data Science Goes Functional“

Data analyses play a crucial role in society…

- No part of modern life is detached from data and their analysis:
  - Governance and policy-making
  - Healthcare
  - Business
  - …

…yet, they can be ridiculously wrong.

American Economic Association
https://www.aeaweb.org › articles › aer.100.2.573

Growth in a Time of Debt

by CM Reinhart · 2010 · Cited by 5435 — Growth in a Time of Debt by Carmen M. **Reinhart** and Kenneth S. **Rogoff**. Published in volume 100, issue 2, pages 573-78 of American Economic…

…yet, they can be ridiculously wrong.

In this paper, we exploit a new multi-country historical dataset on public (government) debt to search for a systemic relationship between high public debt levels, growth and inflation.[1] Our main result is that whereas the link between growth and debt seems relatively weak at "normal" debt levels, median growth rates for countries with public debt over roughly 90 percent of GDP are about one percent lower than otherwise; average (mean) growth rates are several percent lower. Surprisingly, the relationship between public debt and growth is remarkably similar across emerging markets and advanced economies. This is not the case for inflation. We

◇	B	C	I	J	K	L	M
2			Real GDP growth				
3			Debt/GDP				
4	Country	Coverage	30 or less	30 to 60	60 to 90	90 or above	30 or less
26			3.7	3.0	3.5	1.7	5.5
27	Minimum		1.6	0.3	1.3	-1.8	0.8
28	Maximum		5.4	4.9	10.2	3.6	13.3
29							
30	US	1946-2009	n.a.	3.4	3.3	-2.0	n.a.
31	UK	1946-2009	n.a.	2.4	2.5	2.4	n.a.
32	Sweden	1946-2009	3.6	2.9	2.7	n.a.	6.3
33	Spain	1946-2009	1.5	3.4	4.2	n.a.	9.9
34	Portugal	1952-2009	4.8	2.5	0.3	n.a.	7.9
35	New Zealand	1948-2009	2.5	2.9	3.9	-7.9	2.6
36	Netherlands	1956-2009	4.1	2.7	1.1	n.a.	6.4
37	Norway	1947-2009	3.4	5.1	n.a.	n.a.	5.4
38	Japan	1946-2009	7.0	4.0	1.0	0.7	7.0
39	Italy	1951-2009	5.4	2.1	1.8	1.0	5.6
40	Ireland	1948-2009	4.4	4.5	4.0	2.4	2.9
41	Greece	1970-2009	4.0	0.3	2.7	2.9	13.3
42	Germany	1946-2009	3.9	[illegible]	n.a.	n.a.	3.2
43	France	1949-2009	4.9	[illegible]	3.0	n.a.	5.2
44	Finland	1946-2009	[illegible]	[illegible]	5.5	n.a.	7.0
45	Denmark	1950-2009	[illegible]	[illegible]	2.4	n.a.	5.6
46	Canada	1951-2009	1.9	[illegible]	4.1	n.a.	2.2
47	Belgium	1947-2009	n.a.	[illegible]	3.1	2.6	n.a.
48	Austria	1948-2009	5.2	3.3	-3.8	n.a.	5.7
49	Australia	1951-2009	3.2	4.9	4.0	n.a.	5.9
50							
51			4.1	2.8	2.8	=AVERAGE(L30:L44)	

Enter Functional Programming (FP)

- Obligatory disclaimer: Writing robust analytical code is not 1:1 equivalent to coding in a functional style.
  - However, it makes certain parts of it easier.
- I will talk about general FP concepts, …
  - … which you can also use in (nearly) every other language.

Enter Functional Programming

Contents hide

Functional programming

52 languages

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can. This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

Functional programming is sometimes treated as synonymous with purely functional programming, a subset of functional programming which treats all functions as deterministic mathematical functions, or pure functions. When a pure function is called with some given arguments, it will always return the same result, and cannot be affected by any mutable state or other side effects. This is in contrast with impure procedures, common in imperative programming, which can have side effects (such as modifying the program's state or taking input from a user). Proponents of purely functional

TL;DR?

Lots of things that make our life easier!

- Small and pure functions, i.e. without mutable state or side-effects
    - Ideally in a declarative style instead of imperative „step-by-step“ one
    - Both makes it easier to verify „by inspection“
- Function application and composition in a modular manner („bottom-up approach“)
    - This allows again for a declarative style…
    - … and to reason more easily about how different parts of the code interact.
    - This also touches the idea of functions being „first-class citizens“

1. Functions

Functions, example 1

`isPalindrome`

Python (rather naive):

```
def isPalindrome(s: str) -> bool:
    n = len(s)
    for i in range(n):
        if s[i] != s[n-i-1]:
            return False
    return True
```

Pure

Small-ish…

Easy to verify? Off-by-one errors…

Mutable state? Technically yes.

Functions, example 1
`isPalindrome`

Python (FP):

```
def isPalindrome(s: str) -> bool:
    return s == s[::-1]
```

Small.

Pure.

Declarative. Like a mathematical definition.

Thus easy to verify by inspection (if you know Python's list slicing)

No mutable state anymore!

Why bother?

- Small functions are easier to think about and verify, as well as test and debug.
  - Pure functions even more so.
  - Avoiding mutable state makes it even easier.
- All of this becomes especially important if we want to perform more complex numerical calculations.
  - You wouldn’t believe what’s in some „scientific code.“

2. Function application

Function application

- Application: Applying a function to arguments.
    - Put differently: Calling a function with arguments*.
    - Data Science is all about „doing something with data“, i.e. applying functions.

$$f(x) = 3*x$$

$$f(5) = 3*5 = 15$$

Function application, example 1

calculateVAT

Python (naive):

```
# prices = [1.99, 2.95, 0.95, 2.55]

VATs = []
for price in prices:
    current_VAT = 0.19 * price
    VATs.append(current_VAT)

# VATs = [0.3781, 0.5605, 0.1805, 0.4845]
```

Function application, example 1

`calculateVAT` using `map`

Python (FP):

```
# prices = [1.99, 2.95, 0.95, 2.55]

VATs = list(map(lambda x: 0.19 * x, prices))

# VATs = [0.3781, 0.5605, 0.1805, 0.4845]
```

- map is a cornerstone of FP: We apply a function to all elements of a list.
    - The function is provided as a parameter („higher-order function“).
- This is trivially and automatically parallelisable.

Function application, example 2

getSmallElements

- We want to select those prices that are „small“ — e.g. less than 2.00€.
- Python (naive):

```
# prices = [1.99, 2.95, 0.95, 2.55]

smalls = []
for price in prices:
    if price < 2.00:
        smalls.append(current_VAT)

# smalls = [1.99, 0.95]
```

Function application, example 2

`getSmallElements` using `filter`

Python (FP):

```
# prices = [1.99, 2.95, 0.95, 2.55]

small = list(filter(lambda x: x < 2.00, prices))
```

- filter is another cornerstone of functional programming.
  - We only want to retain those elements that fulfill a given „predicate“.
  - The predicate is provided as a parameter („higher-order function“).
- This is also trivially and automatically parallelisable.

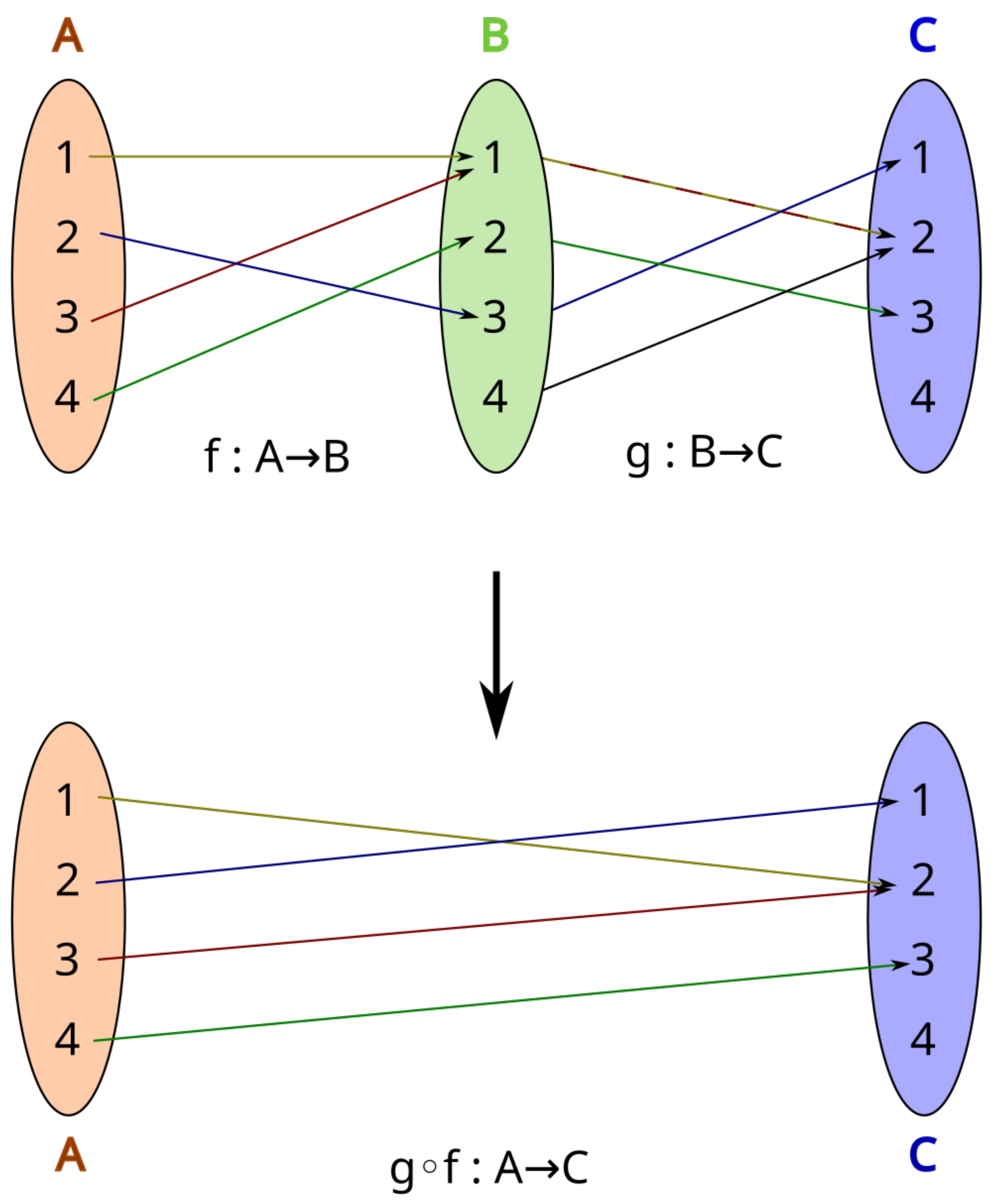
Why bother?

- In data science, we select and transform elements all the time.
- Abstracting these manual processes to simple calls to `filter` and `map` saves repetitive code and makes meaning / intention more clear.
  - This becomes especially apparent once we chain together multiple steps.
- Automatic parallelisation makes it easier to deal with larger datasets.

3. Function composition

Function composition

- Composition: Stringing together multiple functions to a new one.
    - Data science relies heavily on chained transformations.
    - This avoids having mutable state inbetween („df1, df2, df3, etc“).
- (A good type system can catch many errors related to these already at compile time. A strong / expressive type system even more.)



(Image: Stephan Kulla)

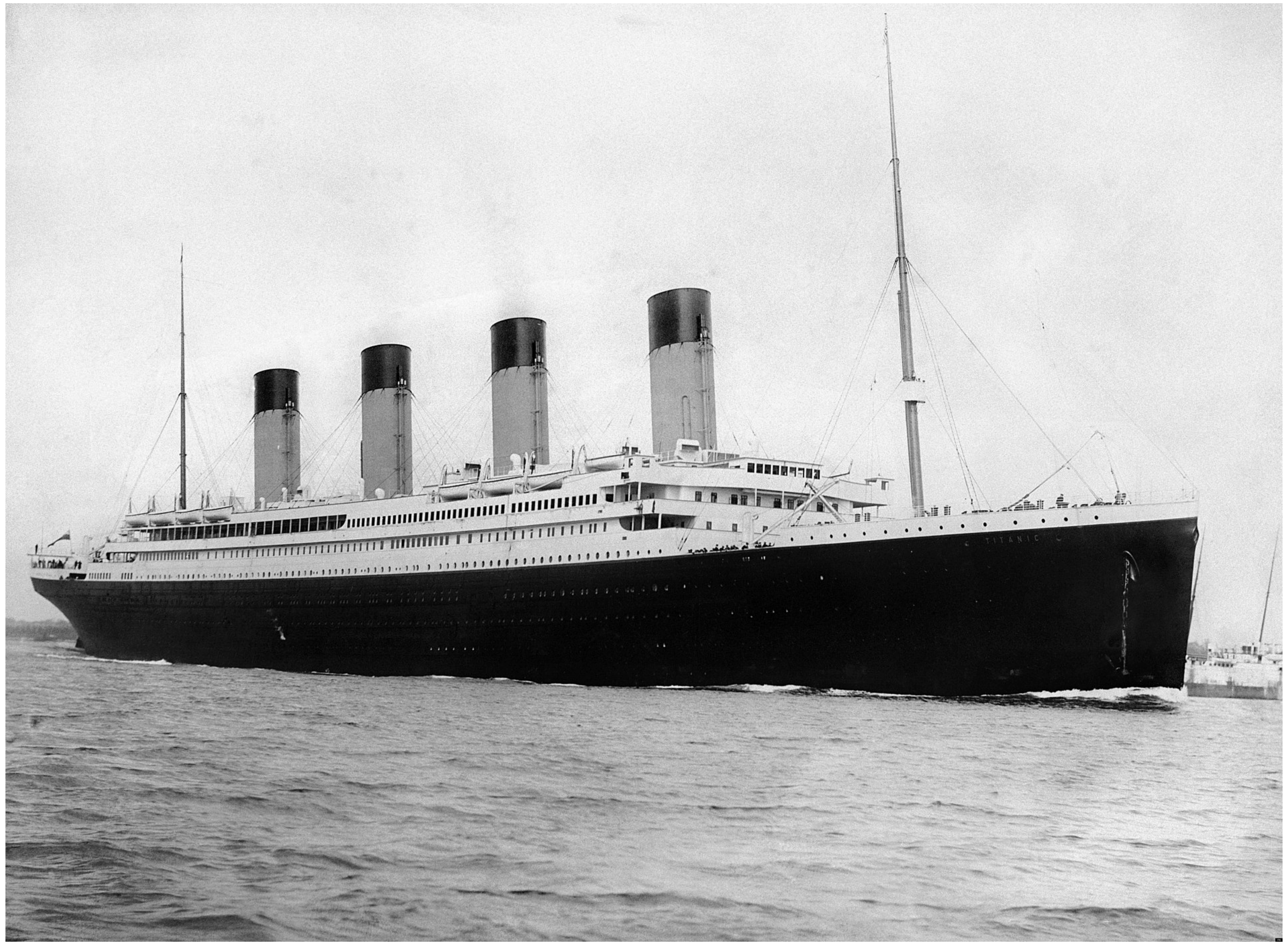
Function composition

- Native Python sadly does not have a built-in operator for this.
  - You can write `some_result = fun3(fun2(fun1(42)))`.
  - But not `some_result = (fun3.fun2.fun1) 42` like in Haskell.
  - Or like `fun1 %>% fun2 %>% fun3` in R.
- As much as I love Python (and don't really love R), I think R syntax will be a bit easier to understand.
  - Hence, let's look at some examples in R!

4. Practical examples

A nice toy dataset

Survival data of the passengers of the RMS Titanic.



(Image: Francis Godolphin Osbourne Stuart)

Let's inspect the dataset first.

```
> head(titanic)
  PassengerId Survived Pclass                                                Name    Sex Age SibSp Parch           Ticket    Fare Cabin Embarked
1           1        0      3                             Braund, Mr. Owen Harris   male  22     1     0        A/5 21171  7.2500              S
2           2        1      1 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38     1     0         PC 17599 71.2833   C85        C
3           3        1      3                              Heikkinen, Miss. Laina female  26     0     0 STON/O2. 3101282  7.9250              S
4           4        1      1        Futrelle, Mrs. Jacques Heath (Lily May Peel) female  35     1     0           113803 53.1000  C123        S
5           5        0      3                            Allen, Mr. William Henry   male  35     0     0           373450  8.0500              S
6           6        0      3                                    Moran, Mr. James   male  NA     0     0           330877  8.4583              Q
> 
```

Let’s come up with a question.

Ideally, we should have started with a question, but whatever.

- Does survival differ between men and women? („Women and children first!“)
  - However, there are different classes, which may confound our results.
    - For now, let’s just focus on the first class.
  - Children vs. adult may confound our results as well.
    - For now, let’s only look at adults.
  - Age may also affect survival, but analysing this is slightly more involved.
    - Let’s postpone that for now :).

We chain together our pipeline accordingly.

Look Ma, no mutable state!

```
> titanic %>%
+   filter(Pclass == 1) %>%
+   filter(Age > 21) %>%
+   select(c("Sex", "Survived"))
      Sex Survived
1  female        1
2  female        1
3    male        0
4  female        1
5    male        1
6    male        0
7    male        0
8    male        0
9  female        1
10   male        0
```

```
> titanic %>%
+   filter(Pclass == 1) %>%
+   filter(Age > 21) %>%
+   select(c("Sex", "Survived")) %>%
+   table()
        Survived
Sex       0  1
  female  2 67
  male   57 36
```

Let's think again about age.

- We only consider men now, as we saw a huge influence of sex on survival.
- This time, we'll use data from all three classes, because we assume that the effect of sex on survival should not differ too much between classes.
- We'll also divide Age into intervals of length 10 to remove some noise.

Indeed, age seems to affect survival.

Behind `mutate`, there's nothing but a `map` 🤫 .

```
> titanic %>%
+   filter(Sex == "male") %>%
+   mutate(Agecut = cut(Age, breaks = seq(0, 100, 10))) %>%
+   select(c("Agecut", "Survived"))
    Agecut Survived
1  (20,30]        0
2  (30,40]        0
3     <NA>        0
4  (50,60]        0
5   (0,10]        0
6  (10,20]        0
7  (30,40]        0
8   (0,10]        0
9     <NA>        1
10 (30,40]        0
11 (30,40]        1
12 (20,30]        1
13    <NA>        0
```

```
> titanic %>%
+   filter(Sex == "male") %>%
+   mutate(Agecut = cut(Age, breaks = seq(0, 100, 10))) %>%
+   select(c("Agecut", "Survived")) %>%
+   table()
          Survived
Agecut       0   1
  (0,10]    14  19
  (10,20]   59  10
  (20,30]  126  23
  (30,40]   77  23
  (40,50]   43  12
  (50,60]   24   4
  (60,70]   13   1
  (70,80]    4   1
  (80,90]    0   0
  (90,100]   0   0
>
```

Finally, let's calculate survival probability.

No state, no side-effects, no problem!

```
> titanic %>%
+   filter(Sex == "male") %>%
+   mutate(Agecut = cut(Age, breaks = seq(0, 100, 10))) %>%
+   group_by(Agecut) %>%
+   summarise(Surv_Prob = mean(Survived))
# A tibble: 9 x 2
  Agecut   Surv_Prob
  <fct>        <dbl>
1 (0,10]      0.576
2 (10,20]     0.145
3 (20,30]     0.154
4 (30,40]     0.23
5 (40,50]     0.218
6 (50,60]     0.143
7 (60,70]     0.0714
8 (70,80]     0.2
9 <NA>        0.129
>
```

Wrap-Up

FP lends itself nicely for data science.

- Small, pure, declarative functions are easy to verify, test, and debug.
- Avoiding mutable state in memory makes it harder to lose track and confuse yourself along the way.
- Typical data science workflows can often be understood as subsequent steps of selecting, transforming and ultimately aggregating data.

But FP is not a magic bullet.

- Steep learning curve in the beginning, especially for people used to imperative.
    - However, I would argue it’s well worth it.
- You’ll always have some side-effects if you interact with the real world.
    - E.g.: reading from or writing to a database.
    - But separating pure and impure parts goes a long way.
- High performance is possible, but sometimes requires some tweaking.

Conclusions

- FP makes certain errors harder, but is not a magic bullet.
  - Finding a balance between FP vs. pragmatic imperative approaches, and getting a feel for which is better when.
- The concepts we discussed can be useful in (nearly) all languages.
- **My recommendation: Take it slow, gradually steal those ideas that seem useful to you. There is no need to instantly commit 100%.**

Thanks for your attention!

- Questions always very welcome!
- Feel free to hit me up:
  - On Telegram: @GermanCoyote
  - On Matrix: @yote:catgirl.cloud
  - And of course in person…
- Feedback: http://nook-luebeck.de/feedback#funktionaler-code-fuer-data-science